CDI APPLICATION TEMPLATE

Department of Insurance
Information Technology Division
Application Development & Maintenance Bureau (ADAM)

Document Prepared by:

Aguilar-Barajas, César A.

2008 Architectural Design prepared by:

2008 Common Jar Collective
Aguilar, César
Behrens, Bradley
Christian, Cathi
Franklin, Rick
Huang, Bill
Love, Scott
Lui, William
Porco, Scott
Wright, Chris
Yum, Peter

Draft

April 17, 2008

Table of Contents

1. Justification

The main purpose for building an application template is to provide a starting point for application development that incorporates resources and features that have been adopted as standards for all CDI web applications.

Having an application template should speed up development time since developers won't need to recreate the standard components that are common in all applications.

2. Overview

The *CDI's application template* consists of two jdeveloper projects: the common project that contains a collection of reusable code for common use and the template project that demonstrates a finalized project implementing all the CDI standards for J2EE application development.

The common project's output is a Jar library that is attached to the *CDI's application* template where the method invocation is sampled.

Reusable classes in the common project can be customized inside the project scope by extending the library classes and making the necessary changes to the project.

The CDI's application template contains two types of classes: sample classes and customizable classes. Sample classes provide an example of how classes should be implemented and how they collaborate with the rest of the classes. Customizable classes are skeleton classes that can be thrown away or used as the starting point for creating specific classes needed for you project.

3. Application Scenario

The application scenario demonstrates the typical usage of the application template. The topic has been randomly selected and doesn't reflect the requirements of any CDI's web application.

CDI's Insurance Company Contact List

CDI maintains a database with contact information of many insurance companies. This information needs to be accessible to all internal users and external users with the difference between the two being the latter can only retrieve their own company information. As a result, the intranet portion of the application is indented for CDI and the Internet portion for external users.

CDI's analysts retrieve company records by choosing it from a report list, or by searching them using a web page form. CDI analysts are authenticated using the Licensing database access control list.

The Internet portion allows Companies to access their individual profile by validating their National Association of Insurance Commissioners (NAIC) id against the column value in our contact list data table.

4. Scope

The *CDI's application template* has the following resources/features:

1. Implements the MVC design pattern
2. Implements the composite view pattern to manage a standard look and feel through the application. A composite view pattern composes web pages of multiple subviews
3. It contains a reusable front controller class and provides sample handler classes, delegate classes, DAO classes, TO classes, and a cached data manager class
4. Uses the log4j library for logging messages using a file appender, a console appender and an SMTP appender
5. Uses a resource bundle properties file to load application parameters that need to be changed when switching from production and test environments
6. Uses the latest CSS guidelines that have been released by the web services unit
7. Samples a security mechanism in the view and controller layers of the MVC model for protecting resources from anonymous access.

5. Execution

Application template can be found in the CVS repository under the name "TemplatePrj". There are no prerequisites for running *application template* in JDeveloper. However, it is highly recommendable to customize the application parameters (explained later in section 7).

6. Patterns

Two different patterns are used in the CDI template based applications. The most important one is the Model-View-Controller (MVC) pattern. The main goal of this pattern is to isolate the business logic from the user interface. The MVC pattern divides an application into three different layers: the presentation (user interface), domain logic and data access. For implementing the presentation layer we used another pattern: the composite view design pattern, which is used to make all JSP pages share a similar layout structure. This is the second pattern has been adopted as the standard in CDI applications.

6.1 MVC Design pattern

The following diagram shows how classes are layered when implementing the MVC design pattern.

California Dept. of Insurance
MVC Design Pattern Implementation

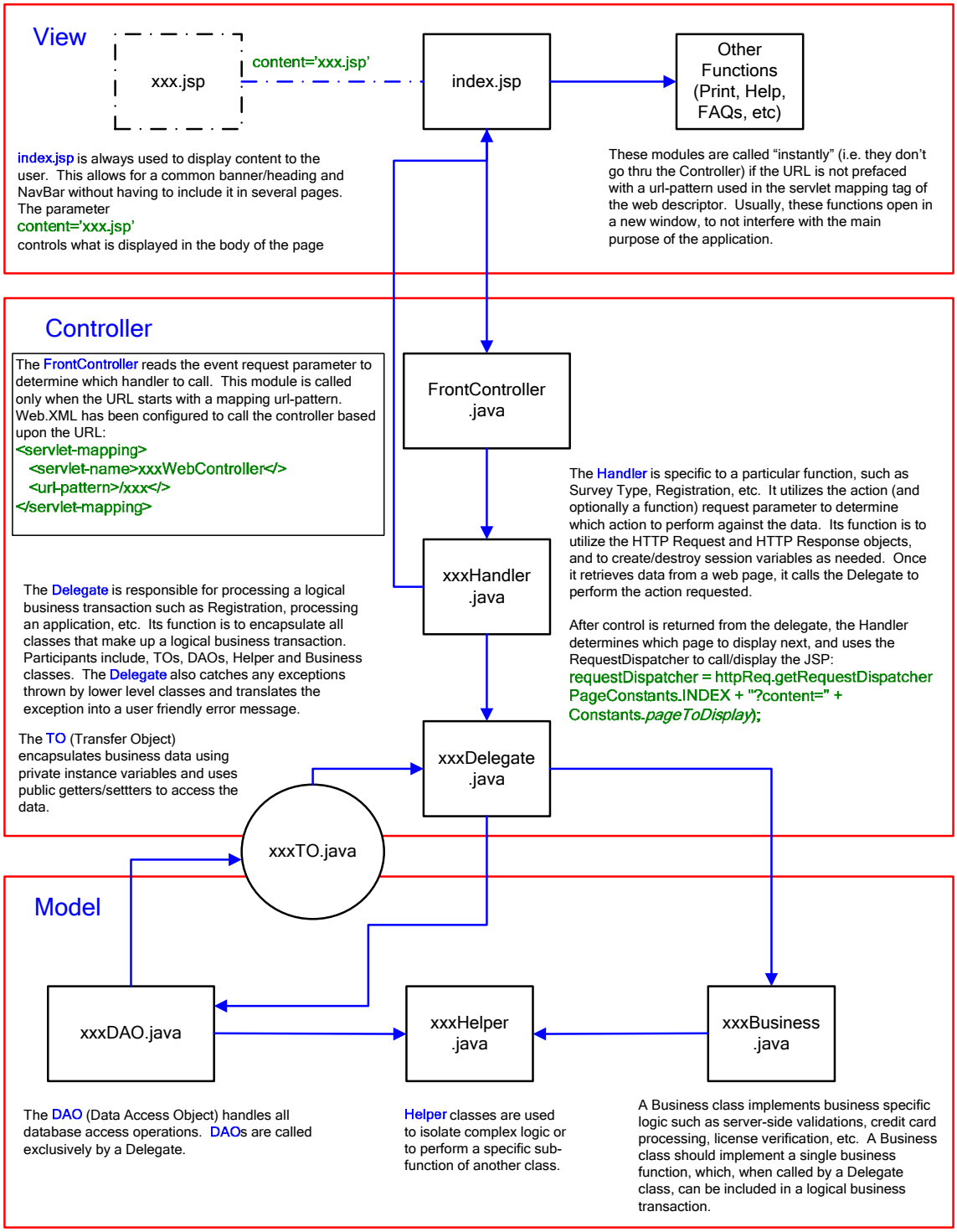

Figure 1 Explanation of the MVC architectural design

6.2 Composite View Design Pattern

The composite view pattern builds composite views from multiple subviews. The composite view pattern implements a templating mechanism to provide a consistent look and feel for all pages. The template itself is a JSP page, with parameters for the parts that need to change with each page. In CDI's web application template, the subviews Url are fed the container structure from parameters which are extracted from the XML page properties files.

The composite view container implemented for CDI is divided in six subviews: header/banner, side navigation bar, application title, page title, main content and footer. In the current implementation all subviews are static except the navigation bar and main content subviews.

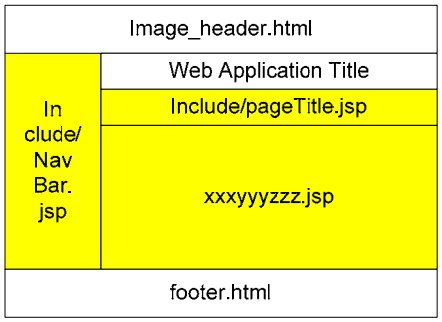

Figure 2 Structure of Composite View Container

7. Quick Setup

7.1 Setting up Log4j

The sample log4j configuration file included in *application template* instructs log4j to write logging messages to a file and to email the messages using a SMTP server. The “CA” element logs messages into the console.

Open *PROJECT_HOME*\cong\log4j.xml and look for element

```
<param name="file" value="../logs/Log4j_template.log"/>.
```

This is the path location to the file where the log messages are going to be written. Replace substring “template” with your own (application’s name is a good choice). If you don’t change the name, it’s very likely you will be mixing log messages with another application.

Locate the SMTPAppender node. Locate child element “param” with an attribute type named “name” and set “to”.

```
<param name="To" value="aguilar-barajasc@insurance.ca.gov,
aguilar-barajasc@insurance.ca.gov" />
```

This element contains the email address where the log messages are going to be sent. Replace the email addresses at your convenience. The SMTPappender contains a filter for logging FATAL messages only. Therefore, any debug, info, warn, error messages won’t be logged using the email appender.

If you need more information about log4j, you can read “Logging with log4j—An Efficient Way to Log Java Applications” available at http://www.developer.com/open/article.php/10930_3097221_1 or “Report Application errors by email” http://www.onjava.com/pub/a/onjava/2004/09/29/smtp-logging.html

7.2 Setting up Application Parameters File

The application parameter file contains configurable parameters for the application.

The type of parameters included in the CDI's web application template includes parameters needed for in view layer of the MVC (view composite container and controller), database parameters and email parameter. However, the set of parameter included in the sample file can be expanded to include any custom variables since the parameters value by using the appropriate API call.

List of elements available in the file:

View Composite Container related parameters:

intranet – Determines whether the intranet or internet portion of the web application should be activated.
applicationTitle – used by the composite view container to fill application name section

Database parameters:

web_app – indicates to the database connection factory whether it should use the database connection pool available on the JNDI or establish database connections on its own.

If "web_app" was set to true:
JNDI – JNDI name of the database connection pool

Otherwise:

production – controls which dbUrl should be used
oracle-driverClass – holds driver class name
db-url-prod – dbUrl used for connecting to production
db-url-test – dbUrl used for connecting to test
db-username – user to be logged in as (schema name)
db-password – the password that coincides with db-username

Miscellaneous parameters:

emailer-application-settings (tree) - Email settings for Emailer class.

File location is *PROJECT_HOME*\conf\ applicationParameters.xml.

Note: Node names must be unique (rule is not enforced programmatically).

7.3 Setting up the Page Properties Files

The pageProperties xml file allows the web application to fill the sections of the view container which make up the composite container declaratively.

The idea behind using a page properties file is to declare in an external file the parameters that are needed by the composite view container to fill the sections of the container.

Declaring the parameter values in a XML file is preferred over declaring them programmatically. Implementation is located in the PageProperties class. Developers only need to understand the metadata in the XML file and don't need to worry about the details of the class implementation.

The number of <page> node elements represents the number of JSP pages that are available in the web application. Therefore, for each screen you need in your application, you need to include a page node in the page properties file.

A description of the elements is listed below:

- **keyName**
  A unique name that is used as an identifier for the web page. If you use duplicate elements, an exception will be thrown. The key name is used as the key value in a hash table. Example: <keyName>InternetIndex</keyName>
- **url**
  The project's relative url path to the content subview. Example: <url>Internet/index.jsp</url>
- **windowTitle**
  The title to be displayed in the browser window. Example: <title>Welcome to Internet Index Page</title>
- **pageTitle**
  The wording to be displayed in the "page title" subview of the container. Example: <pageTitle>Welcome to Internet Index Page</pageTitle>
- **blankNavBar (optional)**
  If present on a page node, the navigation bar will be excluded from the container. This is the only field that can be omitted. If this element is omitted, container displays the navigation bar. Example: <blankNavBar>true</blankNavBar>

The only element that can be omitted in the page properties file is blankNavBar. If you miss one of the other elements, the application will throw an exception, which will be categorized as fatal; therefore, log4j will

generate an email informing the issue (remember to update the email address in the Log4J configuration file).

Sample page properties node:

```
<page>
    <keyName>ErrorPage</keyName>
    <url>Shared/error.jsp</url>
    <windowTitle>Error Page</windowTitle>
    <pageTitle>Internal Application Error </pageTitle>
</page>
```

The following diagram lists all the screens that are used in *application template.*

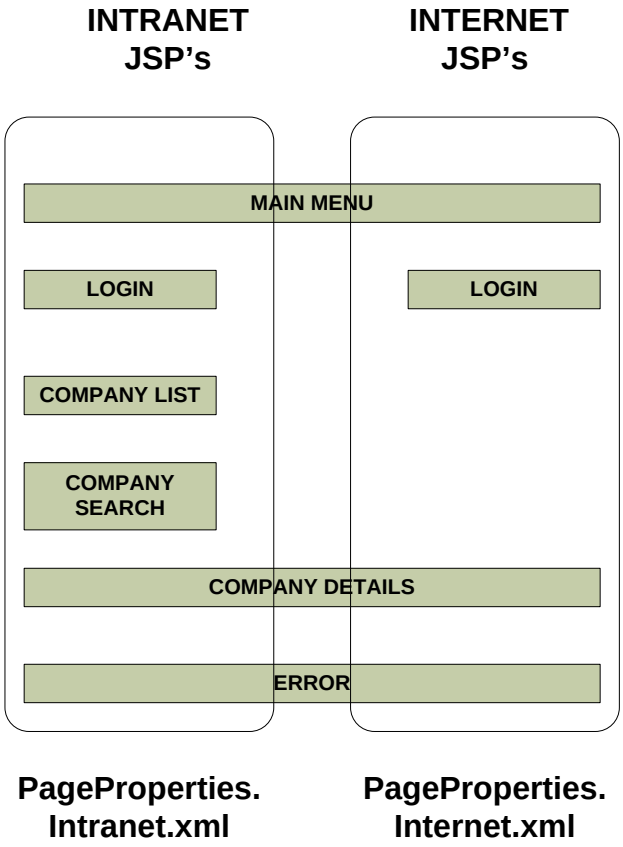

Figure 3 Page Properties Nodes in Application Template

Main Menu, Company Details and Error pages are shared screens between the two scopes of the application. These three pages are stored in the "Shared" folder. There are two versions of the login page, one on the "intranet: scope and one in the internet "scope". Company list, company search are located only in the intranet portion, so they can't be accessed from the internet scope.

The PageProperties class parses the pageproperties XML files and services the other classes with the information stored in the XML page properties files.

8. Application Architecture

8.1 Request Dispatching Login in Application Template

The architecture sampled in the *CDI's application template* has been adopted from all the lessons learned since the first release of the application back in 2006.

The main idea of the application architecture is that the controller and the handlers work together to identify a request and generate the response to the client.

Furthermore, the controller identifies a user request by parsing the event parameter and invoking a handler class. Then, the handler class parses a function parameter to identify which delegate function to invoke.

Therefore, the event and function parameter are joined to identify a user request and invoke the business components.

The sampled architecture used in the *CDI's application template* is described in the following diagram:

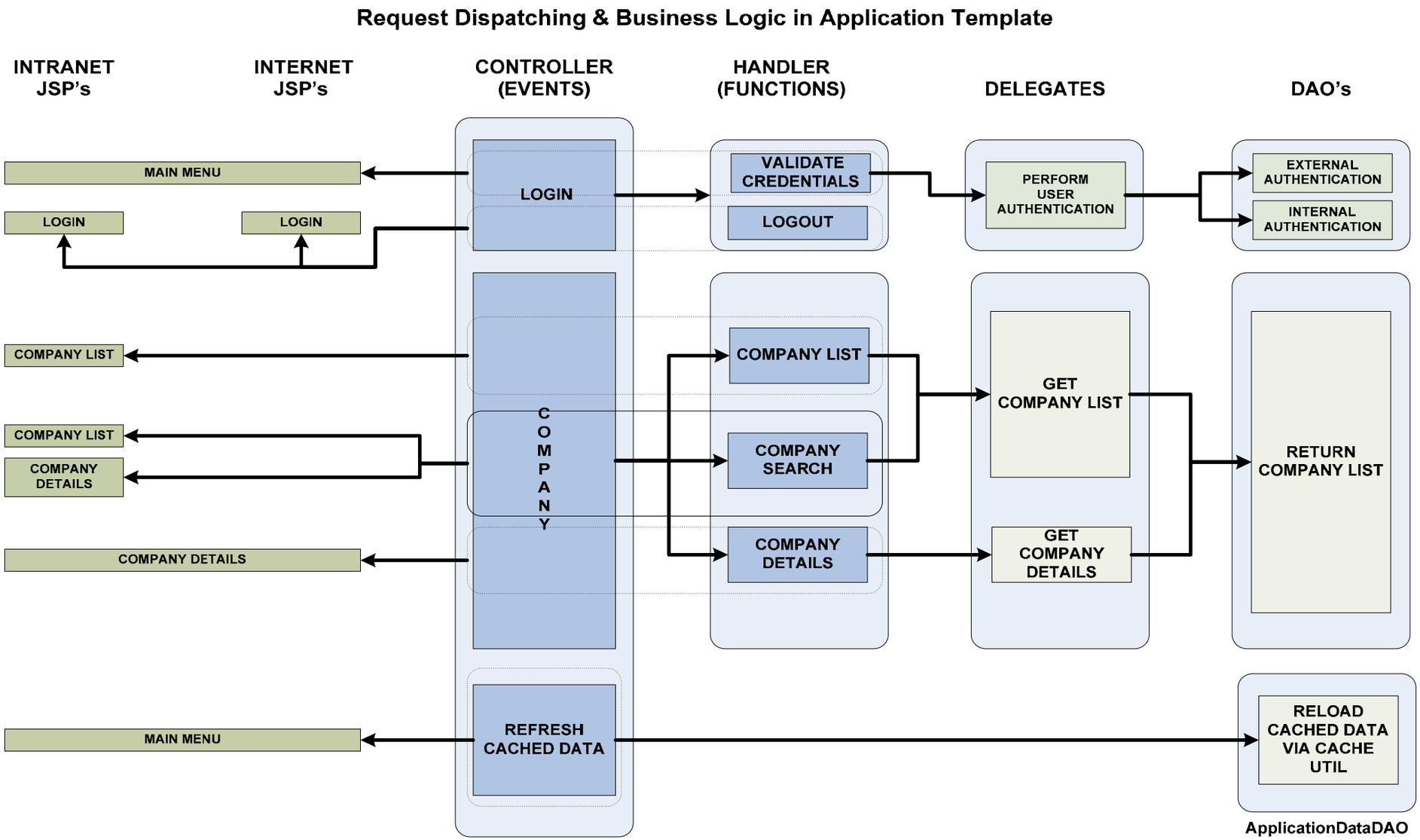

Figure 4 Recommended Request Dispatching design in CDI's web application

8.2 Understanding Class Relationships and Responsibilities

A description to the class responsibilities can be found on the HTML based API reference document has been created with the javadoc utility. HTML files are located in *JDEV__MY_WORK__DIR/TemplateWS /TemplatePrj /javadoc/index.html*.

Class relationships are described in the following UML diagram created for the template application project.

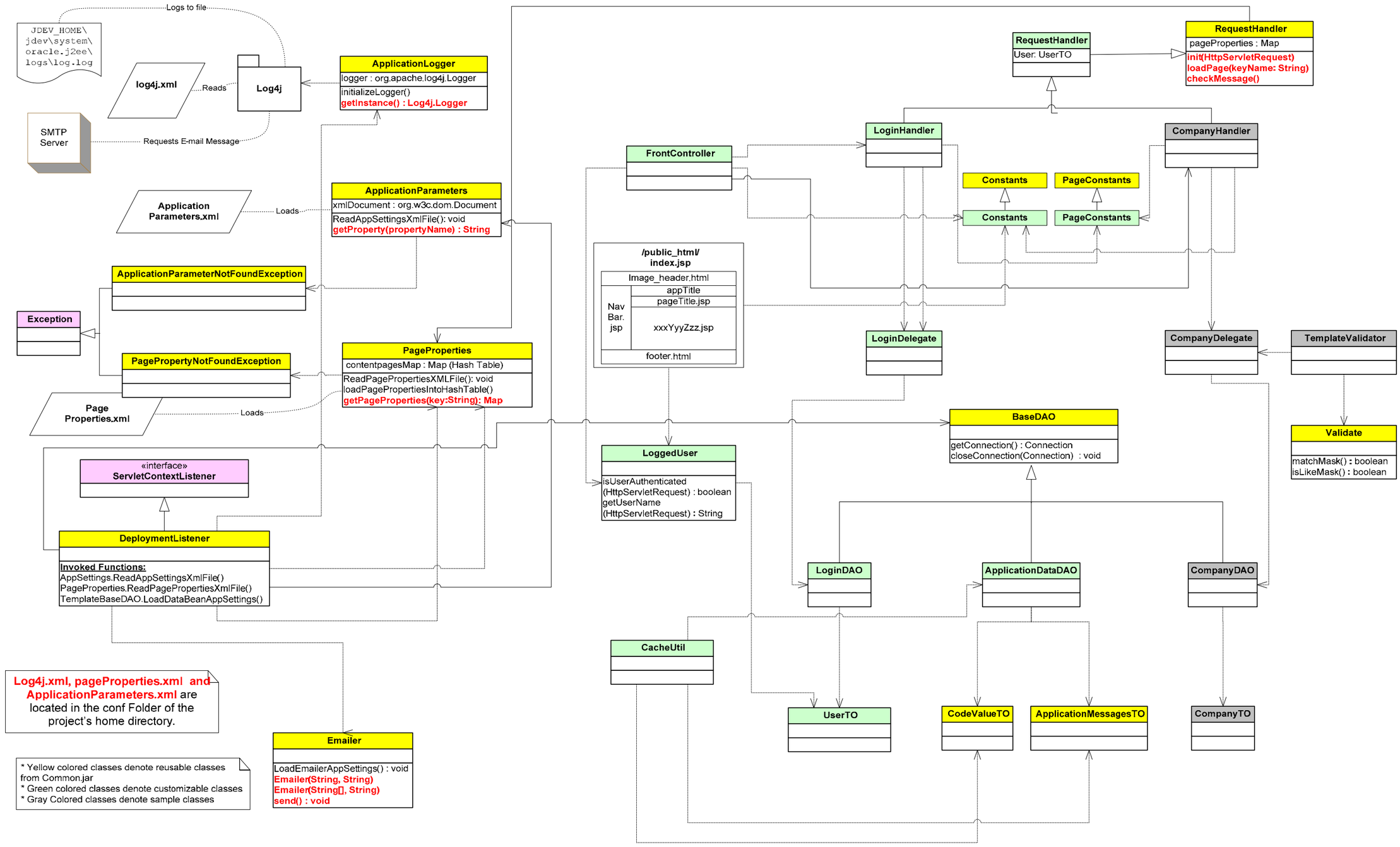

Figure 5 Class Diagram of CDI's web application template

8.3 Libraries

Application Template uses an in-house library that encloses the classes that are common to all applications. This library has been packaged into a JAR file (java archive) that has been called common.jar.

The library contains the compiled classes, the source code and the javadoc documentation. Application template has been set up for accessing/executing the contents of the library when browsing template's source code or when debugging at run time.

This library has already been attached to application template, so you don't need to do anything to stick it into your project if you application template as your base project.

Classes like DeploymentListener, PageProperties, ApplicationParameters, ApplicationLogger and RequestHandler are mandatory to use, so the same design is shared by all applications.

Refer to class diagram of Application template for identify how class are used in application template.

The following diagram shows the libraries content included in application template.

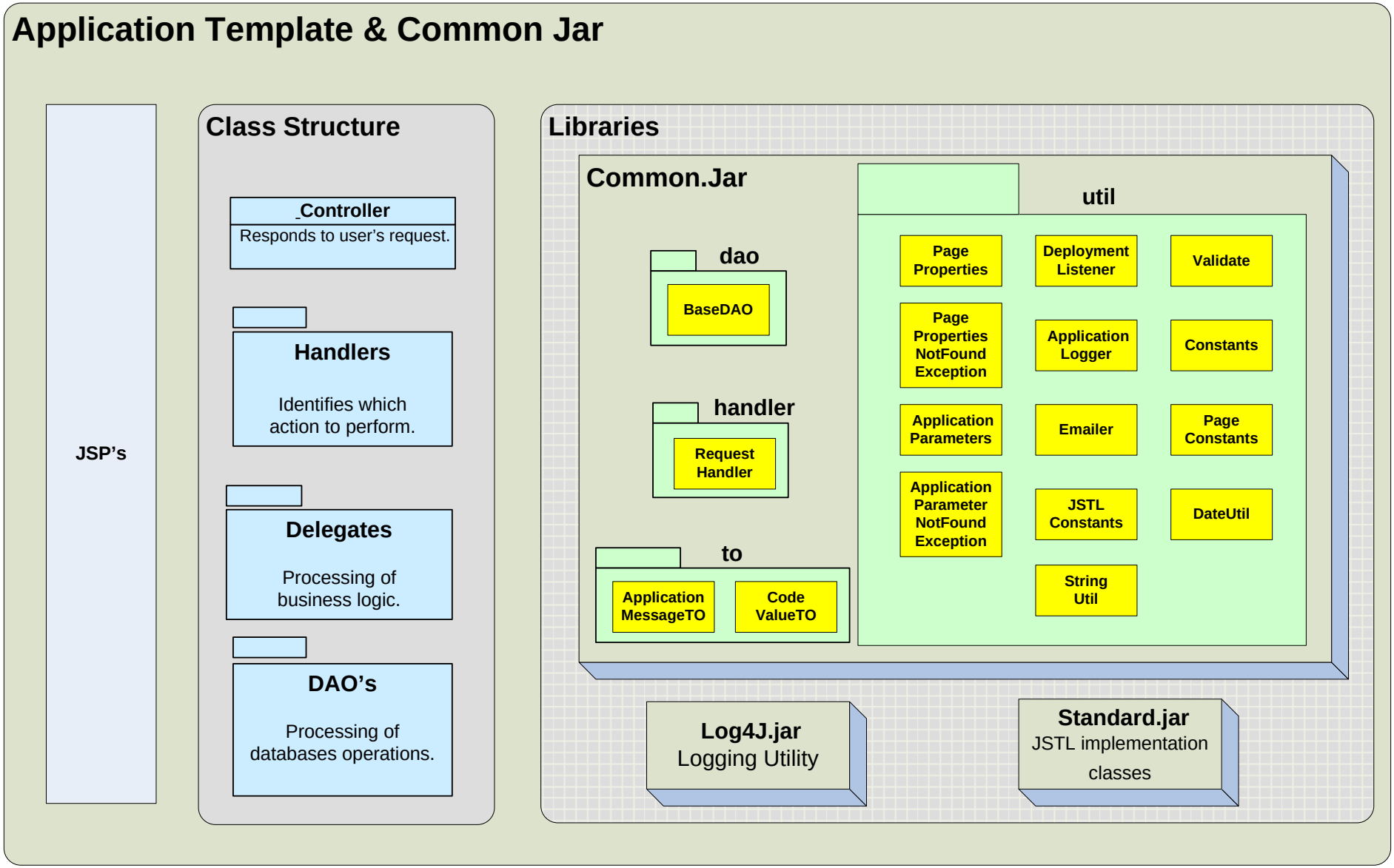

Figure 6 Contents of Jar File and relationship with CDI's web application template

9. Security Framework

Application template samples a controller security mechanism retrieving user credentials from the COSMOS user repository.

Controller security is implemented in the controller layer of the MVC pattern. Therefore, the controller class is responsible for protecting the handler classes and is considered the single point of user validation.

Application security is divided in two major components: authentication and authorization. Authentication consists in validating user's credentials against a user's repository, which could be a database table, an XML file, LDAP directory, etc. Authorization refers to the user's ability to access resources based on business rules.

The following state diagram depicts the transitions happening when protecting a resource using controller security.

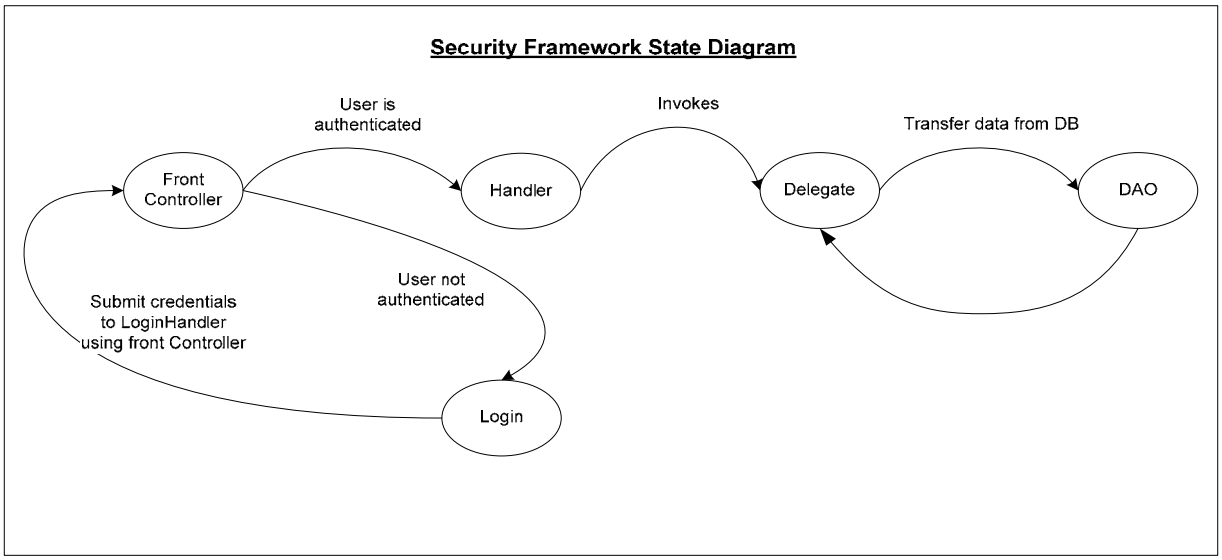

Figure 7 State Diagram for Security Mechanism

Application template uses a HTML login form for entering user's credentials. The following components compose the authentication schema: {Internet, Intranet}/login.jsp, LoginHandler, LoginDelegate, LoginDAO and LoginTO. If you need to change the login tables, you can simply edit LoginDAO where you would change the SQL string to match the tables that match your user's repository.

For a better understanding of how authorization and authentication is implemented in *application template,* the following two sequence diagrams show the time flow of function calls between classes.

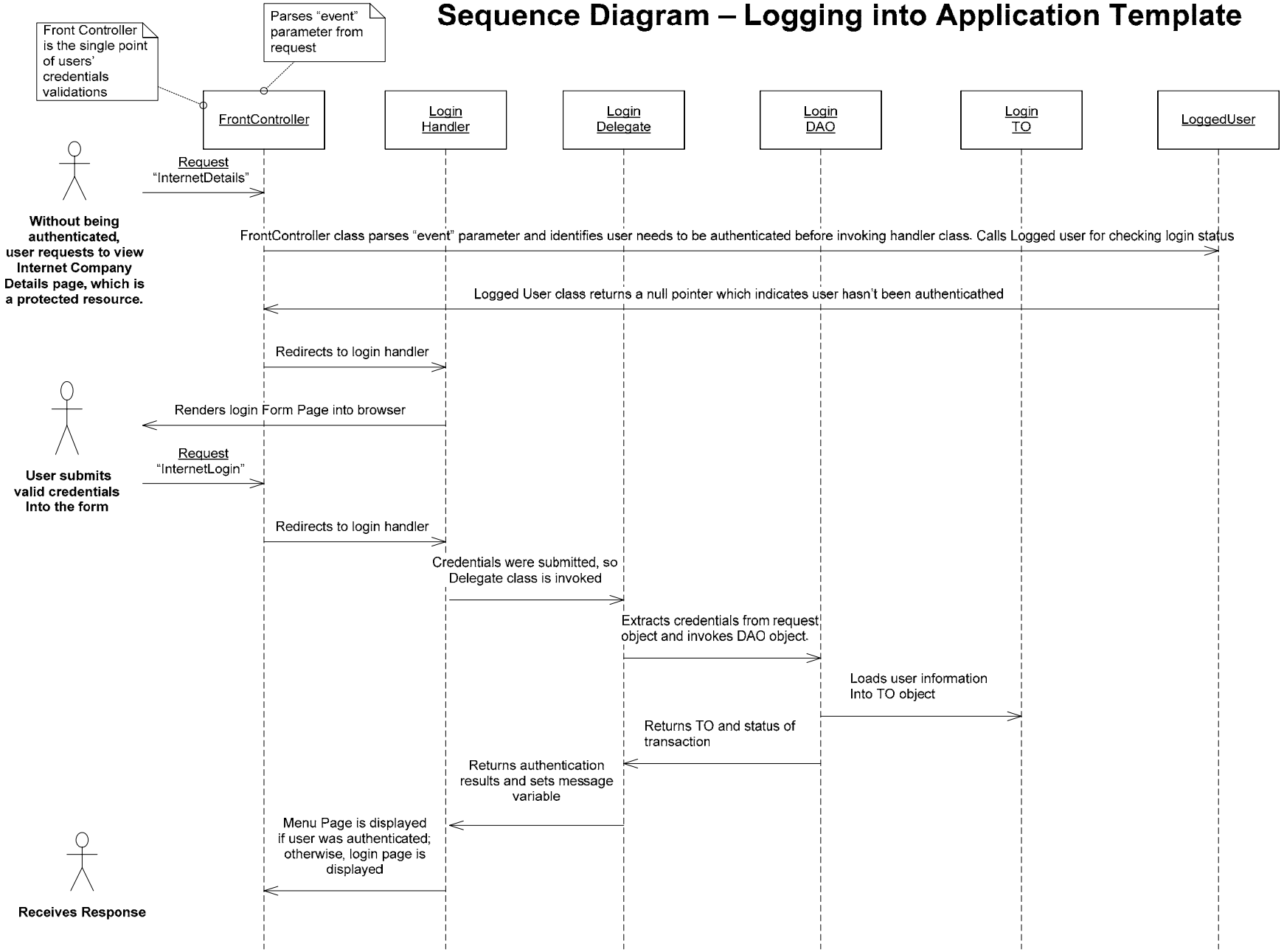

Figure 8 Sequence Diagram: Logging into application template

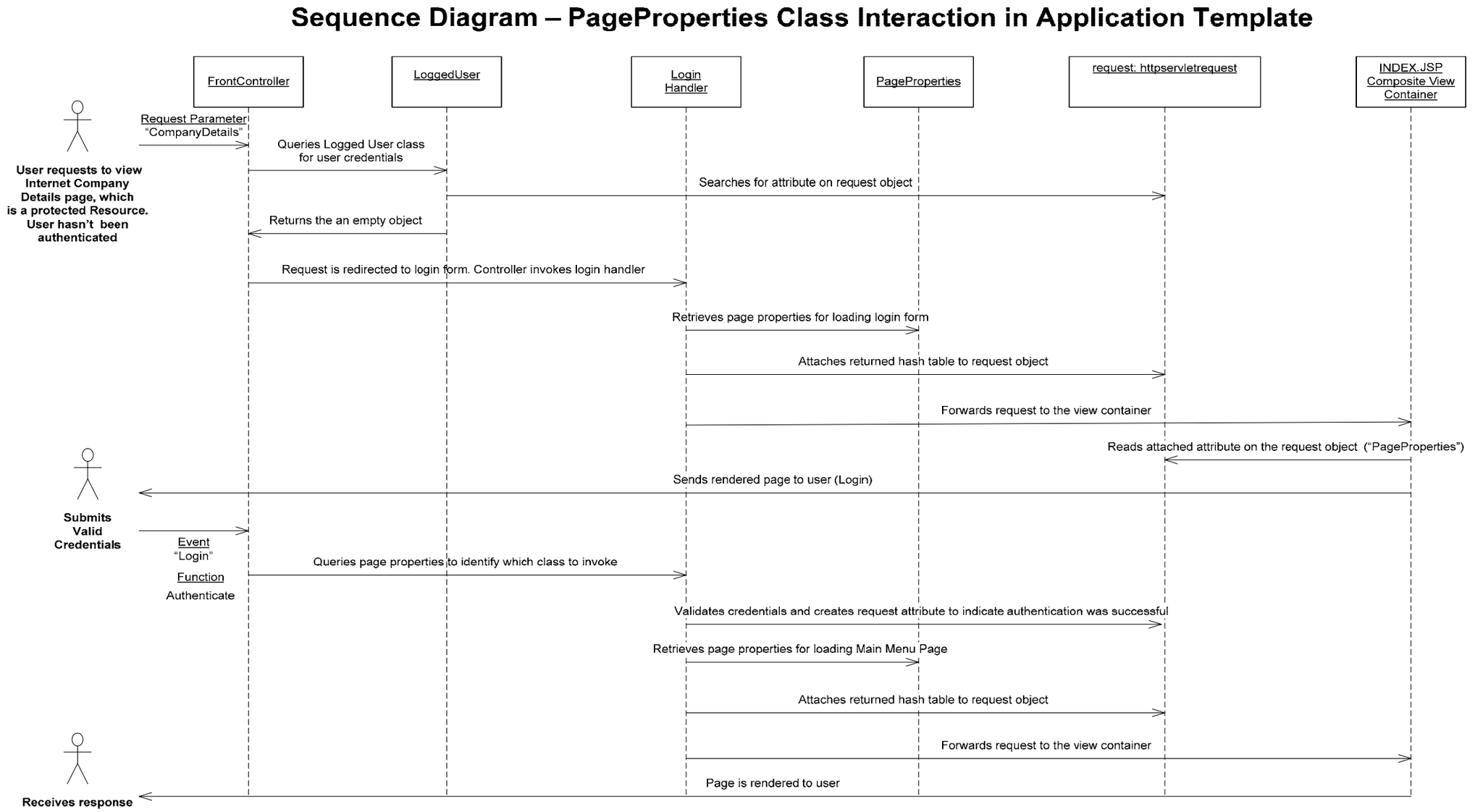

Figure 9 Sequence Diagram: PageProperties Class Interactions

View layer security refers to implementing a security mechanism in the content subview of the composite view pattern. Basically, the idea is to protect individually the JSP pages that are used to fill the content subview of the view container. That is because JSP pages can be invoked directly without calling the front controller and the view container. Hackers would need only to figure out the corresponding URL for accessing the page.

In application template all pages are considered protected, except the container which is the parent Jsp and it is not referred as a content element in the page properties file, On the protected resources, there is logic that checks whether the page came through controller or is accessed directly. If the page is being accessed directly; request is simply forwarded to the front controller.

10. Following Best Practices

10.1 Returning Data from Delegate and DAO classes

Single TO object should be used for single row queries retrievals. For multiple row retrievals, an array list should be used for storing retrieved data.

In *application template*, when the company list resource is requested, the DAO function returns a collection of TOs (CompanyDAO.java loadCompanyList() : Collection) just like the CompanyDelegate class (getCompanyList function). A Single TO is returned when invoking the getCompanyDetails function in the CompanyDelegate class

10.2 Store Application Messages Externally

Application messages should be stored on a database table for easy updating of the applications message without redeploying the application. For not overwhelming the web application with too many database calls, the notion of a Cache Data Manager utility should be created. This utility retrieves the data from the database and caches it in memory for fast and easy accessibility by other components of the application. A cached data manager utility should provide the functionality of refreshing the contents of the cached data. The CacheUtil class demonstrates the structure and functionality of a cached data manager class. It implements the singleton pattern.

In *application template*, CacheUtil retrieves all the application messages and loads them in memory. It also loads the state names with their code values.

10.3 Application Messages

To better manage application messages, they have been abstracted into a Java Bean class called ApplicationMessageTO. This class contains three instance variables:

- Msg_id: A unique identifier to make messages distinctive.
- Msg_text: The actual text message
- Msg_ctgry_cd: The category code for the current message.

The message category code could be either "Severe", "Error" or "Success". The "Severe" category identifies those fatal messages that make the application unavailable (e.g. database is down). The Error category represents messages that are errors, but they are handled by the application and they are returned to the user (user entered a date that doesn't fall in a certain timeframe). Success errors are supposed to report that operations have been executed successfully.

10.4 Displaying Application Messages

Application Template has a unique point where application messages are displayed. This section is located in the composite view container (public_html/index.jsp). Having a single point for displaying application messages requires less maintenance effort when making formatting changes to this section.

10.5 Error Handling

There are two points where exceptions are being handled: the front controller and the delegate classes.

The front controller class is responsible for catching any unhandled exception that occurred in the handlers or run time errors

Delegate classes handle any exceptions that are internally thrown or propagated from DAOs. Propagation of exceptions from Delegate classes to handler classes should be avoided because delegate classes implement a messaging mechanism with handlers to determine the status of any function invocation in the delegate.

To help out with messaging mechanism the handlers extend the Request handler object in the common Jar library which provides helper functions to perform message checking between handlers and delegates.

The mechanism logic is as follows: delegate classes should have a retrievable instance variable to identify the status of an instantiated delegate object. As an example, application template's delegates have an ApplicationMessageTO instance variable. If the delegate's instance variable is null or contains an ApplicationMessageTO whose category code is "success" or "warn" then that means the instantiated delegate object hasn't experienced any issues during the invocation. But, if the category code is "Error" or "Severe" then that would mean delegate’s invocation experienced issues and handler class needs to be informed to take further action.

10.6 Validation

It is recommended to create a helper validation class to isolate the validation business rules from the delegates. *Application Template* contains TemplateValidator, a class that is composed of all the validation routines for the application. TemplateValidator takes advantage of using the Validate class, which contains a set of functions that determine if a string is formatted appropriately according to a regular expression. TemplateValidator class samples how to use the API of Validate.

11. Check List for Running Application Project

- Edit Log4j configuration file (conf/log4j.xml)
  - Edit “to” and “file” nodes. If this is omitted, email error alerts will be missed
  - Refer to “Setting up Log4j” section on this document
- Edit Application Configuration File (conf/app_conf.xml)
  - Edit database connection settings
  - Edit intranet variable to define which portion of the application should start
  - Edit Email settings if in need of using Emailer class
- Edit “Page Properties” files while pages are being added to the application (conf/PageProperties.internet.xml and conf/PageProperties.intranet.xml).
- Application template uses JSTL (java standard tag library). Jdeveloper Preview Version 10.13 needs a small configuration tweak before executing JSP pages that include JSTL. File JDEV_HOME/j2ee/home/jsp/lib/taglib/standard.jar needs to be deleted because it generates a class loading mismatch with the WEB-INF/lib/standard.jar included in the web app that is being executed.