

# SBAM Program Implementation Notes

Tips, Techniques, Patterns and Details

Robert Lacatena

---

## Table of Contents

<b>Introduction .....</b>	<b>5</b>
<b>Project Organization.....</b>	<b>5</b>
Source .....	5
WAR .....	6
<b>Hibernate .....</b>	<b>7</b>
Introduction .....	7
Project Use.....	7
Registering Database Structure Changes .....	8
Gotchas.....	8
<i>Unexpected Updates</i> .....	8
<i>Auto Increment IDs</i> .....	8
<i>SqlQuery Use</i> .....	9
<b>Database and Shared (UI) Objects.....</b>	<b>9</b>
<b>Security .....</b>	<b>9</b>
Introduction .....	9
Database Design .....	10
Session Expiration.....	10
Base Code Implementation .....	10
UI Authentication Implementation .....	11
Servlet Authentication Implementation .....	12
Application Security.....	12
<b>System Messages .....</b>	<b>13</b>
Introduction .....	13
Implementation .....	13
<b>Performance Considerations .....</b>	<b>13</b>
<b>Grid Maintenance .....</b>	<b>13</b>
<b>Validation .....</b>	<b>13</b>
<b>Database Updates .....</b>	<b>13</b>
<b>Component Activation/Deactivation/Refresh .....</b>	<b>13</b>
<b>Portlets .....</b>	<b>14</b>
<b>GWT and GXT Quirks.....</b>	<b>14</b>
Known Bugs.....	14
<b>Programming Tips, Techniques and Patterns .....</b>	<b>14</b>
Database Access.....	14
Database Transactions.....	14
Database Debugging.....	15
<b>Recipes : The SBAM Cookbook .....</b>	<b>15</b>

Introduction .....	15
Components .....	15
Adding a Service .....	16
Instances .....	16
Asynchronous Service Patterns .....	16
NotesIconButtonField .....	17
Adding a Database Table .....	17
Validation Paradigms .....	17
Grid Based Maintenance .....	17
<i>Introduction</i> .....	17
<i>Full Setup Recipe</i> .....	18
<i>Grid Setup</i> .....	19
<i>Add to UI</i> .....	19
<i>Authentication</i> .....	19
<i>Load</i> .....	19
<i>Update / Insert</i> .....	19
<i>Delete</i> .....	19
<i>Validation</i> .....	19
<i>Paging</i> .....	19
<i>Refresh</i> .....	19
<i>Special Grid Buttons</i> .....	19
<i>Special Row Buttons</i> .....	19
<i>User Confirmation</i> .....	19
<i>Combo box with strings</i> .....	19
<i>Combo box with strings and codes</i> .....	19
<i>Combo box with list of instances</i> .....	20
<i>Combo box from database codes</i> .....	20
<i>Dual Grids</i> .....	20
<i>Notes Row Expander</i> .....	20
Icons .....	20
Portals and Portlets .....	20
<i>Adding a Portlet</i> .....	20
<i>One portlet requesting another</i> .....	21
<i>Panel requesting a portlet</i> .....	21
<i>Portlets and Cards</i> .....	21
Fields .....	21
Field, Grid Support Portlets and Panels .....	21
<i>FieldSupport</i> .....	21
<i>GridSupport</i> .....	21
<i>FormAndGrid</i> .....	21
User Access Cache .....	21
Search Caches .....	21
Institution Search .....	21
System Configuration Parameters .....	21
Email .....	22
<b>Design Considerations .....</b>	<b>22</b>
UCNs .....	22



---

## Introduction

Any project, and the resulting system, is the combination (and culmination) of the technologies and design patterns used, along with the habits followed as well as those design decisions that caused a break from those habits. As long as techniques are used with some consistency, and breaks from typical techniques are documented or highlighted in some way, system maintenance, while never simple, can at least be relatively painless.

This document outlines project methods and techniques to be used in maintaining the SBAM system. It will provide a framework for understanding the existing code, as well as recipes and advice for future development or enhancement.

---

## Project Organization

In keeping with most GWT projects, the project is organized in into source (src) and WAR (war) folders.

### Source

Also in keeping with GWT projects, the source folder is first divided into client (i.e. browser), server, and shared code. These folders are in turned divided into functional areas as follows:

- **src**  
All Java source
  - **client**  
All client side (browser) code
    - **services**  
Interfaces required by GWT to implement server side calls
    - **uiobjects**  
Custom user interface objects
    - **util**  
Utility and helper classes for client side code (only)
    - **validation**  
Client side (only) validation classes
    - **SBAM.java**  
The main module
  - **server**  
All server side code
    - **database**  
Any code directly related to database access

- **codegen**  
Code automatically generated by Hibernate
  - **objects**  
Database access objects (primarily customized finder/getter methods)
  - **util**  
Database utility objects
- **fastSearch**  
Customized classes that support optimized (non-DB reliant) search features (such as fast institution searches).
- **servlets**  
Servlet implementations
- **util**  
Utility and helper classes for server side code (only)
- **validation**  
Server side (only) validation classes
- **shared**  
Any code which may be used on both the client and server sides
  - **exceptions**  
Any custom defined exceptions that apply to both the client and server sides.
  - **objects**  
Any data objects (usually serializable data instances)
  - **security**  
Any security related classes
  - **util**  
Utility and helper classes for use on both client and server sides
  - **validation**  
Validation classes for use on both client and server sides
- **SBAM.gwt.xml**  
The module XML (unlikely to need modification)

## WAR

The WAR folder contains a variety of elements – almost all elements apart from Java code – needed to run the application. Some of these elements are standard for GWT or ExtGWT, and are either static (once set up) or else are managed by the IDE. In any case, these areas do not require (and probably should not allow) manipulation and can be safely ignored.

Other folders and files are the specific, proper target for certain changes.

- war
  - conf
  - emails
  - ExtGWT
  - resources

- docs
- images
  - icons
    - form
    - colorful
    - large
      - colorful
    - monochrome
- themes
- sbam
- WEB-INF
  - classes
  - lib
  - web.xml
- SBAM.css
- SBAM.html

---

## Hibernate

### Introduction

Hibernate has been used to facilitate database access, but only as a convenience tool to automatically reflect database changes, and to simplify the construction of the simplest and most common queries (i.e. select all rows from a single table using some limiting criteria).

Most features of Hibernate are not used. In particular, HQL and any use of Hibernate Mappings have been avoided. Table joins are either performed in program logic, or through the construction of native language SQL queries, depending on the performance issues involved.

In some cases, direct database connections are made and JDBC is used instead.

### Project Use

Hibernate is configured to use the c3p0 pooling system.

A modified HibernateUtil class is used to construct hibernate sessions, and a HibernateAccessor class is used to provide for the most common database functions (persist, refresh, getById, getByExample).

The HibernateAccessor class may and should also be extended to work with particular tables, and so perform access on specific tables without requiring type

casting. It can, in simple cases, be used without extension (and with inline type casting).

Note that `HibernateAccessor` differs very slightly from the Home methods created by Hibernate Code Generation in that it uses the `HibernateUtil` class to properly create sessions (as required by Tomcat),

To start any database session, call `HibernateUtil.openSession()` to start a session, and always call `HibernateUtil.closeSession()` to release the session to the pool.


Also call `HibernateUtil.startTransaction()` to start a transaction, and `HibernateUtil.endTransaction()` to end the transaction.

For simple access, the `HibernateAccessor` can be extended, primarily to extend any get or find methods, and to return either an object or list of the appropriate type, by type casting. New finder methods can also be written with specific Hibernate functionality.

There is no need to extend the `persist()` or other methods, as these access Object parameters, however they can be extended to provide for some level of type validation/checking.

## Registering Database Structure Changes

The Hibernate code generation tools are used to generate the Hibernate mapping files, as well as appropriate data objects to represent table row data instances. To register any change to the underlying structure of the database rerun the project's

Hibernate Code Generation Configuration from the Hibernate dropdown icon , or from the Run → Hibernate Code Generation menu.

## Gotchas

### Unexpected Updates

All access is done by beginning a database transaction. If any column values are changed in the database entities returned, and the transaction is ended without a rollback, then those columns will eventually be updated in the database, whether or not an explicit “persist” call is made.

Unless you intend to do updates, use the instance objects which are required in any event by the GWT client. Do not make changes directly to the Hibernate objects unless your intent is to actually update them.

### Auto Increment IDs

Hibernate will not give back the auto increment value assigned for a composite key, so when this is done, the application needs to instead compute that value itself.



## SqlQuery Use

Because of the way Hibernate works, it is critical that any SqlQuery created be constructed with {} around the column names, and then the appropriate entity references be explicitly add where this query is used. Without this, Hibernate gets confused about the column names so when the same name appears in both tables (such as STATUS and CREATED\_DATETIME, which are common to most tables) then both table entities get the values of the first occurrence of the column name (i.e. the value from the first table).

For example:

```
String sqlQuery = "select {agreement.*},{agreement_term.*} from  
agreement, agreement_term where agreement.id =  
agreement.agreement_id";  
SQLQuery query =  
sessionFactory.getCurrentSession().createSQLQuery(sqlQuery);  
query.addEntity("agreement", Agreement.class);  
query.addEntity("agreement_term", AgreementTerm.class);
```

---

## Database and Shared (UI) Objects

hibernate generated database instances

shared "Instances" – don't affect DB directly, can be used on client side and so passed between the two

Special getters/setters (like "active" for "status"), lists of child instances

Duplication (i.e. instances are not unique)... simple key model

---

## Security

### Introduction

Application security is implemented using logon, with users (identified by user name and password) and user roles associated with each user name.

Security is enforced at two levels, in the UI (to provide visual feedback for what a user can and cannot do) and in any servlets that actually access the database (to prevent a user from bypassing security using browser utilities to change JavaScript values).

## Database Design

The *user* table identifies a user and his password, as well as supplying the user's name and basic session tracking information (such as the time of the last session and the total number of sessions to date).

The *user* table also includes a mechanism for expiring sessions (an expiration date and time), although this has not yet been implemented.

The *user role* table lists the roles (as simple strings) available to each user. The valid roles are hardcoded in the *Security Manager* object (*com.scholastic.sbam.shared.security.SecurityManager*). Any user may have any number of roles.

Each role corresponds to a single branch (collection) of activities. Currently, three roles are defined: Maint, Query and Admin.

- Maint = data maintenance
- Query = data inquiry and reports
- Admin = access to administration functions, like the ability to maintain users and system messages

The user role table also includes a read/write field to allow roles to distinguish between the two functionalities in one area, however no code is currently implemented to make use of this, and in fact the call-structure does not support it (i.e. only role names are passed, not role names + read/write caps).

## Session Expiration

To enforce session expiration, the servlets must check the database for the session expiration date and time before performing any further database access or returning any results. The UI should also implement a timer to automatically log the user out, so that the user receives visual feedback when a session has expired.

The session expiration date/time should be updated by the servlets every time an action is performed, and the UI timer should similarly be reset. UI interactions which do not result in servlet calls should also periodically call another servlet simply to update the session expiration date/time to keep it current, although this need not be done for every individual operation. A better mechanism might be to allow the session expiration timer to perform this task (say four times as often as the actual session expiration length).

## Base Code Implementation

The following classes are defined to support security:

*com.scholastic.sbam.shared.security.SecurityManager*

This class defines security constants and common methods needed by both the UI and servlets.

#### *com.scholastic.sbam.shared.objects.Authentication*

This class carries authentication information. It is instantiated and populated by the logon servlets, and passed to the UI for use.

#### *com.scholastic.sbam.client.uiobjects.AppSecurityManager*

This interface declares the *applyRoles* method, which allows every UI object which implements the interface to be told the roles for which the current user qualifies, and to enable or disable UI elements as required.

#### *com.scholastic.sbam.client.uiobjects.LoginUiManager*

The Login UI Manager takes care of managing the two user interface components, a dialog for accepting user names/passwords (and for blocking the rest of the app if the user is not logged in) and a panel for displaying a logged in user name and a logout button.

The Login UI Manager, as the bridge between those two UI components and the application, also takes care of defining the actual backend interaction to perform authentication.

#### *com.scholastic.sbam.client.uiobjects.LoginDialog*

The Login Dialog provides modal window with a user name and password field, along with a log in button. The Login UI Manager takes care of establishing the listener and code for the login dialog to perform actual authentication.

#### *com.scholastic.sbam.client.uiobjects.LoggedInPanel*

The Logged In Panel is a simple panel with the currently logged in user name, and a log out button (or “Please log in” if a user is not currently logged in). The Login UI Manager takes care of establishing a listener for the logout button to perform de-authentication (i.e. terminate an active session).

## UI Authentication Implementation

Separate from adding the Login UI Manager to the application, and the Login Dialog and the Logged In Panel to the UI itself, the application must take care of giving every interested component the list of user roles whenever a user logs in or out (in which case the “current user” has no roles).

This is done through the Application Navigation object (*com.scholastic.sbam.client.uiobjects.AppNav*), which is the main application container. That object implements the *applyRoles* method, which will be called on log in or logout through the Login UI Manager. The Application Navigation object calls the *applyRoles* method for every component, and each component is responsible for then calling *applyRoles* for any interested child components.

## Servlet Authentication Implementation

Because it would be possible to bypass authentication on the JavaScript side, either by directly submitting URLs, or by editing JavaScript information, authentication must also be checked on the servlet side (using database, not user supplied, authentication data). This must be done before any updates are performed, critical processes initiated, or sensitive information is returned to a web page (such as user names and passwords).

To facilitate this, the *AuthenticatedServiceServlet* class has been created. This class implements standard methods to perform authentication, but does not actually force the action. Services that require authentication (which is most of them) should extend this class, then immediately call the *authenticate(String taskName, String roleName)* method to perform authentication.

If the method fails, it will throw an *IllegalArgumentException* with a message that incorporates the *taskName* argument, declaring that the user does not have privileges to perform the task.

A null value can be passed as the *roleName*, or an alternate version of the method can be called without that argument. In either case, the method will require that a user be logged in, but will not check for any particular role to validate the capability. This is useful, for instance, for a task like retrieving and updating sticky notes, which is valid for any user, but only when logged in.

Role names are listed as constants in the *SecurityManager* class.

## Application Security

Since a user can log off at any point in the application, and a new user can log on with new capabilities, the application and all components must be prepared to enable or disable child components according to the user roles.

To do so, all components should implement the *AppSecurityManager* interface. The required *applyRoles()* method should be implemented to enable or disable any components, and to pass on *the applyRoles()* call to any *AppSecurityManager* components.

---

## System Messages

### Introduction

The system includes a simple method for recording and displaying simple system level messages on the Welcome tab. These messages are entered into the database and expire automatically on a specified date. They are displayed before a user signs on, and so should not contain sensitive information.

### Implementation

Messages are stored in the *messages* table. They are automatically loaded into the Welcome tab when the system is started, through a servlet which performs the asynchronous access.

---

## Performance Considerations

Choices, like institution search (trade off of max memory usage for speed) versus other searches (small tables that change infrequently cached in user memory, and refreshed periodically – 5 to 10 minutes) versus Link (direct database access each time, which is slow and CPU intensive, but links are used so infrequently that it's the most straightforward way).

---

## Grid Maintenance

Grid maintenance has been standardized using

---

## Validation

---

## Database Updates

---

## Component Activation/Deactivation/Refresh

sleep/awaken, collapse/expand

---

## Portlets

---

## GWT and GXT Quirks

### Known Bugs

Hidden columns in grids, with auto-expand, cause header versus data cell width misalignments, which are in particular evidenced by a notes row expander.

---

## Programming Tips, Techniques and Patterns

### Database Access

Database access is implemented using Hibernate. A base helper class, *HibernateAccessor*, has been constructed and may *optionally* be extended for each table. The only reason for extending it is to provide custom, commonly needed find/get methods (for instance, to retrieve a list using some commonly referenced parameter, or a single instance based on some combination of unique key fields). This is almost always needed for any but the simplest of classes, if only to provide the most obvious of find/get methods.

As a general rule, any hibernate access should be localized within a *HibernateAccessor* extension class. No hibernate code should be embedded in application classes, in order to make it easier to extricate Hibernate from the application at a later date.

By convention, classes that extend *HibernateAccessor* are placed in the *com.scholastic.sbam.server.database.objects* package, and are named following the pattern *DbTableName*.

In keeping with Hibernate restrictions, every database access must be bracketed by calls to *openSession* and *closeSession*, and *startTransaction* and *endTransaction*.

### Database Transactions

Currently, the database uses the MyISAM engine. MyISAM does *not* support transactions, even though the application has been designed to make use of them.

To implement transactions, the database tables must be changed to use the InnoDB database engine. The only change that should be made to the application would be to change the *hibernate.dialect* property in the *hibernate.cfg.xml* file to the value *MySQLInnoDBDialect*.

## Database Debugging

To turn on SQL generation in the logs to help debug database issues, change the *hibernate.show\_sql* property to *true* in the *hibernate.cfg.xml* file.

## Recipes : The SBAM Cookbook

### Introduction

This section will provide brief recipes (simplified sample code) for doing specific things (i.e. building specific types of features) in the SBAM system.

Whenever possible, most recipes will be brief, although the larger picture of constructing any sort of UI screen involves the combination of many separate recipes, and so when taken as a whole is rather large. This special case is treated slightly differently, to help the programmer to merge the various recipes needed to complete the task.

### Components

Many components should implement the *AppSecurityManager* and *AppSleeper* interfaces, so that the component will enable or disable it's own components, as needed, and will release memory (sleep) when another component is activated, and more important will refresh it's data when it is reawakened.

```
public YourClass extends Whatever implements AppSecurityManager,
AppSleeper {
    Component firstChild;
    Component secondChild;

    public void applyRoles(List<String> roleNames) {
        if (roleNames.contains(SecurityManager.ROLE_ADMIN)) {
            firstChild.enable();
            secondChild.enable();
        } else {
            firstChild.disable();
            secondChild.disable();
        }
    }

    public void sleep() {
        firstChild.sleep();
        secondChild.sleep();
    }
}
```

```
public void awaken() {  
    if (!firstChild.isCollapsed()) {  
        firstChild.awaken();  
    }  
    if (!secondChild.isCollapsed()) {  
        secondChild.awaken();  
    }  
}  
}
```

## Adding a Service

Instance or list (or updateresponse)

special response instance if necessary

3 service components, client service and async, then implementation

Synchronized services (such as for a search where an out-of-date earlier request might deliver a response after a later request).

## Instances

descriptionAndCode

uniqueKey

obtainInstance

getEmptyInstance / getUnknownInstance / getAllInstance

active/status

alternate forms (char vs. Boolean)

child instances

## Asynchronous Service Patterns

Update (UpdateResponse, instance nulls not updated, status fields, id after create, created datetime after create, etc.)

Get

List

Update Note



## NotesIconButtonField

Adding/editing notes on panels

## Adding a Database Table

table, gen with hibernate

instance

dbhelper

dbhelper: get, any find/findall/findfiltered, getInstance

see adding a service (usual services = get, list, validate, update)

## Validation Paradigms

simple shared field validator with no backend access

validation service for one field

validation service for a record (usually attached to the key field)

## Grid Based Maintenance

### Introduction

Grid maintenance is complicated by the number of steps necessary to create an implementation, but most of those steps are fairly straightforward and robotic.

Some steps are optional.

The program first needs a class extending the *BetterRowEditInstance* to encapsulate one row's worth of data.

The core UI component should extend the abstract *BetterEditGrid* class, which will make use of the *BetterGridEditRow* class. The *BetterEditGrid* must be extended to declare the extended *BetterRowEditInstance* being referenced, and to call the specific service which will load the data into the grid.

A server side load service must be created and declared to load the grid.

Validation will require the creation of individual client side validators for each non-trivial field (i.e. any field with more than a minimum length or numeric value requirement).

Some fields will also require server side (i.e. database access) validation. At a minimum, it is often necessary to validate the key field values for new rows do not already exist in the database. This will require the creation of a backend validator, as well as a validator service.

The client side and server side validators must be passed to the grid column when it is created.

Actual database updates must be performed by a server side update service.

All server side services should use standard validation to confirm that the user has such capabilities, even if such authentication is being performed on the client side (to avoid spoofing or direct service calls).

Many grids with very large volumes of data will require some paging facility.

Some grids will require logic to determine when and how to refresh the grid, in case backend data has been changed in the interim.

A very few grids may require special capabilities, implemented as either grid-wide or row-specific buttons.

Some tasks will also require user confirmation as a second step.

Some grids may include filters. This is easily done, using either an automated method, or by hand coding.

### **Full Setup Recipe**

To create a full edit grid for a table,

web.xml

client.services

- DeleteReasonCodeValidationService
- DeleteReasonCodeValidationServiceAsync
- DeleteReasonListService
- DeleteReasonListServiceAsync
- UpdateDeleteReasonService
- UpdateDeleteReasonServiceAsync

client.uiobjects

- AppNav.java -- add config tab
- ConfigUi.java -- add editor tab, editor object, awaken/sleep
- DeleteReasonEditGrid.java

server.objects

- DbDeleteReason.java

server.servlets

    DeleteReasonCodeValidationServiceImpl

    DeleteReasonListServiceImpl

    UpdateDeleteReasonServiceImpl

server.validation

    AppDeleteReasonValidator

shared.objects

    DeleteReasonInstance

    DeleteReasonInstanceBeanModel (or else use tag)

shared.validation

        new validation objects (none for Delete, except std CodeValidator and NameValidator)

### **Grid Setup**

### **Add to UI**

### **Authentication**

### **Load**

### **Update / Insert**

### **Delete**

### **Validation**

### **Paging**

### **Refresh**

### **Special Grid Buttons**

### **Special Row Buttons**

### **User Confirmation**

### **Combo box with strings**

like user capability (role) groups

### **Combo box with strings and codes**

like service types in Services

**Combo box with list of instances**

**Combo box from database codes**

**Dual Grids**

**Notes Row Expander**

## **Icons**

IconSupplier, resources/images organization (icons/colorful and icons/monochrome)

Creating icons (sepia tone version)

## **Portals and Portlets**

AppPortalProvider

creates and configures portlets, adds them to the portal

AppPortalIds

lists all valid portlets, and associated titles, icons, and help text

AppPortalRequester

Interface that allows a portlet to request the creation of another portlet within its assigned provider (which is in turn directed at a specific portal – maybe its own, maybe another)

AppNavTree

simple tree structure for presenting portlets that may be created

App Portlets

get closeable box, help tool if available, minimizeable, get provider if allowed

UserPortletCache

## **Adding a Portlet**

Create icon, add to icon supplier (if necessary)

Add to AppPortletIds, AppNavTree

**One portlet requesting another**

**Panel requesting a portlet**

**Portlets and Cards**

## **Fields**

Field factory

Combo boxes

cached instances like TermType – instance, beanmodels

Integer, Dollar, Date

## **Field, Grid Support Portlets and Panels**

**FieldSupport**

**GridSupport**

**FormAndGrid**

## **User Access Cache**

Agreements, Institutions

## **Search Caches**

cache singleton, threaded load, state, initializer, reload mechanism (admin or detect changes or periodic (e.g. every hour))

## **Institution Search**

InsitutionCache – ucns and words, Initializer servlet, mapsReady

Configuration options

LiveGridView use

ComboBox use

## **System Configuration Parameters**

sys\_config table, AppServerConstants classes, and Initializer servlet

## Email

email folder in war file, email templates and replacements, MailHelper class, sys\_config variables

Creating a new e-mail with a template

Adding a new replacement value

---

## Design Considerations

### UCNs

suffixes and the ucn conversion table... system supports them for retroactive use, but not future use... no ability to choose suffix when adding sites (always “1”), create new suffixes, etc.