|  |
| --- |
| caTissue Suite v1.2 |
| Design document |

|  |
| --- |
| Washington University / Persistent systems  8/16/2010 |

**REVISION HISTORY**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** | **Approved/ Reviewed By** | **Status** |
| 15-Aug-10 | 0.1 | caTissue Suite v1.2 design document | Washington university and PSL |  | Final |

Contents

[Chapter 1. Introduction 7](#_Toc269914325)

[1.1 Audience 7](#_Toc269914326)

[1.2 References 7](#_Toc269914327)

[Chapter 2. Custom identity provider support 8](#_Toc269914328)

[2.1 Design scope 8](#_Toc269914329)

[2.2 Solution architecture 8](#_Toc269914330)

[2.2.1 Detailed design 8](#_Toc269914331)

[2.2.2 Adding a new IDP to the caTissue application 14](#_Toc269914332)

[2.3 I/O considerations 15](#_Toc269914333)

[2.4 Memory considerations 15](#_Toc269914334)

[2.5 Third party components 15](#_Toc269914335)

[2.6 Crash recovery considerations 15](#_Toc269914336)

[2.7 Transaction management 16](#_Toc269914337)

[2.8 Multi threading considerations 16](#_Toc269914338)

[2.9 Configuration / script changes 16](#_Toc269914339)

[2.10 Security considerations 16](#_Toc269914340)

[2.11 Database changes 16](#_Toc269914341)

[Chapter 3. Bulk operation support 18](#_Toc269914342)

[3.1 Design scope 18](#_Toc269914343)

[3.2 Solution architecture 18](#_Toc269914344)

[3.2.1 Detailed design 19](#_Toc269914345)

[3.2.1.1 Bulk operation algorithm 22](#_Toc269914346)

[3.3 I/O considerations 22](#_Toc269914347)

[3.4 Memory considerations 23](#_Toc269914348)

[3.5 Third party components 23](#_Toc269914349)

[3.6 Crash recovery considerations 23](#_Toc269914350)

[3.7 Transaction management 23](#_Toc269914351)

[3.8 Multi threading considerations 23](#_Toc269914352)

[3.9 Configuration / script changes 23](#_Toc269914353)

[3.10 Security considerations 24](#_Toc269914354)

[3.11 Database changes 24](#_Toc269914355)

[Chapter 4. CaTissue-CDMS integration 25](#_Toc269914356)

[4.1 Design scope 25](#_Toc269914357)

[4.2 Solution architecture 25](#_Toc269914358)

[4.2.1 Detailed design 25](#_Toc269914359)

[4.3 I/O considerations 27](#_Toc269914360)

[4.4 Memory considerations 27](#_Toc269914361)

[4.5 Third party components 27](#_Toc269914362)

[4.6 Crash recovery considerations 28](#_Toc269914363)

[4.7 Transaction management 28](#_Toc269914364)

[4.8 Multi threading considerations 28](#_Toc269914365)

[4.9 Configuration / script changes 28](#_Toc269914366)

[4.10 Security considerations 28](#_Toc269914367)

[4.11 Database changes 28](#_Toc269914368)

[Chapter 5. Template based label generator 29](#_Toc269914369)

[5.1 Design scope 29](#_Toc269914370)

[5.2 Solution architecture 29](#_Toc269914371)

[5.2.1 Detailed design 29](#_Toc269914372)

[5.2.2 Label generator hierarchy 30](#_Toc269914373)

[5.3 Template based label generator implementation 31](#_Toc269914374)

[5.3.1 Steps to generate the label 33](#_Toc269914375)

[5.4 I/O considerations 33](#_Toc269914376)

[5.5 Memory considerations 33](#_Toc269914377)

[5.6 Third party components 33](#_Toc269914378)

[5.7 Crash recovery considerations 33](#_Toc269914379)

[5.8 Transaction management 33](#_Toc269914380)

[5.9 Multi threading considerations 33](#_Toc269914381)

[5.10 Configuration / script changes 33](#_Toc269914382)

[5.11 Security considerations 33](#_Toc269914383)

[5.12 Database changes 34](#_Toc269914384)

[Chapter 6. Keyword search 35](#_Toc269914385)

[6.1 Design scope 35](#_Toc269914386)

[6.2 Solution architecture 35](#_Toc269914387)

[6.2.1 Detailed design 35](#_Toc269914388)

[6.3 I/O considerations 38](#_Toc269914389)

[6.4 Memory considerations 39](#_Toc269914390)

[6.5 Third party components 39](#_Toc269914391)

[6.6 Crash recovery considerations 39](#_Toc269914392)

[6.7 Transaction management 39](#_Toc269914393)

[6.8 Multi threading considerations 39](#_Toc269914394)

[6.9 Configuration / script changes 39](#_Toc269914395)

[6.10 Security considerations 39](#_Toc269914396)

[6.11 Database changes 39](#_Toc269914397)

[Chapter 7. Single sign on implementation (SSO) 40](#_Toc269914398)

[7.1 Design scope 40](#_Toc269914399)

[7.2 Solution architecture 40](#_Toc269914400)

[7.2.1 Detailed design 40](#_Toc269914401)

[7.2.2 Authentication using CAS 42](#_Toc269914402)

[7.2.3 Integration with other applications for SSO 42](#_Toc269914403)

[7.3 I/O considerations 42](#_Toc269914404)

[7.4 Memory considerations 43](#_Toc269914405)

[7.5 Third party components 43](#_Toc269914406)

[7.6 Crash recovery considerations 43](#_Toc269914407)

[7.7 Transaction management 43](#_Toc269914408)

[7.8 Multi threading considerations 43](#_Toc269914409)

[7.9 Configuration / script changes 43](#_Toc269914410)

[7.10 Security considerations 43](#_Toc269914411)

[7.11 Database changes 43](#_Toc269914412)

**Figures**

[Figure 1: IDP feature class diagram 9](#_Toc269748199)

[Figure 2: IDP UML representation 10](#_Toc269748200)

[Figure 3: IDP Migration class diagram 12](#_Toc269748201)

[Figure 4: Database changes for IDP 15](#_Toc269748202)

[Figure 5: Bulk operation controller class 17](#_Toc269748203)

[Figure 6: Bulk operation processor class diagram 19](#_Toc269748204)

[Figure 7: Bulk operation database changes 22](#_Toc269748205)

[Figure 8: caTissue- CDMS integration caCore API 24](#_Toc269748206)

[Figure 9: Class diagram for the caTissue-CDMS integrator 24](#_Toc269748207)

[Figure 10: New caCore API in caTissue for CDMS integration 25](#_Toc269748208)

[Figure 11: Label generator hierarchy 29](#_Toc269748209)

[Figure 12: Class diagram to represent tokens 30](#_Toc269748210)

[Figure 13: Class diagram for lucene database indexer 34](#_Toc269748211)

[Figure 14: Class diagrarm for keyword searcher 35](#_Toc269748212)

[Figure 15: Keyword index refresher 36](#_Toc269748213)

[Figure 16: SSO component / sequence diagram 39](#_Toc269748214)

[Figure 17: CAS Authentication handler class diagram 40](#_Toc269748215)

# Introduction

The caTissue v1.2 release contains few new features e.g. support for custom identity provider, ability to perform bulk operations in caTissue etc. This document explains the design for these features. A brief introduction is provided for each section. But to understand these features in details, please refer to the requirements document mentioned below in the references section.

## Audience

* Existing adopters
* Developers
* All stakeholders

## References

* [caTissue Suite v1.2 Share Point](http://cbmiportal.wustl.edu/projects/caTissue/default.aspx)
* [caTissue Suite v1.2 requirements document](http://cbmiportal.wustl.edu/projects/caTissue/Shared%20Documents/Forms/AllItems.aspx?RootFolder=%2fprojects%2fcaTissue%2fShared%20Documents%2f2.%20Use%20case%20and%20Requirements&FolderCTID=&View=%7b44C05BA2-81CC-4E9D-B0AA-61ECD4162FD5%7d)
* [caTissue Suite technical guide](http://cbmiportal.wustl.edu/projects/caTissue/Shared%20Documents/Forms/AllItems.aspx?RootFolder=%2fprojects%2fcaTissue%2fShared%20Documents%2f4.%20Guides%2fTechnical%20Guide&FolderCTID=&View=%7b44C05BA2-81CC-4E9D-B0AA-61ECD4162FD5%7d)
* [caTissue Suite user manual](http://cbmiportal.wustl.edu/projects/caTissue/Shared%20Documents/Forms/AllItems.aspx?RootFolder=%2fprojects%2fcaTissue%2fShared%20Documents%2f4.%20Guides%2fUser%20Manual&FolderCTID=&View=%7b44C05BA2-81CC-4E9D-B0AA-61ECD4162FD5%7d)

# Custom identity provider support

Traditionally, caTissue has supported only CSM based authentication of its users. There have been demands to integrate the application with the organization’s LDAP authentication server or other such legacy / custom authentication mechanisms. So an enhancement to integrate with custom identity provider (authentication and identity authorization mechanism) has been provided in the caTissue Suite.

With this enhancement, different organizations can integrate caTissue with their own systems to authenticate users. However the CSM based authentication is supported as the default choice. This enhancement ensures that if the institute already has the infrastructure for the identity providers, there is no need to replicate all the user data again for caTissue Suite and the existing infrastructure can be reused.

Currently, along with the CSM based authentication, support for authentication using LDAP server is also provided. The framework can be easily extended for new types of IDPs.

## Design scope

Following are the brief requirements that are in scope for the design of IDP support feature.

* Ability to integrate caTissue user authentication with customized identity providers like organization specific LDAP server etc.
* Ability to migrate users from traditional CSM data source to the custom IDPs
* Ability to change the IDP at any time in production with minimal code changes.

## Solution architecture

This section explains the architecture that is followed to implement IDP feature. First the design approach is explained and it is then supported by the detailed architecture diagrams and other class diagrams.

## Detailed design

The most crucial part of the custom IDP support is to externalize the actual authentication mechanism. That means that the actual implementation class has to be configurable so that the organization that wants to integrate caTissue with its own server can implement its own authentication algorithm. The best way to achieve this is to have one common java interface and allow the organizations to configure the actual implementation class in a properties file. Then caTissue application can pick up the configured class and invoke the authentication method. Same approach can be followed for migration of user accounts from one IDP to other.

Following is the class diagram using which the above approach is explained in details below

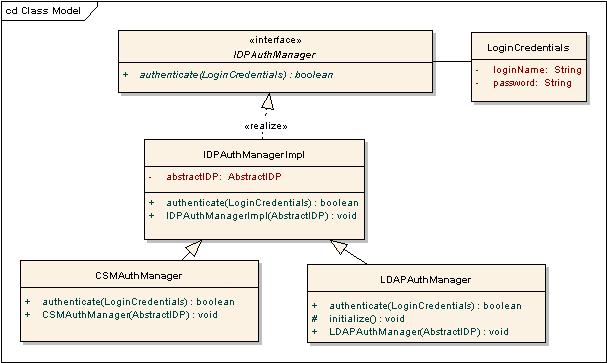


Figure 1: IDP feature class diagram

As shown in the diagram above, IDPAuthManager is the interface which needs to be implemented by all the implementations of authentication managers. Above diagram represents two default managers that are provided in the caTissue application.

1. CSMAuthManager : Provides authentication mechanism for CSM based systems (default)
2. LDAPAuthManager: Provides LDAP authentication support for caTissue.

IDPAuthManager interface has following method that is used to authenticate caTissue users

authenticate (String username, String password)

This method takes user name and password and then internally checks the credentials against the user’s identity provider and returns true if the credentials are valid or otherwise false.

IDPAuthManagerImpl class is an abstract implementation of IDPAuthManager interface. This is the base class for all authentication manger classes. It contains implementation of all the common methods. Also it has a constructor which takes in an instance of AbstractIDP class.

AbstractIDP class is a UML representation of the actual IDP. It contains properties and behavior of the IDP e.g. it contains the display name of the IDP, different user and domain properties etc. Every AuthManager class contains a reference of the AbstractIDP class which would contain details of the associated IDP. For example CSMAuthManager is associated to CSMIDP class which contains information about the CSM IDP.

Following is the detailed class diagram for AbstractIDP class

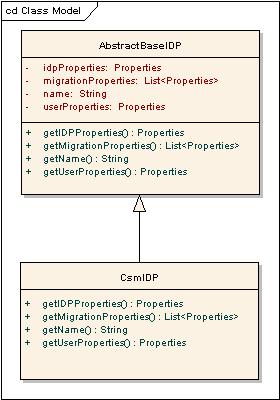


Figure 2: IDP UML representation

As shown in the diagram, AbstractIDP contains various properties which each of the IDPs are supposed to have. Most of the properties and methods are self explanatory.

**Authentication process:**

This section explains the actual authentication process in steps

* While deploying the caTissue application, it is expected that the appropriate implementation of IDPAuthManager interface is specified in a file called “IDPConfiguration.xml”. This file can contain multiple IDP related information since caTissue is now capable of pointing to multiple IDPs at one time.
* Once the user submits his user name and password, In the LoginAction class, the above mentioned file is parsed and appropriate instance of the authentication manger class is invoked. Then the user entered credentials are passed to the invoked class and ‘authenticate ()’ method is called. And based on the return value of this method, control is passed to the appropriate page.

**Migration process:**

Once multiple IDPs are configured by users, administrator might want to migrate the users from one IDP to another. This facility is also provided in this feature. One common use case of this migration of the user is to transfer user accounts from local CSM database to the custom IDP to avoid storing passwords in caTissue system itself.

The migration functionality achieves this requirement using the “IDPConfiguration.xml” configuration file. The IDPs defined in the XML contain information specifying which other IDPs the users belonging to this domain can migrate to, along with the rules that need to be satisfied by the user to be able to migrate to the given domain. For example, for the caTissue application, the users need to migrate to their WUSTLKey domain. The IDPConfiguration.xml for caTissue will in this case contain the definition of the CSM and WUSTLKey IDPs, with the CSM domain specifying that CSM users can migrate to WUSTLKey IDP, along with the rule to be satisfied for migration.

Following are the classes that provide the above mentioned functionalities.

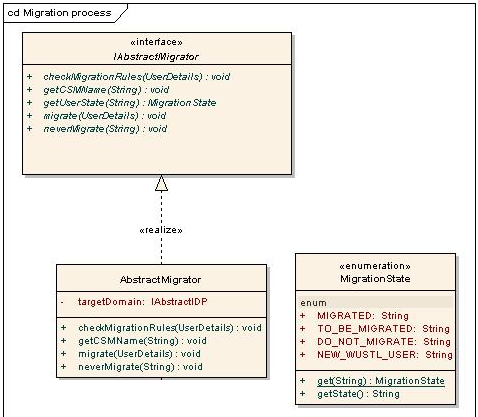


Figure 3: IDP Migration class diagram

IAbstractMigrator.java: This interface defines the behavior of the migratory classes that need to be defined. This interface has to be implemented by all the migratory classes.

Methods:

checkMigrationRules(UserDetails userDetails):This method uses the rule definition to determine if the user whose details have been provided using the userDetails is eligible for migration to the specified IDP. If this method returns true only then the user is given an option to migrate.

String getCSMName(String key): Given the users migrated login name (key), this method returns the CSM login name of the user.

void migrate(UserDetails userDetails): Given the details of the user to be migrated, this method performs the actual migration of the user by updating the database appropriately with the migration state as MIGRATED.

void neverMigrate(String loginName): Given the login name of the user, this method performs updates the database appropriately with the migration state as DO\_NOT\_MIGRATE. A user with such a migration state is not prompted for migration on subsequent logins.

AbstractBaseMigrator.java: This abstract class implements the IAbstractMigrator interface and provides the generic implementation of the methods defined by this interface.

Attributes:

IAbstractIDP targetDomain: This attribute of type IAbstractIDP contains a reference to the IDP to which the users are to be migrated to. For example, in case of migration from CSM to WUSTLKey IDP, this attribute will contain a reference to the WUSTLKey IDP.

Constructor:

AbstractMigrator(IAbstractIDP targetDomain): This constructor initializes the migratory instance with a reference to the target IDP to which the users will be migrated to using this migrator. This class provides the implementation of the methods defined by the interface being implemented.

MigrationState.java: The migration state of the user is defined by the enumeration MigrationState.java, with the following enumerated values.

* MIGRATED
* TO\_BE\_MIGRATED
* DO\_NOT\_MIGRATE
* NEW\_WUSTL\_USER

With this understanding of the different classes involved in the migration process, following are the sequence of events in the actual user account migration process

* When any registered user logs in into the caTissue application, a check is performed in the database to see if any migration record exists for the user.
* If no record is present then a check is performed to see if any of the migration rules are applicable for logged in user by calling ‘checkMigrationRules()’ method on IAbstractMigrator class.
* If the user satisfies any of the migration rules, he is redirected to the migration page.
* On this page user can select the domain he wants to be migrated and enters his domain credentials where he wants to migrate.
* Then the domain credentials are validated, then ‘migrate()’ method is called on the ‘IAbstractMigrator’ implementation for the selected domain. This method internally gets the ‘UserDetails’ object as the argument and performs the migration process for the logged in user. The actual implementation of migration depends on the target domain. This method is expected to update the database for this user to update his migration status as ‘MIGRATED’
* There is one other scenario where user can specify that he/she does not want to be migrated. In this case, ‘neverMigrate(String loginName)’ method is called to update the database accordingly. This method updates the migration status as ‘DO\_NOT\_MIGRATE’

## Adding a new IDP to the caTissue application

This section explains the steps involved in configuring caTissue application to work with a new IDP. As mentioned in the design section, caTissue application code does not have to be modified at all by end users to add or remove an IDP. This integration is achieved by the open-close principle where the required information for the new IDP can be specified in the configuration files which are read by the core caTissue Suite code. Any new java classes introduced because of the new IDP can be added as external JAR files in the class path of the server.

Following are the steps to configure new IDP with caTissue.

1. Configure IDPAuthentication.xml properly, details of configuration are provided in section 1.2.
2. With new IDP implementation caTissue supports migration of a user from one IDP to other. While configuring new IDP if migration functionality is required
   * Define rule interface for migration. (e.g. for WASHU IDP users, login name which ends with .wustl.edu are eligible for migration)
   * Define Migrator class. This is a java class which contains actual migration logic implementation.

Configure **IDPAuthentication.xml**

<?xml version="1.0" encoding="UTF-8"?>

<IDPS>

<IDP name="<IDP\_NAME>">

<class name="<AUTH\_MANAGER\_CLASS\_NAME>" />

<idpdetails>

<defaultidp><DEFAULT\_IDP\_VALUE></defaultidp>

<displayname><DISPLAY\_NAME></displayname>

</idpdetails>

<idp-configuration>

Any properties required by the new IDP can be specified here as name value pairs. E.g. if the IDP uses LDAP server for the authentication, then the properties for LDAP URL, server name can be specified here.

</idp-configuration>

<user-attributes>

If required any user attributes such as first name last name could be specified

here.

</user-attributes>

<migration-details>

<migrator>

<rule-class>RULE\_CLASS\_NAME</rule-class>

<migrator-class>MIGRATOR\_CLASS\_NAME</migrator-class>

</migrator>

<migrator>

<rule-class>RULE\_CLASS\_NAME</rule-class>

<migrator-class>MIGRATOR\_CLASS\_NAME</migrator-class>

</migrator>

</migration-details>

</IDP>

</IDPS>

Once user adds the required information and adds all the implementation classes in the class path, there are no additional changes required and the caTissue Suite will work with the newly added IDP after a server restart.

## I/O considerations

There are 2 xml files viz. “IDPConfiguration.xml” and “IDPMigration.xml” that are used in this feature. These files are accessed at the server startup and stored in memory to avoid further I/O operations.

## Memory considerations

Above mentioned files are accessed at server startup and then this information is kept in memory in a hash map in IDPManager class. This in memory storage is quite lightweight since it stores the IDP configuration information only in strings and no heavy objects are stored in memory.

## Third party components

No third party tools/ components are used in this feature.

## Crash recovery considerations

There are no processes that will affect the state of the application even if server is abruptly shut down. If the server is brought down abruptly during the migration, the whole migration transaction will be rolled back.

## Transaction management

When user is migrated from one IDP to other, there is an operation which updates the database with the user’s migration status. This is a standard atomic operation which uses caTissue DAO framework. If there is a customized IDP for which the existing code is extended, then it is the responsibility of the new code to manage transactions local to the code.

## Multi threading considerations

There is no multi threading involved in this feature.

## Configuration / script changes

The Ant scripts for fresh database installation as well as database upgrades have been updated to support this feature.

## Security considerations

This feature allows for extensions to the existing code. The existing code for this feature does not introduce any SQL injection or XSS vulnerability risks. If the code is extended, then it is the responsibility of the new code to handle security risks.

## Database changes

To accommodate various user properties related to IDP, there is an addition of one table into caTissue database. Following is the schema for the newly added table.



Figure 4: Database changes for IDP

Following is the brief explanation of this table and its columns

CSM\_MIGRATED\_USER : This table contains information about the migration status of the caTissue user.

LOGIN\_NAME: Login name of the caTissue user.

MIGRATED\_LOGIN\_NAME: Login name of the user after migration

TARGET\_IDP\_NAME: Name of the IDP, the user belongs to

MIGRATION\_STATUS: user’s migration status e.g. MIGRATED, DO\_NOT\_MIGRATED, NOT\_MIGRATED etc

# Bulk operation support

Bulk data operation is a new feature in caTissue that allows users to add or edit bulk data in caTissue at one time.

There are quite a few use cases where users / administrators need to import some data into caTissue from external data source. It can be the data that needs to be migrated from the earlier version or large number of specimens that need to be inserted into caTissue database which are coming from some other applications. Another use case would be to disable activity status of many users at the same time. Doing such things manually would take long time and is cumbersome. So using this feature, users can perform such operations on large number of records simultaneously and with ease.

To be able to add or edit multiple records of the same object, bulk operation performs following steps

* It expects an XML template which represents the object model for the object which user is trying to save or update.(object here represents different entities in the caTissue e.g. Specimen, User, Collection Protocol etc)
* Once the XML template is available, bulk operation exports an empty CSV file in the appropriate format.
* User takes this empty CSV file and fills it with the actual data for the multiple records of the object.
* Once the CSV file is filled, user uploads the file; application inserts / updates all the records in the CSV file into the caTissue database.

## Design scope

The design scope for bulk operation includes all the use cases and requirements mentioned above in the introduction section.

## Solution architecture

The most important aspect of bulk operation is to upload huge amount of data within reasonably quick time at the same time make sure that every object that is inserted undergoes the appropriate validation process. So the crucial part of the design process is to make sure that the design allows for quick I/O to read the values from the CSV file , make sure that there is no redundant in-memory storage. Bulk operation is inherently an API way of inserting the objects into the caTissue database. So the BO feature internally uses caCore framework to insert/ update objects in the caTissue. This has several advantages. CaCore already takes care of all the validations so it can be reused. Also using caCore, BO can reuse its exception handling as it is.

There are scenarios where user would try to insert large number of objects using bulk operation feature. So it is important to have this feature run in the background using parallel threads. So every bulk operation job starts in a separate thread so that different users can start more than one bulk upload jobs at the same time.

This section explains these design considerations that are followed in implementing bulk operation feature in details.

## Detailed design

While performing bulk operation, user can insert / edit any object in the caTissue domain model along with any annotation defined using dynamic extensions as long as the template for these objects is uploaded. This means that bulk operation feature has to be intelligent enough to deal with any domain object of caTissue as well as any entity (defined annotation) from dynamic extensions.

To solve this problem, a controller pattern is used where a centralized controller class is responsible for all domain objects and operations (insert/ edit)

Following class diagram explains the controller class involved in the feature



Figure 5: Bulk operation controller class

As shown in the diagram, the class ‘BulkOperationProcessController’ acts as the central controller. So whenever a bulk operation is started by the user, the control first comes to this class.

Methods:

preProcess() : This method initializes the different parameters and classes required for the bulk operation process .

process(): This method contains the actual implementation of the bulk operation process. This is the starting point of the actual bulk operation algorithm.

postProcess(): This method performs the cleanup operations after the processing is completed. The final report for the bulk operation is generated in this method.

‘BulkOperationControllerFactory’class is the factory class used for obtaining the instance of the BulkOperationProcessController’ class.

As mentioned above , the process method acts as the starting point for the BO algorithm. In the next section, this algorithm is explained along with the different classes involved. The member variables in the controller class in the diagram above are explained in this section.

Bulk operation processors

As mentioned at the start of this section, in bulk operations, user can insert different types of objects which can be

* caTissue domain objects (static domain model)
* Dynamic extensions entities (dynamic domain model)
* Category classes defined in dynamic extensions (dynamic domain model)

For each of these types of objects, a different processor is needed since the actual way of inserting these objects in the database varies with the type. E.g. the way the ‘Participant’ object is inserted into the database is different from the way ‘SmokingHistory’ entity (annotation) data is stored into the database.

Keeping this in mind, there are different processor classes which are used to perform bulk updates. Following diagram explains the classes involved in this context.

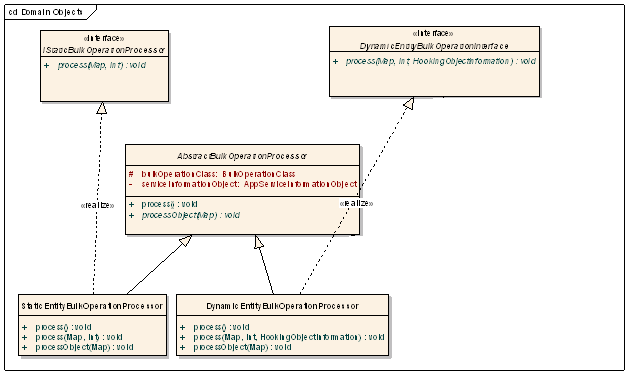


Figure 6: Bulk operation processor class diagram

AbstractBulkOperationProcessor is the base class for all types of bulk operation processor classes.

Attributes:

bulkOperationClass : This represents the template xml file using which user is trying to perform bulk upload. This is used in all processors to lookup the structure of the actual input CSV file.

serviceInformationObject: This internally represents the caCore application service delegator class. Bulk operation processors use this object to delegate the insert or edit method calls to the host application caCore service.

Methods:

process(): This is the main method using which the bulk operation is performed.

processObject(Map): This method is used to process one object at a time. The map represents one row in the input CSV file. One row in the CSV file represents information for one caTissue object. So the purpose of this method is to insert or update one object based on the data entered by the user.

IStaticBulkOperationProcessor is the interface representing processing for the static domain objects of caTissue. StaticEntityBulkOperationProcessor class implements this interface. All the methods on this class are self-explanatory.

IDynamicBulkOperationProcessor is the interface representing two types of dynamic object processors.

DynamicEntityBulkOperationProcessor class represents the processor to insert/ edit dynamic extensions entities (annotations)

DynamicCategorryBulkOperationProcessor class represents the processor to insert/ edit dynamic extensions category classes.

The only difference between static and dynamic processors is that the dynamic bulk operation processors have an extra argument to the process method. This extra argument is of type ‘HookingObjectInformation’ which contains the information about the static entity to which the newly inserted dynamic object needs to be associated with. E.g If there is an entity defined viz. ‘SmokingHistory’ which is captured along with ‘Participant’ object in caTissue. When user tries to bulk upload such participants, at that time, when system is inserting SmokingHistory data, this data needs to be associated with the participant data that is entered. The ‘HookingObjectInformation’ object represents the information using which dynamic processors can associate static and dynamic data.

## Bulk operation algorithm

This section explains the algorithm which is used in the bulk operation.

* When user submits a bulk operation job, then in a separate thread, control comes to the ‘BulkOperationProcessContoller’ class. Every bulk operation job starts in a separate thread. So each job gets a new instance of ‘BulkOperationProcessController’
* Controller then creates a BulkOperationClass object which represents the template metadata for bulk operations and then determines based on the template whether the operation contains only static data or it contains dynamic data as well. Accordingly it populates the processor instances.
* Controller class then first invokes the StaticBulkOperationProcessor which inserts one object at a time. If the static object has any dynamic data associated with it. Then the controller class creates a HookInformationObject and populates the information. This object is then passed to the dynamic processors where they process one dynamic entity at a time and then associate it with the static entity using the hook information object.
* Finally once all the objects are processed, in the postProcess method of controller class, a report is created which reports how many objects were successfully processed and erroneous objects etc.

## I/O considerations

Extensive I/O operations are involved in this feature. The bulk operation data is uploaded as a CSV file. Entire contents of the CSV file are not read and stored in memory in advance. File is accessed one row at a time to avoid large in memory storage.

Also once the bulk operation job is complete, a final report is stored as a zip file on the server. This file is created in memory and written as a batch update to avoid large number of I/O operations. This output file adds two extra columns to the input CSV file; one for the identifier and other for the status of the record in that row. So this file write operation is substantial in size but is optimized with the use of batch updates

## Memory considerations

The template for the bulk operation is stored in memory as long as the job is in progress. Otherwise, there is no other in memory storage for long duration. The size of the template is always fixed and is not dependent on the number of rows to be processed. So keeping the template does not cause any scalability or performance issues.

## Third party components

This feature internally uses [caCore](https://cabig.nci.nih.gov/concepts/caCORE_overview?pid=primary.2006-07-07.4911641845&sid=caCORE_Overview&status=True) framework of caTissue application to interact with database.

## Crash recovery considerations

Each bulk operation job is performed in a separate thread. If the server is brought down abruptly, there is a possibility that some BO job is in progress and the abrupt shutdown can cause inconsistent database state. This can be addressed at the server startup where system can check if for any BO job the status is still “in progress”, then it is apparent that it is caused by the abrupt shutdown of the server. So in this case, system will change the status to failure and generate the appropriate report file. Also the report file will have the last record processed so that user can resume processing the rest of the objects again as a separate job.

## Transaction management

Bulk operation internally makes use of caCore APIs to store the objects in database. So if there are multiple objects in the bulk operation, and if insertion of some of the objects causes exception and result into failure, entire bulk operation is not rolled back. Each row in the CSV file is treated as one transaction and rollback across multiple object insertion/ modification is not implemented neither is in scope in the feature. For failure cases, status is updated in the final report.

## Multi threading considerations

Each bulk operation job starts as a separate thread so at a given time, multiple jobs can be in progress. We can have multiple deployments in which the jobs can parallel process bulk operations and load them into the same db instance for scalability. If there are any issues because of abrupt server shutdown, then the failed jobs are processed at server startup to set their status as failed and that time clean up is done.

## Configuration / script changes

There is an Ant script target viz. “add\_bulk\_operation\_template” added which uploads the template xml file into the caTissue database.

Also to run the bulk operation through client program, additional Ant script has been added viz. “runBulkOperation”

## Security considerations

Bulk operation feature depends on underlying caCore framework for the database interaction, validation and security considerations.

## Database changes

User can view the bulk operation jobs that he/she started and can download the report using a dashboard feature that is provided as part of the bulk operation feature. To show the previous jobs and related information to the user, a new table is added to caTissue viz CATISSUE\_BULK\_OPERATION

Following diagram represents the schema of this table



Figure 7: Bulk operation database changes

The purpose of each column is described below

IDENTIFIER: the unique identifier for each BO job.

OPERATION: Bulk Operation Name.

CSV\_TEMPLATE: CSV\_TEMPLATE column names are saved.

XML\_TEMPLATE: XML metadata template is saved.

DROPDOWN\_NAME: Bulk Operation Name visible in drop down menu on UI of Bulk Operation page.

# CaTissue-CDMS integration

This feature allows institutes to integrate their legacy CDMS systems with the caTissue application. Typical use cases for this integration is that caTissue captures the participant and specimen data but there is no provision for capturing participant’s clinical trial data. For such clinical trial data, generally a dedicated application is used in many institutes. The purpose of this feature is to associate the clinical data in CDMS systems with the actual participants registered in the tissue banking application (caTissue)

Currently the core use case of associating participant in caTissue with his / her clinical data in CDMS system is the only one targeted in this feature. Also there is provision for two way communication so that both applications can initiate the interaction.

## Design scope

The use case described above is the scope for the design of this feature. The most important design considerations in this feature are

* It should be extensible so as to work with most CDMS applications seamlessly as long as CDMS application supports caCore APIs. It should be configurable so as to turn on / off this feature.

## Solution architecture

One of the ways external applications can communicate with caTissue is through the caCore APIs. It is a tried and tested way of an API level interaction. So caCore framework is the backbone of the solution for this integration. The caTissue application exposes the APIs that are needed for the CDMS system and in turn expects the same from the CDMS systems. Following sections will explain the details of the APIs and difference classes involved.

## Detailed design

As mentioned above, the communication between caTissue and CDMS is through caCore APIs. So this section explains how this is implemented and the description of the classes involved.

**CaTissue-CDMS communication**

When caTissue needs to enter clinical data for a particular participant in CDMS application, it needs to have the URL of CDMS application for the particular clinical data and participant mapping. This is achieved in following way

* CaTissue application expects an API from the CDMS application that gives out the required URL.

This API takes in an input object which contains the caTissue specific information i.e. participant id and collection protocol identifier etc.

* Once caTissue application gets the URL, control is passed to the page where user can enter the required information.

Following class diagram explains the above mentioned caCore class and the API

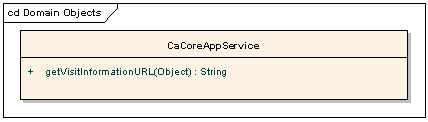


Figure 8: caTissue- CDMS integration caCore API

CaCoreAppService:

This is the actual caCore service exposed from the CDMS system.

Methods:

String getVisitInformationURL(Object) : This method is used to get the required URL based on the input object. It returns the actual URL which caTissue requires. The argument to this method is an instance of CatissueCDMSURLInformationObject class which contains the information based on which the URL is constructed.

When caTissue needs to have the access to the CDMS URL, the caCore communication is initiated in the class viz. CDMSIntegrator. This class is responsible for initializing the CDMS caCore service and setting up the session.

Following is the class diagram for this class

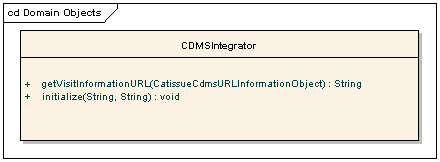


Figure 9: Class diagram for the caTissue-CDMS integrator

Methods:

String getVisitInformationURL (CatissueCdmsURLInformationObject): This method calls the appropriate caCore API internally to retrieve the desired CDMS URL.

Initialize (String, String): This method initializes the CDMS caCore service and sets up the session. It takes in the currently logged in user’s login name and password as the arguments.

**CDMS-caTissue communication**

In caTissue, there is a provision for two way communication where CDMS application can interact with caTissue.

Catissue has provided this support at SCG level where CDMS application can query caTissue caCore API to get the URL of the SCG page for the appropriate specimen collection group. This is implemented in following way

* CaTissue has provided a new caCore API where CDMS can pass the required input and can retrieve the URL for SCG. The input for this API is the CatissueCdmsURLInformationObject class
* Once CDMS gets the URL, it can pass the control to the URL where user can enter information about the participant.

Following is the class diagram for the caCore API along with the new API

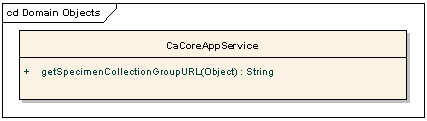


Figure 10: New caCore API in caTissue for CDMS integration

Methods:

String getSpecimenCollectionGroupURL(Object): This method takes an CatissueCDMSURLInformationObject class as an argument and returns the appropriate URL back based on the information passed in the argument object.

Initialize (String, String): This method initializes the caTissue caCore service and sets up the session. It takes in the currently logged in user’s login name and password as the arguments

## I/O considerations

When caTissue passes control to the CDMS application, it depends on the CDMS application for any I/O involved. Otherwise it does not involve any other major I/O operations.

## Memory considerations

This feature does not involve major in memory storage.

## Third party components

CDMS integration has a dependency on caCore framework. Currently caTissue application expects CDMS application to have running caCore service.

## Crash recovery considerations

This feature does not introduce any processes which can cause unstable application state on abrupt server shutdown.

## Transaction management

CDMS integration internally depends on caCore framework for the transaction management. Also for any CDMS application database interaction, caTissue application has no control and it completely depends on the CDMS application to handle the transactions properly.

## Multi threading considerations

There is no multi threading involved in this feature.

## Configuration / script changes

No new configuration / script changes have been done in this feature

## Security considerations

This feature depends on CDMS application to handle its own security risks. When CDMS application accesses caTissue application, caTissue is responsible for handling the security risks like XSS attacks or SQL injection. CaTissue application uses its own stinger framework to handle these security risks.

## Database changes

No database changes

# Template based label generator

In caTissue v1.1.2, application generates the labels for specimens in a fixed way. Every new specimen that is being stored gets a new number which is based on how many specimens there are in the database. This format of specimen label generation is not flexible and intuitive. So an enhancement is being implemented where users will be able to specify the format that they want for their specimens.

One more limitation of the earlier label generation feature was that flag specifying whether turn the feature on or off was at application level. So user couldn’t use the feature for few collection protocols while disabling it for others. So the new template based label generator is configured in such a way that users can enable / disable it at CP level.

## Design scope

The use cases described above is the scope for the design of this feature. The most important design considerations in this feature are

* It should be extensible so as to work with any new format that user wants.
* It should be configurable so as to turn on / off this feature at the CP instance level.

## Solution architecture

The design of the template based label generator is based on the tokens that user can use to come up with different formats for the specimens. Tokens can be defined for the most commonly used factors that user wants to use in the labels. E.g. most institutions use the collection year of the specimen in the label so that specimens can be easily searchable year wise. Similarly PPI (Participant protocol identifier) is used widely to make sure that the specimen labels are always unique. For all such cases, tokens can be created which then can be used while specifying the label format. For each token then, a java class is created which returns the actual value of the token for the given specimen. Using this concept, the final value of the label is then generated and assigned to the specimen.

## Detailed design

In the label generator design, the major change compared to the earlier version of label generator is that the format of the label is captured at Collection protocol level. If the format is specified, then the new label generator is used, otherwise the previous version of label generator is used if it is turned on while deploying.

As mentioned above, the label format is captured with the use of the tokens which are predefined. The format is specified in following manner.

%PPI\_TOKEN%\_%SPEC\_COLL\_YEAR\_TOKEN%\_SP\_UID\_TOKEN%

Every token is enclosed within a pair of ‘%’ sign. So the above mentioned format will result into tokens like

“<PPI of the associated participant>\_<collection year of the specimen>\_<specimen identifier>”

e.g. “X03\_2010\_1” where X03 is the PPI, 2010 is the collection year and 1 is the specimen identifier.

Following sections will explain how this is implemented.

The label formats are captured as part of the Collection Protocol definition. A new attribute is added to the Collection Protocol domain object which stores the format that needs to be applied for all the specimens captured as part of that CP.

## Label generator hierarchy

As mentioned above, if the format is specified explicitly by the user at CP level, then the enhanced label generator is used, otherwise the default (previous version of) label generator is used to generate specimen labels.

To implement this, an interface “LabelGenerator” is defined for all implementations of label generators. There are 2 concrete implementations of this interface “DefaultSpecimenLabelGenerator” and “DefaultTemplateBasedLabelGenerator”

Following class diagram represents this hierarchy

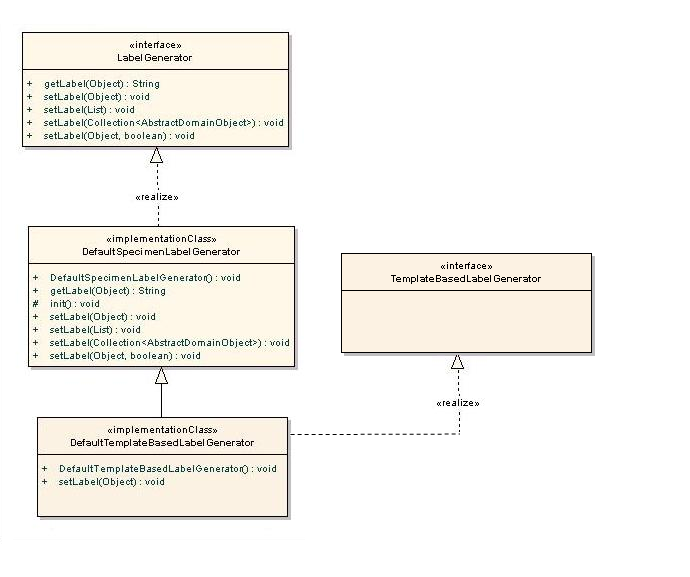


Figure 11: Label generator hierarchy

The class “DefaultSpecimenLabelGenerator” is the previous implementation of the label generator. The new class “DefaultTemplateBasedLabelGenerator” is derived from the default implementation and it provides the new logic of generating labels based on the tokens as mentioned in the sections above.

This new class also implements a new interface viz “TemplateBasedLabelGenerator”. This is a marker interface to let the system know that the algorithm is based on the templates defined by the user.

When user specifies a class to be used for the label generation, system inspects the class to determine whether it implements the “TemplateBasedLabelGenerator” If it does, then system shows the text boxes for the user to enter the templates while defining the CP. Otherwise, these text boxes are hidden from the user and it is assumed that the label generation algorithm does not use the templates.

## Template based label generator implementation

DefaultTemplateBasedLabelGenerator is the implementation class which reads the user defined format and parses the different tokens used and based on these tokens generates the actual label.

List of the predefined tokens are kept in a properties file using which user can define the format. For each token the logic of calculating the value for that token is implemented in a separate java class. The actual label formats as specified by the user are stored as part of the Collection Protocol domain object. If the template based label format is present for any CP, then template based label generator is used to generate the label, otherwise default label generator is used.

The mapping of token name and the class is maintained in the properties file in following way

<token name=”PPI\_TOKEN” class = “edu.wustl.labelgenerator.LabelTokenForPPI”/>

Following class diagram represents the hierarchy of classes involved in the new version of label generator.

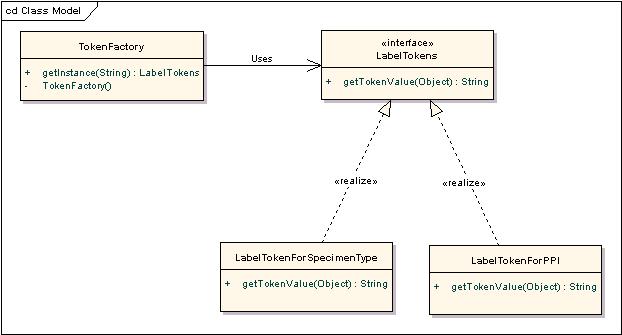


Figure 12: Class diagram to represent tokens

**Labeltokens**

It is the interface which defines the methods to be implemented by each class that calculates the value of the token.

Methods:

String getTokenValue (Object) : This method takes in an Object (which is of type Specimen in this case) and then returns the value of the token based on the argument.

E.g. for “PPI\_TOKEN”, it will get the associated Participant from the specimen and return the PPI of participant as the token value.

The “LabelTokenForSpecimenType” is the class to calculate value of “SP\_TYPE” token which returns the specimen type of the specimen.

TokenFactory

It is the factory class which returns the class associated with a given token.

## Steps to generate the label

Following are the steps (algorithm) to generate labels based on the user defined format.

* When a user enters the label format in the collection protocol definition, the “DefaultTemplateBasedLabelGenerator” class is invoked.
* Specimen action class then takes the user entered format and passes it to the above mentioned class. This class then parses it based on the defined syntax of the format and stores the list of the tokens used in memory.
* For each token it gets the associated class from the “TokenFactory” and invokes method “getTokenValue” by passing the given specimen object to the method.
* Once all the token values are obtained, the label generator class then concatenates them to generate the final label

## I/O considerations

Label generation feature gets the list of configured tokens and associated implementation class from a properties file stored on server. This file is accessed at server startup and then this mapping is stored in memory to avoid further I/O operations.

## Memory considerations

As mentioned above, the mapping of tokens and the implementation classes is stored in memory. This is not expected to be heavy since the number of tokens is not large.

## Third party components

This feature does not involve any third party tools / components.

## Crash recovery considerations

Label generation does not introduce any processes which can cause inconsistent application state in case of abrupt server shutdown.

## Transaction management

There are no operations which involve transactions and management thereof.

## Multi threading considerations

This feature does not involve any multi threading operations.

## Configuration / script changes

This feature does not introduce any additional Ant scripts or configuration files.

## Security considerations

This feature depends on caTissue’s underlying framework to handle security concerns.

## Database changes

As mentioned above, the user defined label formats are stored at collection protocol level. To accommodate this, following columns are added to the “CATISSUE\_SPECIMEN\_PROTOCOL” table.

* LABEL\_FORMAT - to specify the label format for parent specimen
* DERIV\_LABEL\_FORMAT- to specify the label format for derivative specimen
* ALIQUOT\_LABEL\_FORMAT- to specify the label format for Aliquot specimen

Label formats can be specified at derivative or aliquot specimens also. If it is specified at that level, that is used to calculate the labels otherwise, label format defined at parent level is used along with the child specimen identifier to ensure uniqueness. To accommodate this, additional two columns are added to store derivative and aliquot specimen label formats.

# Keyword search

A new feature called keyword search is introduced into caTissue Suite which allows users to do free text search and get the data matching the search strings within the attributes of caTissue application.

A text box is provided in the application where user can enter any search string and internally this feature matches all the specified attributes of all the caTissue entities and provides the matching results. E.g. if there is a specimen with label “MOL\_2010\_4” and if user wants an easy access to this specimen, he / she can just enter the term and keyword search feature will directly open the specimen page for the user. If there are more than one records matching like in case if user enters “john” and there are 2 users with this name and one institute with a name containing the text, this feature will list all these objects in a search result page and users can then visit any entity that they want.

## Design scope

Following are the factors in scope for the design for this feature

* It should be highly configurable so that administrators can decide which attributes should be searched to get the matching results.
* It should be very easy to integrate the existing simple search UI in caTissue so that the search results are shown in a consistent manner.
* User should be able to enter most commonly used regular expressions like “starts with molecular” or “spec\*” etc
* It should work with all the supported databases in caTissue consistently.
* It should not alter the database in any manner.

## Solution architecture

As mentioned above, this feature is expected to be very lightweight and is not expected to increase the load on caTissue database. At the same time, it is expected to support different search regular expressions and needs to be fast enough. Also, the attributes in the database that are searched need to be configurable.

Based on all these factors, using “Lucene” to search for the matching results on the indexes created on top of databases is considered while designing for the feature. In this approach, lucene indexes are created on the file system on the server by querying the database while deploying the caTissue application. This is a onetime activity and once the index is created, when caTissue objects are inserted/ updated, caTissue bizlogic classes update the index so that it s always in synch with the database.

Following sections explain how this design is carried out and the different classes and interfaces involved.

## Detailed design

**Indexing**

The most important factor in the keyword search is the lucene indexes that are created outside the database. While deploying the application, the keyword search properties file is updated to enter following properties

* Attributes / columns in the database that need to be excluded for the indexing
* Database details to be used for indexing
* Location where the lucene index file needs to be stored on the server

Following classes are used to read the database and create the lucene indexes.

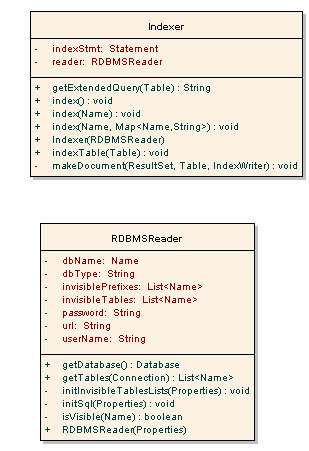


Figure : Class diagram for Lucene database indexer

**RDBMSReader**

This class is responsible for querying the database and gets the list of tables and columns available. Then it retrieves all the data from the database that needs to be indexed. This table is used for indexer and searcher classes to interact with the database. This class contains the methods needed by other classes to query the database.

**Indexer**

This class is responsible for creating Lucene based indexes based on the database and the configured list of entities and their attributes to be indexed. The class uses the RDBMSReader instance to query the database. Once it gets the information from the database, it converts it into String and stores it into index file. The index files are created and stored based on the Lucene guidelines to optimize the performance. The class exposes different methods which index one table at a time.

For every segment of information, Lucene provides the option of either indexing it, storing it or both. The contents of each record are converted to a string and the string is indexed but not stored. At the same time the database name, table name and primary key values of the record are stored but not indexed. When the document corresponding to the record matches a search query, the stored values of database name, table name and primary key values are returned. This information is used to retrieve the record.

The search and fetch functionalities are separated in order to fetch only the selected records. The search is performed on indexes and does not involve databases

**Searching the records**

Once the indexing process is complete, keyword search feature can be used. When user enters a search term, the Lucene APIs are used to search the information in the indexes to get the primary keys of the matched records. Once primary keys are retrieved, they can be used to fetch the complete information about the records and show it in the results.

Following class diagram represents the classes involved in searching the records based on the user search term.



Figure : Class diagrarm for keyword searcher

**Searcher**

This class is responsible for querying the database based on the lucene indexes. This class gets the search term and then uses the Lucene APIs to get the table, column names of matching records. Once it gets this information, it queries the database for the detailed information and returns it as the MatchList object. MatchList object represents the list of matched results.

**Refreshing the index**

The Lucene index is created only once, and thereafter whenever there are any insertions / modifications to the caTissue objects, the index needs to be updated so that subsequent searches are in synch with the latest state of the database.



Figure : Keyword index refresher

**IndexRefresher**

This class exposes methods which are used by caTissue application to update the index whenever a new record is added to caTissue or any existing records are updated. Separate methods are provided for insertions and updations.

**Keyword search centralized manager**

“Titli” class manages the entire work flow when user enters the search term. Following is the flow managed by this class

* It makes sure that the index is properly created and is accessible. If not, then it throws an exception.
* If the index is available, it invokes an instance of “Searcher” class and passes the search term.
* Searcher returns the MatchList object which contains the list of matching results. Once the results are returned, this class converts this result into a data structure that simple search feature of caTissue works with. This is done to make sure that the results are shown in a consistent way. Also doing this makes sure that the existing code is reused to show the results.
* Once the data structure is returned, the control is passed to the Simple search to show the results.

## I/O considerations

This feature needs to store and access the index files that are stored on the application server. Creating the index files is a onetime activity. Batch insertions are used to avoid large number of I/O operations. To access the index files, this feature uses the underlying Lucene APIs to optimize the I/O operations.

## Memory considerations

This feature does not introduce any heavy in memory objects.

## Third party components

Keyword search makes extensive use of free and open source [Lucene](http://lucene.apache.org/java/docs/index.html) for free text search.

## Crash recovery considerations

If the server is brought down abruptly while the initial indexing is still in progress, then the incomplete indexes will need to be cleaned up and the whole process will need to be rerun to ensure proper functioning of the feature.

## Transaction management

This feature does not make any write operations on database. It accesses file system to write indexes and it takes care of the transaction and appropriate exception handling.

## Multi threading considerations

This feature does not involve any multi threading operations.

## Configuration / script changes

An Ant target named “run\_keyword\_search\_indexer” is added that starts the indexing process for keyword search.

## Security considerations

This feature depends on caTissue’s underlying framework to handle security concerns. As mentioned in the detailed design section, while showing the results on UI, keyword search is integrated with the caTissue’s legacy simple search module. This integration makes sure that all the authorization concerns and security checks are taken care of by the caTissue code itself.

## Database changes

There are no database changes because of this feature.

# Single sign on implementation (SSO)

In caTissue v1.2 version, the application needs to support deployment on single sign on environment. This will allow institutions to keep the authentication at a centralized server so that once the user is logged into the SSO server; individual applications will not need to authenticate the user again.

This section explains how the SSO support is provided in caTissue application; how it is deployed with the SSO server. Also the integration between the IDP framework and SSO server is explained in details.

## Design scope

Following are the factors in scope for the design for this feature

* The caTissue application needs to recognize the presence of the SSO server once it is deployed
* If the user is already logged in, caTissue application should not prompt the user for login again and it should work seamlessly.
* The SSO server can be deployed on its own independent server or it can reside on the same caTissue server. Application should be able to work in either case.
* All the configured IDPs should work seamlessly even in the SSO environment.

## Solution architecture

As mentioned above, all the authentication requests need to be routed through the SSO server and caTissue on its own should not handle user’s authentication. At the same time, user should be able to login using his IDP credentials in the SSO environment.

To achieve these requirements, use of CAS (Central authentication service) SSO server is considered where CAS can be deployed in a separate web server or can reside on the same server which hosts caTissue. CAS needs to be configured in such a way that the actual authentication is handled by the IDP framework itself. Then the ticket generated by CAS is stored in the user session and then for every subsequent request, a background request is submitted to CAS server to validate the ticket. The ticket generated by CAS is also stored in the browser cookie so that if user accesses any other application which is in the CAS SSO environment, user will not need to login again since the ticket will be obtained from the cookie and used for authentication.

## Detailed design

When CAS is used as a SSO solution, all the login requests need to be redirected to CAS so that it can authenticate the user credentials and generate the ticket. Following is the component / sequence diagram of caTissue configured with SSO

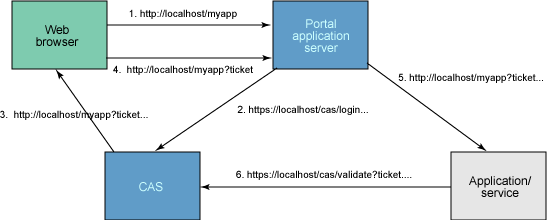


Figure 16: SSO component / sequence diagram

As shown in the diagram, “CAS” is the SSO server (which can be separate or the same application server which hosts caTissue application). “Portal application server” is the application server hosting the caTissue application. “Application/ Service” is the actual caTissue application representation.

The sequence of events that take place when user tries to access the caTissue application is as follows

1. The user attempts to access an application using its URL. The user is redirected to the CAS login URL over an HTTPS connection, passing the name of the requested service as a parameter. The user is presented with a username/password dialog box.
2. The user enters ID and password details and CAS attempts to authenticate the user. If authentication fails, the target application never hears about it -- the user remains at the CAS server.
3. If authentication succeeds, then CAS redirects the user back to the target application, appending a parameter called a *ticket* to the URL. CAS then attempts to create an in-memory cookie called a *ticket-granting cookie*. This is done to allow for automatic re-authentication later -- if present, then it indicates that the user has already successfully logged in and the user avoids having to re-enter his username and password.
4. The application then validates that this is a correct ticket and represents a valid user by calling the CAS serviceValidate URL by opening an HTTPS connection and passing the ticket and service name as parameters. CAS checks that the supplied ticket is valid and is associated with the requested service. If validation is successful, CAS returns the username to the application.
5. Once the validation is successful at step # 4, control is passed to the home page of the application and user can then access the application.
6. One important thing to note here is that user does not even come to know about above mentioned steps since those take place internally in the CAS and caTissue application servers.

## Authentication using CAS

CAS provides a java interface which developers can use to integrate our own authentication mechanism in the CAS server. In caTissue, the actual authentication needs to be carried out using one of the configured IDPs in the application itself so CAS server needs to be configured to provide a custom authentication handler which can work with caTissue’s framework of IDPs.

Following class diagram represents the authentication handler that is configured in the CAS.



Figure 17: CAS Authentication handler class diagram

**CaTissueAuthenticationHandler**

This is the class that is configured as the authentication handler in CAS. This class implements the CAS interface “AuthenticationHandler”

Whenever a login request is submitted for caTissue, CAS invokes following method on this class to validate the credentials

boolean authenticate(Credentials credentials) throws AuthenticationException;

The internal implementation of this class is such that it invokes appropriate IDP implementation class for the authentication based on the logged in user name. IDP implementation then checks for the credentials by querying underlying server and returns the status to CatissueAuthenticationHandler which then returns the same to CAS.

Once the user is validated, CAS generates ticket and as mentioned above, this ticket is then stored in the browser cookie as well as in the current user session so that user can directly access other associated applications without having to login.

## Integration with other applications for SSO

If other applications need to share the SSO with caTissue, then these applications have to have the same user base and support at least one IDP with caTissue application. Once this is ensured, then other applications will need to configure themselves with the same SSO server and they can start working on the same SSO server seamlessly.

## I/O considerations

This feature depends on the actual IDP implementation to authenticate the user. Otherwise this feature does not introduce any I/O operations.

## Memory considerations

This feature does not introduce any heavy objects in memory.

## Third party components

This feature makes extensive use of [CAS](https://wiki.jasig.org/display/CASUM/Home) SSO server. CAS is an open source and free SSO implementation providing easy integration with different applications and different authentication mechanisms.

## Crash recovery considerations

This feature does not introduce any processes that can cause inconsistent application state in case of abrupt server shutdown.

## Transaction management

This feature depends on underlying CAS server for any transaction management. Otherwise there are no additional transaction management risks.

## Multi threading considerations

This feature does not involve any multi threading operations.

## Configuration / script changes

There will be a property in the “caTissueInstall.properties” file where user will need to specify the URL of CAS server. Otherwise there are no additional script changes.

## Security considerations

This feature depends on CAS’ underlying framework to handle security concerns.

## Database changes

There are no database changes because of this feature.