Jax Framework Guide

Release 1.0

Ron Cemer

April 03, 2016

CONTENTS

An Open-Source, AJAX-based Web Application Framework

Author: Ronald Cemer

Version: 2014-05-26.01

CHAPTER

ONE

REVISION HISTORY

2014-05-26.01 - Ron Cemer - Add documentation for AJAXSearchGrid features in popupSearch/* and crudSearch sections. Clean up formatting.

2013-03-12.01 - Ron Cemer - Convert to reStructuredText.

2011-10-01.01 - Ron Cemer - Initial version.

CHAPTER

TWO

INTRODUCTION

2.1 Why was this document written?

This document was written in order to introduce new users to Jax Framework ™, and to cover the various subsystems within the framework.

2.2 Audience

This document is targeted at the web application developer who is interested in rapidly developing fast, efficient web applications which have the responsiveness and feel of desktop applications.

2.3 New versions of this document

For the latest version of this document, see the Jax Framework |tradel Documentation Page.

For the latest version of Jax Framework ™, see the SourceForge.net Project Page.

2.4 Contributions

Jax Framework ™ was developed by veteran professional software developer Ronald Cemer. Third-party packages included in Jax Framework ™ include:

- AngularJS
- Twitter Bootstrap
- jQuery
- jQuery-UI
- jQuery DataTables [DEPRECATED]
- jQuery ColorBox
- jQuery Field
- Datejs
- Spyc
- phpdaogen

- phpreportgen

2.5 Feedback

Missing information, missing links, missing characters? Contact the maintainers of this package at the SourceForge project page.

2.6 Copyright information

2.7 What do you need?

In order to use Jax Framework ™, you need a web server (Apache 2.x is preferred) running PHP version 5.1.6 or later. PHP version 5.3.x or later is preferred.

AJAX functionality involves both server-side and client-side code. The server-side code is written in PHP, and the client-side code (which runs in the browser) is written in JavaScript.

In order to minimize the amount of code which needs to be hand-written, Jax Framework ™ includes code generators to generate both the Data Access Objects (DAO) and standard Create/Retrieve/Update/Delete (CRUD) pages, driven from YAML files. To use this functionality, the user needs a basic understanding of the YAML format. The specifics of the YAML format used by Jax Framework ™ are covered in this document.

CHAPTER

THREE

JAX FRAMEWORK ™ SUBSYSTEMS OVERVIEW

In this chapter, we will introduce the various subsystems within Jax Framework ™.

3.1 Data Access Objects with phpdaogen

The phpdaogen project implements a simple Data Access Objects pattern using a code generator.

Phpdaogen handles database schema (DDL) creation, introspection and transformation functionality; generates DAO classes; and provides a thin layer between the application and the database.

Your database schema can be created directly in the database using your database design tool of choice, and then exported to YAML or XML DDL file(s). The preferable approach is to use the YAML or XML DDL file(s) as the master copy. This allows you to automatically transform the DDL of a live database server at deployment time, to match the schema defined in the DDL file(s).

DDL file(s) can be in YAML format (filename.ddl.yaml or filename.ddl.yml), or XML format (filename.ddl.xml). Any tool which can read a DDL file, can also read the schema directly from the database. This is generally slower though, and requires you to specify your database connection parameters on the command line.

Phpdaogen provides a code generator which generates classes for each table it finds in the schema. The two classes which will be used by application code are the data object class the data access object (DAO) class. By default, these classes are generated only once, and extend corresponding abstract classes which are re-generated each time the code generator is run. This allows the data object and DAO (concrete) classes to be extended with custom functionality, should you desire. If the code generator is run with the -noabstract command line option, then no abstract classes will be generated; all of the logic will be put straight into the concreate data object and DAO classes. You can use this option if you know that you will never want to customize the data object classes or DAO classes. When generating abstract classes (the default), all of the generated code goes into the abstract classes, leaving the concrete classes completely empty for ease of customization. The abstract data object class is a very simple class containing public variables for the columns in the table, one function to set all fields to default, and one function to load all fields from an associative array. The abstract data access object class includes functions to insert, update and delete rows in the table, a load() function to load a row by its primary key, and functions to find rows by various columns.

Phpdaogen also provides a thin layer between the application and the database, enabling the application and the DAO classes to access the database using a consistent interface, regardless of the database vendor. Phpdaogen provides the following classes and interfaces:

- Connection class (abstract) and its concrete subclasses: Provides a consistent interface for a database connection, and queries performed on that connection. All queries require a PreparedStatement instance.
- PreparedStatement class: Provides an implementation of (simulated) prepared statements, where question marks (?) are used for placeholders. Parameters are substituted for the placeholders, and automatically escaped according to the database engine's escaping rules. When used properly, this greatly minimizes or eliminates the possibility of SQL injection attacks.

- DAOCache (interface) and its implementation classes: Provides a way to cache the results of queries performed using DAO classes, to reduce the database workload for often-executed queries which do not update. There are concrete implementations for filesystem caching and memcached caching.
- ChildRowUpdater class: Greatly simplifies the updating of so-called 'child' rows, which are one-to-many relations, when saving a 'parent' row.
- DDL class: Provides database schema (DDL) loading, introspection, transformation, synchronization and exporting/serialization functionality.

Phpdaogen is NOT a full Object/Relational (O/R) mapper, and does not emulate missing features in some databases. As a result, it will generally be much faster than an O/R mapper. However, care should be taken to design tables and write SQL queries in such a way that they will run properly across all database engines you want to support in your application. Using mixed case (e.g. camelCase) table names and column names may work fine in MySQL, for example, but they will utterly fail in PostgreSQL, therefore it is advised to use lowercase names for all database objects, and separate words within the names with underscores if desired. Also note that Sybase and MS SQL Server are missing support for the standard SQL date type, so if you use this type, you may run into problems because phpdaogen may substitue the datetime type for the date type when talking to Sybase or MS SQL Server.

3.2 AngularJS

Jax Framework ™ includes AngularJS, and uses it internally for some of its components, including jax-grid:

- AngularJS

3.3 Twitter Bootstrap

Jax Framework ™ includes the full distribution of Twitter Bootstrap, both CSS and JavaScripot components, and uses Bootstrap as its CSS foundation:

- Twitter Bootstrap: Twitter Bootstrap

3.4 jQuery and jQuery-UI Components

Jax Framework ™ includes the following jQuery-related Components:

- jQuery: The core jQuery library
- jQuery-UI: The User Interface library for jQuery
- **jQuery DataTables: [DEPRECATED] Provides searchable, sortable tables for tabular data.** Replaced by jax-grid.js and ajaxSearchGrid.js.
- **jQuery ColorBox: Enables opening a page in a modal pop-up frame within another page.** Also enables modal dialog-like interaction with the user.
- jQuery Field: Provides some convenience functions for dealing with forms and fields.

It is recommended that the reader have at last a cursory understanding of what each of these components does. An in-depth understanding of these components is only required for more advanced tasks within Jax Framework ™, since the code generators provided with Jax Framework ™ will automate the normal-use-case usage of most of these components. Only custom use-cases will need to be coded by hand.

3.5 Search and CRUD Generators

One of the many nice features of Jax Framework ™, is the ability to search for rows in a table using an interactive search table, and to have the search results appear as you type, without using page reloads. This is accomplished by using the AJAXSearchGrid class, jax-grid component and corresponding back-end search scripts. The job of the Search Generator is to generate these search scripts, along with single-row loader scripts which are used to retrieve a single row by its primary key. The search generator is driven by YAML files in the gencfg subdirectory, where the filename format is tablename.yaml (where tablename is the name of the database table).

Perhaps the most important user-visible piece of core functionality provided by Jax Framwork ™ is CRUD pages. These pages allow Creation, Retrieval, Updating and Deletion of database rows. The CRUD pages are generated by the CRUD generator, driven by the table-specific YAML file (see previous paragraph). This makes it easy to create a table, design a CRUD page for the table, and be editing rows in the table in short order.

Each YAML file in the gencfg directory can have entries for the search generator, the CRUD generator, both, or neither. There can be zero or more entries for the search generator, and zero or more entries for the CRUD generator. Each search and CRUD in a specific YAML file must be for the table whose name the YAML file is named after.

3.6 Search and Loader Includes

The search generator outputs search includes and loader includes, as well as JavaScript files for pop-up searches. A pop-up search is an inline searchable jax-grid component which pops up when an icon image is clicked, and allows the user to select a related row whose id populates a field in a form in order to establish a relation between the row being edited in the form, and a related row in another table. For example, when entering a client's order, the user may know the client's name but not their client Id. The client Id is what is needed to link the order to the proper client. So on the form, a pop-up search icon would be provided which, when clicked, expands a searchable jax-grid component which is searching the client table. The user enters a few characters of the client name. When the proper client is visible in the table, the user clicks on the appropriate row in the table. The search table disappears, and the appropriate client Id is plugged into the form. The user then proceeds with order entry.

The job of the search generator is to create both the client-side JavaScript files and the server-side PHP includes to enable these jax-grid searches, as well as to generate server-side, single-row loader include files which are used to locate and retrieve single rows. These loader includes are used for populating the form in the CRUD page when a request is made to edit an existing database row, as well as for looking up related rows when an identifier column in a form is changed (e.g. when a client Id is selected or entered, in order to display the corresponding client name within the form).

3.7 Filters and Validators

Jax Framework ™ includes classes to perform input filtering and validation. This is very handy in CRUD pages, since it allows you to specify most or all of your form filtering and validation in the YAML file, greatly reducing the amount of coding required to get reliable, functional CRUD pages working.

Filters include integer, decimal, trim, uppercase, lowercase, etc. You can add your own custom filter classes by extending the Filter abstract class.

Validators include not empty, not zero, range, min/max length, list of valid values, email address, no duplicates, foreign key, etc. You can add your own custom validator classes by extending the Validator abstract class.

Validators generate field-specific error messages which appear above their corresponding fields in the form, and prevent the user from saving the contents of the form until there are no validation errors.

3.8 Reporting with phpreportgen

The phpreportgen project implements a nested report generator, along with an image-based charting class. With phpreportgen, you set up the report structure and output format, issue your query, and feed it the rows one at a time. Phpreportgen takes care of formatting, totalling and page breaks (depending on output format).

3.9 Other Classes and Includes

There are PHP classes for Token Replacement within strings, including some specific filtering/escaping functionality (HTML/XML escaping, URL escaping, Numeric with fixed scale, Yes/No).

There is a customizable Permissions class which, by default, ties into the example user, role, permission (and others) tables which are provided in the skeleton directory. This class, together with these sample tables, provide a complete, role-based permission model, allowing the administrator to control which pages are visible to which roles, and which users belong to which roles. Individual permissions can also be used to enable/disable specific functionality within individual pages.

There is a customizable ConnectionFactory class which should always be used to establish new database connections. This factory class creates and returns instances of the Connection abstract class. It is desirable that you modify the ConnectionFactory class to read a configuration file of your choice (although not visible under the web server's document root, so that users cannot see your database configuration – for security reasons). The configurable parameters should be which Connection subclass to use (MySQLConnection [deprecated], MySQLiConnection, PostgreSQLConnection, etc.), the database server host or IP address, the username, the password, and the initial database to use. The default implementation of this class has all of these values hard-coded based on the example database which is created by following the instructions in INSTALL.txt in the root of the Jax Framework ™ project.

Jax Framework ™ contains a set of customizable includes for requiring user login to secure selected pages. The requireLogin.include.php include, when included in a page, requires that the user be logged in in order to access that page. If the user is not logged in, it presents a login form, defined in loginForm.include.php. Once the user is logged in, the page is executed as normal (assuming the user has proper permissions for that page). These includes can be customized to your liking. By default, they use the sample user/permission/group tables which are included in the skeleton directory.

Jax Framework ™ includes validation.include.php, which performs some simple validation and error reporting functions which are used by the CRUD pages, and requestURL.include.php which is used to retrieve the full URL of the current PHP script, with or without the query string.

Page layout templates are used to define the overall layout and navigation of the site. These are customizable for your application. These include header.include.php, footer.include.php, topNav.include.php, leftNav.include.php.

3.10 Model-View-Controller

Jax Framework ™ takes Model-View-Controller (MVC) concepts and applies them to AJAX. The implementation is very simple and easy to follow. The three tiers are separated as follows:

- Model: The database schema, DAO and data object classes generated by phpdaogen, search and loader PHP includes, and individual PHP pages which are viewed by users, are all part of the data model. These encompass all of the business logic.
- View: The *_view.include.php, as well as the page layout templates, are all part of the view. These encompass what the user sees.
- Controller: The *_controller.js files, as well as all other JavaScript files which are involved in user gesture processing, are part of the controller.

This is a very different layout than what you see with most MVC-oriented frameworks. This is because Jax Framework ™ is a completely AJAX-based framework which offloads as much of the user gesture processing as possible to the browser. The result is a very snappy and fast user interface, with the feel of a desktop application, all while running in a browser.

3.11 CRUD Callbacks

The CRUD generator creates files which look for specific callback functions, also called "hooks" and call them if they exist. This allows you to add your custom functionality on top of the existing CRUD functionality which the CRUD generator creates, allowing you to take advantage of the consistency and rapid development benefits of generated code, while still giving you a great degree of customizability.

For a given CRUD page (e.g. appuser.php), there are three generated files which are automatically re-generated each time the CRUD generator does its thing:

- A generated model file (e.g. generated/appuser_generated.php)
- A generated view file (e.g. generated/appuser_view_generated.php)
- A generated controller file (e.g. generated/appuser_controller_generated.js)

and three one-time-generated files which are only generated if they don't exist:

- A model hook file (e.g. appuser.include.php)
- A view hook file (e.g. appuser_view.include.php)
- A controller hook file (e.g. appuser_controller.js)

Each one-time-generated hook file can contain a specific set of hook functions (callbacks) which will be called when specific events occur, allowing you to customize the CRUD page extensively.

You can also put your own functions in each hook file and call them from within the hook callback functions.

3.12 Page Layout and Templates

Page layout templates are used to define the overall layout and navigation of the site. These are customizable for your application. These include header.include.php, footer.include.php, topNav.include.php, leftNav.include.php.

It is the responsibility of every page's view (*_view.include.php) to include header.include.php at the top of the page, emit that page's content, then include footer.include.php at the bottom.

You can customize header.include.php and footer.include.php to provide custom navigation for your site. You may wish to abandon topNav.include.php and leftNav.include.php altogether, replacing them with your own navigation. However, if you do this, be sure to properly implement the security checks which are implemented in these incldues, and also be sure to obey the $barePage PHP variable.

Some pages may be displayed inside a pop-up window or dialog. When this happens, it is not desirable for the page to include header, navigation or footer content. This can be suppressed by setting $barePage to true before the view is output. This can be conditional for pages which can appear on the normal site navigation and also within pop-up windows or dialogs, or unconditional for pages which always appear within pop-up windows or dialogs.

CHAPTER

FOUR

PHPDAOGEN

In this chapter, we will introduce you to the concept of Data Access Objects (DAO), and phpdaogen. We will also cover database schemas, and how phpdaogen handles them.

4.1 Connection and PreparedStatement

4.1.1 Introduction

One of the problems with plain PHP is that it has a different set of functions, some of which work differently than others, for accessing different Relational Database Management Systems (RDMS). There is one set of functions for accessing MySQL, another set of functions for accessing PostgreSQL, another set for accessing databases via ODBC, and so on.

One of the things which phpdaogen tries to do, is to abstract the database access out so that you can focus on more important things. It abstracts out the creation of new database connections; as well as the building and execution of SQL queries, including the proper escaping of literal values in those queries to prevent SQL injection attacks. Since SQL injection attacks are some of the most catastrophic and most exploited web application vulnerabilities, it is important to use safe SQL queries. Phpdaogen makes this easy for you.

Another problem with the database access layer of most applications, is that applications work best with objects but relational databases store things in rows and columns. There needs to be something unobtrusive which sits between the application and the RDBMS and does the translation between objects and rows/columns. Historically, this has been an Object/Relational (O/R) mapping tool. However, O/R mapping tools are very heavyweight libraries which often do more than what is absolutely needed. As a result, they tend to run more slowly than we would like. Phpdaogen provides good mapping between objects and the RDBMS, but is able to "get out of the way" when you just want to deal with the results of an SQL query or execute a specific SQL update statement directly on the RDBMS without fiddling with objects. So you end up with the best of both worlds: A fast, lightweight object layer, and the ability to do pure SQL when needed – without risking SQL injection attacks as long as you propery use PreparedStatement for all queries.

4.1.2 Connection

The Connection class is an abstract class. There is one subclass of Connection for each type of RDBMS supported by phpdaogen. In the examples, we use MySQL, so the ConnectionFactory class looks something like this:

```
...
class ConnectionFactory {
    public static function getConnection() {
        return new MySQLiConnection('localhost', 'jax', '123jAx321', 'jax');
    }
}
```

This is a very naive implementation, and is only provided as a starting point. What if you need to run your database on a different server, or use a database other than MySQL?

A better implementation might look something like this:

```
...
class ConnectionFactory {
    public static function getConnection() {
        // The config dir should NOT be anywhere under the document root!
        $dbSettingsFile = dirname(dirname(dirname(__FILE__))).'/config/dbsettings.php';
        if (file_exists($dbSettingsFile)) {
            include $dbSettingsFile;
        }
        unset($dbSettingsFile);
        if ( (!isset($dbConnectionClass)) \|| ($dbConnectionClass == '') ) {
            $dbConnectionClass = 'MySQLiConnection';
        }
        if ( (!isset($dbServer)) \|| ($dbServer == '') ) {
            $dbServer = 'localhost';
        }
        if ( (!isset($dbUsername)) \|| ($dbUsername == '') ) {
            $dbUsername = 'jax';
        }
        if ( (!isset($dbPassword)) \|| ($dbPassword == '') ) {
            $dbPassword = '123jAx321';
        }
        if ( (!isset($dbDatabase)) \|| ($dbDatabase == '') ) {
            $dbDatabase = 'jax';
        }
        if (!class_exists($dbConnectionClass, false)) {
            include(dirname(__FILE__).'/'.$dbConnectionClass.'.class.php');
        }
        $db = new $dbConnectionClass($dbServer, $dbUsername, $dbPassword, $dbDatabase);
        // Throw exceptions when we have a failed query or we try to free an invalid result set.
        $db->throwExceptionOnFailedQuery = true;
        $db->throwExceptionOnFailedFreeResult = true;
        return $db;
    }
}
```

Looking at the code, you can see how it's getting its connection parameters from a PHP script which (if it exists) is outside of the document root. In fact, it could exit anywhere on the server's filesystem as long as it's readable by the PHP interpreter at runtime. This gives you the flexibility of changing your configuration by creating/editing the dbsettings.php file, and setting specific variables to control the type of RDBMS, and the database connection parameters. Another option would be to make ConnectionFactory parse a configuration file using parse_ini_file(), and pick out the appropriate settings from the parsed configuration. That solution would probably run somewhat more slowly though.

When the application is done with a Connection instance, the application must call the Connection's close() function in order to close the database connection. The Connection instance can then be safely discarded. Re-using a closed Connection instance is not allowed.

4.1.3 PreparedStatement

The PreparedStatement class allows you to use templated SQL statements with question mark (?) charcters for placeholders where literal values belong. These literal values come from PHP variables which are set from various sources. To create a PreparedStatement, do something like this:

```
$ps = new PreparedStatement('select * from user where user_name = ?');
```

When the PreparedStatement is initially created, it will contain no parameters. For the PreparedStatement to be useable, it needs to have all parameters filled in (one parameter per question mark placeholder). Parameters are filled in from first to last, using the setBoolean(), setInt(), setFloat(), setDouble(), setString() and setBinary() functions. Which function is used for a given parameter depends on the parameter's SQL data type. To clear the parameter list in order to re-use the same PreparedStatement with new parameters, call its clearParams() function.

Many RDBMS support paging of query results, causing a query to return a subset of the entire rowset it would normally return. This is accomplished differently with different databases, so you can't count on your SQL statement to handle this. For example, MySQL uses the "limit" keyword which has one or two numeric parameters. If only a single parameter is specified for MySQL's limit keyword, it returns the first N rows of the result set, where N is the parameter value. If two parameters are passed with the limit keyword, the first parameter is the number of rows to skip and the second parameter is the maximum number of rows to return after skipping. PostgreSQL does something similar, except that it uses separate offeset and limit keywords. Sybase and MS SQL Server use the top keyword, and cannot skip initial rows. Because of these variations in how RDBMS handle this crucial piece of functionality which was omitted from the SQL standard, PreparedStatement has two additional arguments to its constructor: $selectOffset and $selectLimit. Both default to zero. If $selectOffset is zero, no initial rows are returned. If $selectLimit is zero, there is no limit to the number of rows returned. So to skip 20 rows and return no more than 5 rows after that, you would do something like this:

```
$ps = new PreparedStatement("select * from user where user_name like 'a%'", 20, 5);
```

4.1.4 How it All Works Together

Once a PreparedStatement has had its parameters filled out, it's ready to be used for a query or an update (depending on which type of SQL statement it contains).

Here is an example of how to list all users:

```
$db = ConnectionFactory::getConnection();
$ps = new PreparedStatement('select * from user where id >= ? and id <= ? order by user_name');
$ps->setInt(3);
$ps->setInt(12);
$rs = $db->executeQuery($ps);
while ($row = $db->fetchObject($rs)) {
    print_r($row);
}
$db->freeResult($rs);
$db->close();
```

The executeQuery() function returns a result set identifier, which can be used to retrieve the rows returned by the query. If your SQL statement is an update-only statement which returns no rows, use the executeUpdate() function.

This version is shorter, but uses more memory:

```
$db = ConnectionFactory::getConnection();
$ps = new PreparedStatement('select * from user where id >= ? and id <= ? order by user_name');
$ps->setInt(3);
$ps->setInt(12);
print_r($db->fetchAllObjects($db->executeQuery($ps), true));
$db->close();
```

Rows can be fetched as objects [fetchObject(), fetchAllObjects()] or associative arrays [fetchArray(), fetchAllArrays()]. When you're done with a result set, always free it using $db->freeResult($rs). For fetchObject(), fetchAllObjects(), fetchArray() and fetchAllArrays(), you can free the result set before returning by adding passing the boolean

true as the second argument to the function call. This will cause the Connection to fetch the object(s) or array(s), free the result set and return what it fetched. Only do this if you know you're done with that result set.

By default, any query which fails will throw an Exception. This means that if you try to insert, update or delete a row and the operation violates a foreign key constraint, an Exception will be thrown. If you attempt to execute an SQL query which contains a syntax error, an Exception will be thrown. In some cases, when you know there are no syntax errors in your SQL (you've already tested your SQL statement), you may wish to prevent Exceptions from being thrown for a specific query. There are three variables which control this. They are documented in the the Connection abstract class. They are $throwExceptionOnFailedQuery, $showSQLInExceptions and $throwExceptionOnFailedFreeResult. If you need to disable one or more of these for a single query, it is recommended that you save their state, disable the ones you want disabled, execute the query, then restore their state. If a query fails and Exception throwing is disabled, then the executeQuery() or executeUpdate() function will return a boolean false value. Use ($returnValue === false) or ($returnValue !== false) to check the return value, since a return value of integer zero will also equal false if you use the (==) and (!=) operators.

The Connection class contains a couple of other functions which are useful. After successfully executing a statement which performs an update using executeUpdate(), you can determine how many rows were actually updated by calling the Connection's getUpdatedRowCount() function. Similarly, after successfully executing an SQL insert into a table with an auto-incrementing primary key by calling executeUpdate(), you can call the Connection's getLastInsertId() function to get the auto-incrementing primary key value of the row which was just inserted.

To begin a database transaction, call the Connection's beginTransaction() function. To commit the transaction, call commitTransaction(). To roll the transaction back, call rollbackTransaction(). Note that transactions can be nested. There is an internal nesting counter which begins at zero. It is incremented with each call to beginTransaction(). When it makes the transition from zero to one, the transaction is begun on the RDBMS. The counter is decremented by calling commitTransaction(). When it transitions from one to zero, the transaction is committed on the RDBMS. Calling rollbackTransaction() also decrements the counter, but sets a boolean which causes the Connection to roll the transaction back when the counter reaches zero instead of committing it.

The generated DAO classes use instances of the Connection and PreparedStatement classes for all of their database access operations. In fact, when you create a DAO object instance, you pass the Connection instance into the DAO class' constructor. The DAO instance then stores that Connection instance away so it can use it to access the database.

4.2 Database Schema

4.2.1 Introduction

In an RDBMS, the database schema is created and altered using Data Definition Language (DDL) SQL statements. In phpdaogen, the database schema can be created directly in the database using DDL SQL statements, or (preferably) expressed in one or more YAML or XML files. When the schema is expressed in YAML or XML files, phpdaogen can automatically generate SQL statements to create the corresponding tables, indexes and foreign keys for any supported RDBMS, or to bring an existing database's schema into alignment with the current schema expressed in the schema files. These schema files, along with scripts to initialize a database or update its schema, reside in the ddl subdirectory of the framework's root directory. The ddl subdirectory is a sibling to the phpdaogen directory (both appear under the same parent directory).

Schema files can exist in any subdirectory under the ddl subdirectory. For YAML schema files, the filename must end with '.ddl.yaml' or '.ddl.yml'. For XML schema files, the filename must end with '.ddl.xml'. If you're writing an accounting system, for example, you may wish to organize your application by module. In this case, you might have schema files like the following:

- ddl/base.ddl.yaml: contains the schema for the core tables which are not associated with any specific module
- ddl/modules/ap/ap.ddl.yaml: contains the schema for the Accounts Payable module's tables
- ddl/modules/bank/bank.ddl.yam: contains the schema for the Bank module's tables

- ddl/modules/gl/gl.ddl.yam: contains the schema for the General Ledger module's tables

And so on. This is just an example. You are free to organize the schema files any way you prefer, as long as they conform to the above-mentioned naming conventions and contain valid phpdaogen database schema definitions as described in this manual.

The preferred format for schema files is YAML, although XML format is also supported. Since XML tends to be more picky, verbose and difficult to read, the recommended format is YAML.

4.2.2 YAML DDL File Format

The structure of a YAML DDL file is as follows:

```
tables:
  (table name):
columns:
  (column name):
    type: (type)
    size: (size)
    scale: (scale)
    null: (No|Yes)
    default: (default value)
    sysVarDefault: (system variable to use for default value)
    autoIncrement: (No|Yes)
    useTimeZone: (Yes|No)
  (columnName):
    ...
  primaryKey:
    columns: [ (comma-separated list of primary key column names) ]
  indexes:
    (index name):
      fulltext (No|Yes)
        NOTE: Fulltext indexes currently only work with MySQL version 5.6 and later.
              Attempting to create a fulltext index on MySQL prior to version 5.6 will result in an S
              Fulltext indexes are ignored on non-MySQL RDBMS.
              For fulltext searching to work (without generating SQL errors), there must be a fulltex
              on ONLY the column being searched.  Therefore, it is recommended to create fulltext inc
              single columns only.  You can create multiple fulltext indexes per table when using MyS
      unique: (No|Yes)
        NOTE: The "unique" attribute is ignored, and should be omitted, for fulltext indexes.
      columns: [ (comma-separated list of index column names) ]
    (index name):
      ...
  foreignKeys:
    (foreign key name):
      foreignTable: (name of table referenced by the foreign key)
      columns:
        (unique name; unused):
          local: (column name in referring table)
          foreign: (referenced column name in referenced table)
        (unique name; unused):
          ...
    (foreign key name):
      ...
    inserts:
      -:
        keyColumnNames: Optional, comma-separated list of column names which uniquely identify
            this insert, so it can be determined at a later time whether this row already exists
```

```
            in the table.
        updateIfExists: No|Yes; whether to update the other fields in the existing row if a row
            already exists with the same combination of values in the columns listed in
            keyColumnNames; only applicable if keyColumnNames lists one or more of the columns
            in the insert.
        (column name):
          value: (column value)
          filename: (filename containing column value)
              This can be a relative path, relative to the directory where the YAML schema file
              exists, or it can be an absolute path which begins with a forward slash (/).
          sysVarValue: (system variable to use for value, such as CURRENT_TIMESTAMP)
              NOTE: Only one of [value, filename, sysVarValue] may be specified per column.
          quoted: No|Yes; whether to wrap the column value in SQL quotes and escape it
              accordingly.  Set this to Yes for character-based column types, dates, times, and
              so on.  Set it to No for numeric column types.
      -:
        ...
```

Items in parentheses get replaced with their actual values. If two or more options are listed, the first option is the default which is used if the parameter is omitted. Some attributes are optional, depending on column types, etc. Also note that although in the example above, attributes are listed one per line, it is possible (and preferable in most cases) to simply put them all on one line within a pair of curly braces, like this:

```
tables:
  appuser:
    columns:

      id: { type: integer, null: No, autoIncrement: Yes }

      when_added: { type: datetime, useTimeZone: Yes, null: No, sysVarDefault: CURRENT_TIMESTAMP }

      user_name: { type: varchar, size: 32, null: No, default: "" }

      email_addr: { type: varchar, size: 255, null: No, default: "" }

      password_hash: { type: varchar, size: 255, null: No, default: "" }

      first_name: { type: varchar, size: 30, null: No, default: "" }

      last_name: { type: varchar, size: 30, null: No, default: "" }

      is_active: { type: smallint, null: No, default: 1 }

      ...
```

In YAML, tabs should not be used for indentation; only spaces. The indentation depth controls the nesting of elements, so be careful to get your indentation correct.

Phpdaogen uses the Spyc YAML parsing library to parse YAML documents.

Column parameters:

- type: SQL data type. One of: integer, smallint, bigint, decimal, char, varchar, binary, varbinary, text, blob, date, time, datetime
- size: The size of the column. Only applicable for decimal, char, varchar, binary or varbinary type columns.
- scale: The numeric scale (number of fractional digits) for the column. Only applicable for decimal type columns.
- null: Yes to allow NULL values in the column; No to not allow. Optional. Defaults to No.

- default: The default value for the column. If both defaultValue and sysVarDefault, a default value will be selected based on the column type. The defaultValue and sysVarDefault parameters are mutually exclusive (you cannot specify both).
- sysVarDefault: When you need to use an RDBMS system variable as the default for a column, use this. Currently the only system variable that is supported is CURRENT_TIMESTAMP, and only for datetime type columns where useTimeZone is Yes.
- autoIncrement: For integer, smallint and bigint type columns, setting this to yes causes the column to automatically receive an auto-incrementing unique identifier for inserts which do not include a value for this column.
- useTimeZone: For datetime type columns, this can be set to Yes to enable storing the timezone along with the date/time (or storing the date/time in a neutral timezone such as GMT or UTC). For MySQL, this results in the use of the timestamp type rather than the datetime type. Keep in mind that MySQL only allows one column of type timestamp per table.

Inserts are used to pre-populate certain tables when they are first created. This is used to load up a database with initial (required) data, such as default permissions lists. Here is an example of how to define a table and populate it with some default rows:

```
tables:
  ...
  perm:
    columns:
      id: { type: integer, null: No, autoIncrement: Yes }
      when_added: { type: datetime, useTimeZone: Yes, null: No, sysVarDefault: CURRENT_TIMESTAMP }
      perm_name: { type: varchar, size: 40, null: No, default: "" }
      description: { type: varchar, size: 40, null: No, default: "" }
    primaryKey:
      columns: [ id ]
    indexes:
      permname:
        unique: Yes
        columns: [ perm_name ]
      descr:
        unique: Yes
        columns: [ description ]
    inserts:
      -:
        keyColumnNames: [ perm_name ]
        updateIfExists: Yes
        perm_name: { value: "all", quoted: Yes }
        description: { value: "All Permissions", quoted: Yes }
      -:
        keyColumnNames: [ perm_name ]
        updateIfExists: Yes
        perm_name: { value: "appuser", quoted: Yes }
        description: { value: "Administer Users", quoted: Yes }
      -:
        keyColumnNames: [ perm_name ]
        updateIfExists: Yes
        perm_name: { value: "approle", quoted: Yes }
        description: { value: "Administer Roles", quoted: Yes }
```

4.2.3 XML DDL File Format

The XML DDL file format is similar to the YAML DDL file format, except that it is expressed in XML, and instead of using camelCase tag names, it uses a slightly different set of hyphenated tag names. For exact details on the XML

DDL file format, have a look at ddl.dtd in the phpdaogen directory. This DTD file can be used to validate an XML DDL file.

4.3 Data Access Objects

4.3.1 Introduction

Data Access Objects (DAO) bridge the gap between a relational data model and an object-oriented programming language such as PHP.

Phpdaogen provides a code generator which generates classes for each table it finds in the schema. The two classes which will be used by application code are the data object class the data access object (DAO) class. By default, these classes are generated only once, and extend corresponding abstract classes which are re-generated each time the code generator is run. This allows the data object and DAO (concrete) classes to be extended with custom functionality, should you desire. If the code generator is run with the -noabstract command line option, then no abstract classes will be generated; all of the logic will be put straight into the concreate data object and DAO classes. You can use this option if you know that you will never want to customize the data object classes or DAO classes. When generating abstract classes (the default), all of the generated code goes into the abstract classes, leaving the concrete classes completely empty for ease of customization. The abstract data object class is a very simple class containing public variables for the columns in the table, one function to set all fields to default, and one function to load all fields from an associative array. The abstract data access object class includes functions to insert, update and delete rows in the table, a load() function to load a row by its primary key, and functions to find rows by various columns.

In all discussions in this guide, it is assumed that we are using the default mode of operation for the phpdaogen code generator, which generates abstract classes.

Consider the following table schema:

```
appuser:
  columns:
    id: { type: integer, null: No, autoIncrement: Yes }
    when_added: { type: datetime, useTimeZone: Yes, null: No, sysVarDefault: CURRENT_TIMESTAMP }
    user_name: { type: varchar, size: 32, null: No, default: "" }
    email_addr: { type: varchar, size: 255, null: No, default: "" }
    password_hash: { type: varchar, size: 255, null: No, default: "" }
    first_name: { type: varchar, size: 30, null: No, default: "" }
    last_name: { type: varchar, size: 30, null: No, default: "" }
    is_active: { type: smallint, null: No, default: 1 }
  primaryKey:
    columns: [ id ]
  indexes:
    username:
      unique: Yes
      columns: [ user_name ]
    emailaddr:
      unique: Yes
      columns: [ email_addr ]
    lastfirstname:
      unique: Yes
      columns: [ last_name, first_name ]
  foreignKeys: ~
```

This results in (the equivalent of) the following DDL SQL in the MySQL dialect:

```
CREATE TABLE user (
    id int(11) NOT NULL AUTO_INCREMENT,
```

```
    when_added timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    user_name varchar(32) NOT NULL DEFAULT '',
    email_addr varchar(255) NOT NULL DEFAULT '',
    password_hash varchar(255) NOT NULL DEFAULT '',
    first_name varchar(30) NOT NULL DEFAULT '',
    last_name varchar(30) NOT NULL DEFAULT '',
    is_active smallint(6) NOT NULL DEFAULT '1',
    PRIMARY KEY (id),
    KEY username (user_name),
    KEY emailaddr (email_addr),
    KEY lastfirstname (last_name, first_name)
) ENGINE=InnoDB;
```

Phpdaogen will then generate something like the following abstract data object class (pruned down for brevity):

```
html/classes/dao/abstract/AppuserAbstract.class.php:

abstract class AppuserAbstract {
    public $id;
    public $when_added;
    public $user_name;
    public $email_addr;
    public $password_hash;
    public $first_name;
    public $last_name;
    public $is_active;

    public static function createDefault() {
        $v = new Appuser();
        $v->defaultAllFields();
        return $v;
    }

    public function defaultAllFields() {
        ...
    }

    public function loadFromArray($arr) {
        ...
    }
}
```

This data object class is very straightforward, and requires no discussion. The concrete data object class looks something like this:

```
html/classes/dao/Appuser.class.php:

class Appuser extends AppuserAbstract {
}
```

Phpdaogen will also generate something like the following DAO class (also pruned down for brevity):

```
html/classes/dao/abstract/AppuserDAOAbstract.class.php:

abstract class AppuserDAOAbstract {
    public static $ALLOWED_QUERY_OPERATORS = ...
    public static $ALLOWED_NUMERIC_QUERY_OPERATORS = ...
    public static $ALLOWED_STRING_QUERY_OPERATORS = ...
    public static $ALLOWED_BINARY_QUERY_OPERATORS = ...
```

```
    protected $connection;
    protected $cache = null;

    public function __construct($connection, $cache = null) {
        $this->connection = $connection;
        $this->cache = $cache;
    }

    public function getCache() {
        return $this->cache;
    }

    public function setCache($cache) {
        $this->cache = $cache;
    }

    public function insert(&$user) {
        ...
    }

    public function update($user) {
        ...
    }

    public function delete($id) {
        ...
    }

    public function load($id) {
        ...
    }

    public function findByIdPS($id, $queryOperator = '=', $orderBy = null, $offset = 0, $limit = 0)
        ...
    }

    public function findById($id, $queryOperator = '=', $orderBy = null, $offset = 0, $limit = 0) {
        ...
    }

    public function findByWhen_addedPS($when_added, $queryOperator = '=', $orderBy = null, $offset =
        ...
    }

    public function findByWhen_added($when_added, $queryOperator = '=', $orderBy = null, $offset = 0,
        ...
    }

    public function findByUser_namePS($user_name, $queryOperator = '=', $orderBy = null, $offset = 0,
        ...

    }

    public function findByUser_name($user_name, $queryOperator = '=', $orderBy = null, $offset = 0, $
        ...
    }

    public function findByEmail_addrPS($email_addr, $queryOperator = '=', $orderBy = null, $offset =
```

```
        ...
    }

    public function findByEmail_addr($email_addr, $queryOperator = '=', $orderBy = null, $offset = 0,
        ...
    }

    public function findByPassword_hashPS($password_hash, $queryOperator = '=', $orderBy = null, $offs
        ...
    }

    public function findByPassword_hash($password_hash, $queryOperator = '=', $orderBy = null, $offse
        ...
    }

    public function findByFirst_namePS($first_name, $queryOperator = '=', $orderBy = null, $offset =
        ...
    }

    public function findByFirst_name($first_name, $queryOperator = '=', $orderBy = null, $offset = 0,
        ...
    }

    public function findByLast_namePS($last_name, $queryOperator = '=', $orderBy = null, $offset = 0,
        ...
    }

    public function findByLast_name($last_name, $queryOperator = '=', $orderBy = null, $offset = 0, $
        ...
    }

    public function findByIs_activePS($is_active, $queryOperator = '=', $orderBy = null, $offset = 0,
        ...
    }

    public function findByIs_active($is_active, $queryOperator = '=', $orderBy = null, $offset = 0, $
        ...
    }

    public function findAllPS($orderBy = null, $offset = 0, $limit = 0) {
        ...
    }

    public function findAll($orderBy = null, $offset = 0, $limit = 0) {
        ...
    }

    public function findWithPreparedStatement($ps) {
        ...
    }
}
```

This class is much more complex, and merits some discussion. First note that the constructor requires a Connection instance, which gives it something to use to talk to the database. Also note the optional $cache second argument to the constructor. You can pass it a subclass of DAOCache, and it will use that for caching. So if you do a lot of reads using a DAO class, you can use a local disk cache or a memcached cache to reduce database load and greatly improve performance, as long as you're not doing any type of read-modify-write operations on that DAO and don't mind reading possibly stale data. You can tune your cache expiration time to limit the maximum age of stale data,

mitigating the stale data problem somewhat while still maintaining good performance. Caching can work wonders for the speed and performance of high-volume web applications. The concrete DAO class looks something like this:

```
class AppuserDAO extends AppuserDAOAbstract {
}
```

4.3.2 The insert(), update(), delete() and load() Functions

insert()

The insert() function adds a new row to the table. Its only argument is a data object. To insert a new row, do something like this:

```
$db = ConnectionFactory::getConnection();
$appuserDAO = new AppuserDAO($db);
$row = Appuser::createDefault();
$row->user_name = 'johndoe';
$row->first_name = 'John';
$row->last_name = 'Doe';
(set additional fields here)
$appuserDAO->insert($row);
echo "Newly inserted row's id is {$row->id}\\n";
$db->close();
```

update()

The update() function saves an updated row back to the table (updates an existing row). Its only argument is a data object. To update an existing row, do something like this:

```
$db = ConnectionFactory::getConnection();
$appuserDAO = new AppuserDAO($db);
if (!($row = $appuserDAO->load(2))) {
    echo "user id 2 not found!\\n";
} else {
    $row->last_name = 'Smith';
    $appuserDAO->update($row);
}
$db->close();
```

delete()

The delete() function deletes an existing row from the table. Its only argument is the primary key value (in the case of a compound primary key, it will require one argument per primary key column). To delete an existing row, do something like this:

```
$db = ConnectionFactory::getConnection();
$appuserDAO = new AppuserDAO($db);
$appuserDAO->delete(2);    // Deletes any existing row with id 2.
$db->close();
```

load()

The load() retrieves an existing row by its primary key. Its only argument is the primary key value (in the case of a compound primary key, it will require one argument per primary key column). To load an existing row, do something like this:

```
$db = ConnectionFactory::getConnection();
$appuserDAO = new AppuserDAO($db);
if (!($row = $appuserDAO->load(2))) {
    echo "user id 2 not found!\\n";
} else {
    print_r($row);
}
$db->close();
```

4.3.3 The Finder Functions

For each column in the table, phpdaogen generates a pair of finder functions. For example, for the first_name column, phpdaogen generates the following two functions:

```
public function findByUser_namePS($user_name, $queryOperator = '=', $orderBy = null,
    $offset = 0, $limit = 0) {
    ...
}
public function findByUser_name($user_name, $queryOperator = '=', $orderBy = null, $offset = 0,
    $limit = 0) {
    return $this->findWithPreparedStatement($this->findByUser_namePS($user_name, $queryOperator,
        $orderBy, $offset, $limit));
}
```

The first function exists solely for convenience. Given search criteria, it builds and returns a PreparedStatement instance containing the parameterized SQL to find and retrieve the corresponding rows. The second function (which would be the more likely function to be called by an application) simply calls the first function to build the PreparedStatement, then calls the findWithPreparedStatement() function with the resulting PreparedStatement, returning the rows it found.

The $queryOperator argument, which defaults to '=', can be any of the allowed query operators for the column's data type. Take a peek inside any generated DAO class, and have a look at the $ALLOWED_QUERY_OPERATORS, $ALLOWED_NUMERIC_QUERY_OPERATORS, $ALLOWED_STRING_QUERY_OPERATORS and $ALLOWED_BINARY_QUERY_OPERATORS constants to see the list of operators which can be passed in for $queryOperator.

The $orderBy argument allows you to determine the sort order of the returned rows. It can be either null, or a comma-separated list of column names and "asc" or "desc" keywords. This string is used literally within the "order by" clause of the SQL query.

The $offset and $limit arguments are passed into the PreparedStatement constructor, and can be used for result set pagination. $offset is the number of initial rows to skip, and $limit is the maximum number of subsequent rows to return after skipping.

The finder functions whose names do NOT end with 'PS' return an array of data objects representing the matching rows which were found in the database table. If no rows were found or the $offset and $limit pagination parameters are outside the range of the result set, an empty array will be returned.

The findAllPS() and findAll() functions only accept the $orderBy, $offset and $limit arguments. They return all rows in the table, subject to pagination by way of $offset and $limit.

The findWithPreparedStatement() function accepts only a single argument: a PreparedStatement instance which contains an SQL statement to retrieve the desired rows from the table. The statement should begin with 'select * from tableName' or similar. Joins are fine. The important criteria are that all columns in the table must be returned, and that the main table which is being queried must be the table which corresponds to the DAO currently in use. This function is used by all other finder functions in the DAO. It returns an array of data objects, or an empty array if no rows were found.

All finder methods must be used with care, especially if there are millions of rows in the table, since they could take a long time to complete, resulting in a Denial of Service (DOS) attack vulnerability. Additionally, since all finder methods attempt to load their entire result set into an in-memory array, failing to limit the size of the result set with the $limit argument could result in using up all available memory leading to an out of memory error.

CHAPTER

FIVE

SEARCH AND CRUD GENERATORS

5.1 Introduction

In database-driven applications, one of the most important tasks is the creation, retrieval, update and deletion of rows in the database tables. This is called CRUD, which stands for Create/Retrieve/Update/Delete.

In some ways, Jax Framework ™ is a CRUD-centric framework. Manual development of CRUD pages is an inherently tedious process. Jax Framework ™ makes CRUD development go much more smoothly by employing code generators to generate CRUD pages from YAML configuration files. For further customization, the generated CRUD pages support user-defined callback functions called 'hooks' which exist in separate, user-created files which follow specific naming conventions. Customization consists of creating an include file and defining specific callback functions which will be called automatically. These callback functions can alter and/or supplement the behavior of the CRUD page in order to add custom functionality with minimal coding.

5.2 Generator Configuration YAML File Format

The generator configuration YAML files are located in the gencfg subdirectory off of the Jax Framework ™ root directory, and are named after their corresponding table names. For example, the generator configuration file for the user table would be named gencfg/user.yaml.

5.2.1 Top-Level Entries

A generator configuration (or gencfg) file has the following structure:

```
(global settings)
searches:
  Searches defined here will result in generated server-side search includes which can be used
  to allow the user search for rows.  Searches return JSON-encoded data, in a special format
  custom-made for consumption by the jax-grid component via the AJAXSearchGrid class.
  ...
autocompleteSearches:
  Searches defined here will result in generated server-side search includes which can be used
  to satisfy autocomplete requests.  Searches return JSON-encoded data, in a special format
  custom-made for consumption by the jQuery-UI autocomplete functionality.
  ...
popupSearches:
  Popup searches defined here will generate client-side JavaScript includes which will produce
  jax-grid-based inline popup searches which tie into the server-side search includes to fetch
  their data.  Popup searches are used for establishing foreign key relations by allowing the
  user to look up the appropriate row in the referenced table, and then importing the id into
  the form.
```

```
  ...
loaders:
  Loaders defined here will generate server-side single-row loader includes which can be access
  on the client in order to fetch a single row from a table.  These are handy for looking up a
  description from an entered Id, or for loading a row into a form.  Single-row loader includes
  may also join in information from other table(s) and put that data into attributes of the
  resulting row which is then returned to the client.  Loaders return the resulting data in
  JSON-encoded format.
  ...
cruds:
  CRUD pages defined here will combined searches, popup searches, loaders, form fields, filters,
  validators, etc. to create a complete working CRUD page.
  ...
```

The global settings which appear at the top of the file can contain any combination of the following:

- tableDescription: Singular human-readable description of the table entity.
- tableDescriptions: Plural human-readable description of the table entity.
- jaxInclude: Path to where the jax/include directory is, relative to document root; defaults to jax/include.
- jaxJQuery: Path to where the jax/jquery directory is, relative to document root; defaults to jax/jquery.
- jaxJS: Path to where the jax/js directory is, relative to document root; defaults to jax/js.
- loggedInId: If you've installed your own security system instead of using the user/perm/role-based example, you must set this to specify the PHP statement which retrieves the primary key id of the currently logged in user, in order for the permissions to work; defaults to $loggedInUser->id.

5.2.2 Searches

The searches section of the YAML file looks like this:

```
searches:
  (search name):
    searchCommand: (optional value of the 'command' request parameter to pass when making
        server-side search requests to satisfy this search; if omitted, this will default to
        "search<search name>s" with the first character of search name uppercased)
    outputPath: (output path for search includes, relative to the html directory; defaults to
        'include/search')
    docRootPath: (path back to the document root from the output path; defaults to an
        upward-pointing relative path based on outputPath)
    searchTemplate: (filename of search template to use; always in templates/search; defaults to
        search.include.php)
    phpClasses:
      \[NOTE: this section is optional, since the generated code uses a class autoloader.]
      (PHP class name): { path: (path to PHP class file) }
      (PHP class name): { path: (path to PHP class file) }
      ...
    phpIncludes:
      (unique include identifier; unused): (path to PHP include file)
      (unique include identifier; unused): (path to PHP include file)
      ...
    extraSelectColumns: [ (comma-separated list of column names from other table aliases) ]
      NOTE: This can be either in the form of an array or simply a string containing expressions
            to be appended to the column list in the select.
    joins: (sql join clauses; the main table is always aliased as 'pri')
    searchableColumns:
      -:
```

```
      columnName: (searchable column name, as it is known in the table in which it resides)
      tableAlias: (table alias for the table in which the searchable field exists; defaults to
          'pri')
      title: (title for this column; automatically generatd from column name if omitted)
      sqlType: (SQL type)
      queryOperator: (SQL query operator; defaults to '=')
      unsignedSearch: (No|Yes; only applies to numeric fields \[integer, smallint, bigint,
          decimal])
    -:
      ...
  andWhere: (sql "where" subclause which will be ANDed with the remainder of the "where"
      clause; the main table is always aliased as 'pri')
  andWhereAssignments:
    -:
      expression: (PHP expression of value to put into PreparedStatement placeholder)
      psType: (PreparedStatement data type) }
  groupBy: (optional sql "group by" subclause which will inserted immediately after the
      "where" clause)
  rowProcessingPHPCode: (optional PHP code which runs inside a loop, to pre-process every row.
      The $row variable will contain the current row to be processed.  You can use this
          function to calculate and set any calculated attributes in $row.)
  forbiddenColumns: [ (comma-separated list of columns NOT to be sent to the client) ]
(search name):
  ...
```

Within an entry under the searchableColumns subsection, the queryOperator parameter can be any of the following, depending on the sqlType for the searchable field:

Allowed Query Operators by SQL Column Type

sqlType	Allowed queryOperator Values
integer, smallint, bigint or decimal	=, <>, <, <=, >, >=
char, varchar or text	=, <>, <, <=, >, >=, beginsWith, contains, endsWith, fulltext

Other SQL types are not currently directly searchable by users without special coding.

NOTE: Fulltext search currently only work with MySQL version 5.6 and later. Attempting to create a fulltext index on MySQL prior to version 5.6 will result in an SQL error. Fulltext indexes are ignored on non-MySQL RDBMS. For fulltext searching to work (without generating SQL errors), there must be a fulltext index on ONLY the column being searched. Therefore, it is recommended to create fulltext indexes on single columns only. You can create multiple fulltext indexes per table when using MySQL.

Every entry in the searches section generates a resulting server-side search include which can then be included in CRUD pages or other pages, to enable the user to search the rows of the corresponding table to select rows for CRUD operations or for the purpose of establishing foreign key relations between rows in different tables. The name of the generated file is searchName_search.include.php, where the searchName portion is equal to the (search name) entry. Each file looks for a request parameter named 'command', which is equal to 'search(SearchName)s', where (SearchName) is the value of (search name) parameter for that search, with the first character uppercased. This way, multiple search includes can be included in the same page, and each can be triggered by passing different values into the 'command' request parameter.

It is possible to write searches by hand, using a generated search as an example. This is only recommended if there is no other way to accomplish the desired task. If you do this, you should put the file under include/customSearch to keep it separate from generated searches.

5.2.3 Autocomplete Searches

The autocompleteSearches section of the YAML file looks like this:

```
autocompleteSearches:
  (search name):
    searchCommand: Optional value of the 'command' request parameter to pass when making
        server-side search requests to satisfy this search; if omitted, this will default to
        "autocomplete<search name>s" with the first character of search name uppercased.
    outputPath: Output path for search includes, relative to the html directory; defaults to
        'include/search'.
    docRootPath: Path back to the document root from the output path; defaults to an
        upward-pointing relative path based on outputPath.
    searchTemplate: The filename of the search template to use; always in templates/search;
        defaults to search.include.php.
    phpClasses:
      \[NOTE: this section is optional, since the generated code uses a class autoloader.]
      (PHP class name): { path: (path to PHP class file) }
      (PHP class name): { path: (path to PHP class file) }
      ...
    phpIncludes:
      -: (path to PHP include file)
      -: (path to PHP include file)
      ...
    extraSelectColumns: [ (comma-separated list of column names from other table aliases) ]
      NOTE: This can be either in the form of an array or simply a string containing expressions
            to be appended to the column list in the select.
    joins: SQL join clauses; the main table is always aliased as 'pri'.
    searchableColumns:
      -:
        columnName: Searchable column name, as it is known in the table in which it resides.
        tableAlias: Table alias for the table in which the searchable field exists; defaults to
            'pri'.
        sqlType: SQL data type.
        queryOperator: SQL query operator; defaults to '='.
        unsignedSearch: No|Yes; only applies to numeric fields \[integer, smallint, bigint,
            decimal].
      -:
        ...
    andWhere: SQL "where" subclause which will be ANDed with the remainder of the "where"
        clause; the main table is always aliased as 'pri'.
    andWhereAssignments:
      -:
        expression: PHP expression of value to put into PreparedStatement placeholder.
        psType: PreparedStatement data type.
        searchResultLabelExpression: PHP expression to build search result label.  This must
            extract columns from the $row object and format them for display in the autocomplete
            list.  This expression will be executed once per row.
        searchResultValueExpression: PHP expresion to build search result value.  This must
            extract the identifier column from the $row object and prepare it for use in
            autocompletion.  The end result will be an identifier value to be plugged into the
            input field when the current row's label is selected from the autocomplete list.
            This entry is optional.  If omitted, the table's primary key column will be used.
  (search name):
    ...
```

The autocompleteSearches section works very similarly to the searches section (above). For more details, see that section.

5.2.4 Popup Searches

The popupSearches section of the YAML file looks like this:

```
popupSearches:
  (popup search name):
    outputPath: The output path for client-side JavaScript include file, relative to the html
        directory; defaults to 'js/search'.
    popupSearchTemplate: The filename of the pop-up search template to use; always in
        templates/search; defaults to popupSearch.js.
    searchPresentation: AJAXSearchGrid for AJAXSearchGrid presentation, or dataTables for (deprecated
        dataTables presentation.  If omitted, this defaults to dataTables for backward
        compatibility.  All new YAML files generated by makegencfg will have searchPresentation
        set to AJAXSearchGrid.
    defaultSorts: An optional list of objects, where each object must have an attribute named
        "attr" which is the attribute (column) name on which to sort, and and attribute name
        "dir" which is set to 1 for ascending sort or -1 for descending sort.  This attribute
        only applies to AJAXSearchGrid searchPresentation.
    searchCommand: The value of the 'command' request parameter to pass when making server-side
        search requests to satisfy this search; required for pop-up searches.
    idColumn: The name of the \[integer, smallint or bigint] primary key column which will be
        retrieved when a users selects a row from this popup search.
    beforeSearchCallback: An optional function to be called at the beginning of each search.
        The function receives the AJAXSearchGrid as its only argument.  This attribute only
        applies to AJAXSearchGrid searchPresentation.
    modifyURLCallback: An optional function to modify the server-side request URL before making
        the server-side request.  The function receives the AJAXSearchGrid and the URL as its
        arguments, and returns the modified URL.  The function will get called once for each
        server-side search request.  This attribute only applies to AJAXSearchGrid searchPresentatior
    afterSearchCallback: An optional function to be called at the end of each search.  The
        function receives the AJAXSearchGrid as its only argument.  This attribute only applies
        to AJAXSearchGrid searchPresentation.
    rowSelectJavaScriptCallbackFunction: The name of a JavaScript function which will be called
        when the user selects a row from this popup search; this function must have one argument
        which will be the id of the selected row.
    columns:
      (column name):
        displayType: For dataTables [DEPRECATED] searchPresentation, this is the jQuery DataTables
            column type; one of: string, numeric, date, html.  For AJAXSearchGrid searchPresentation,
            this is the display type; one of: string, html.  If this is set to html, then raw
            HTML can be injected into the data cells.  Be careful with this, as you could open
            yourself up for injection attacks.
        sortable: No|Yes; whether the user can sort the table on this column.
        heading: Column heading text.
        headerCSSClass: Any additional CSS classes you wish to apply to the header cell.
        columnCSSClass: Any additional CSS classes you wish to apply to the data cells.
        fnRender: [DEPRECATED]  Optional DataTables callback function for rendering the cell.
            This attribute only applies to the deprecated dataTables searchPresentation.  You
            can specify either the name of a JavaScript function which you've defined in another
            JavaScript include, or define an anonymous function right in the YAML file as long
            as you follow proper YAML syntax and remain mindful of the affect of indentation on
            the YAML parser.  For documentation on this function, look for fnRender on this
            page: `DataTables - Usage <http://www.datatables.net/usage/columns>`_.
        columnFilters: An optional list of column filters to apply to the column data as it is
            being displayed.  These are the names of the developer-defined column filters to
            apply.  This works in parallel with the columnFilters attribute at the pop-up search
            level, which is used to define column filter functions.  This attribute only applies
            to AJAXSearchGrid searchPresentation.
        headerTemplate: An optional AngularJS HTML template to use when rendering the header
```

```
            cells for this column.  If this is present, it overrides all other column settings
            when rendering the header cells for this column.  This attribute only applies to
            AJAXSearchGrid searchPresentation.
        columnTemplate: An optional AngularJS HTML template to use when rendering the data cells
            for this column.  If this is present, it overrides all other column settings when
            rendering the data cells for this column.  This attribute only applies to AJAXSearchGrid
            searchPresentation.
      (column name):
        ...
    invisibleColumns: An optional list of additional column names to be requested from the
        server.  This allows the popupSearch to receive columns which are not listed in the
        columns list, such as columns on which calculations are based but which are not
        displayed directly.  This attribute only applies to AJAXSearchGrid searchPresentation.
    columnFilters: An optional list which maps developer-defined column filter names to column
        filter functions.   Each function takes the following arguments, from which it
        calculates and returns the filtered row data to be displayed:
            val: The unfiltered column value, with any previous filter functions applied.
            row: The row object.
            rowIdx: The index of the row, relative to the current page of results.
        Column filters defined here can be referenced inside individual columns and even chained
        together within the same column.  This attribute only applies to AJAXSearchGrid
        searchPresentation.
    extraQueryParams: An optional object which maps parameter names to parameter values for the
        query string in the AJAX URL which is used to fetch search results from the server.
        These extra parameters will be appended to the AJAX request URL before each search.
        Parameter names and values must NOT be URL-encoded in this map; encodeURIComponent()
        will be called on them automatically as they are appended to the AJAX request URL.
        If any of the keys match existing query parameters on the search URL, their existing
        values will be replaced with the values from this map.  This attribute only applies to
        AJAXSearchGrid searchPresentation.
    fnDrawCallback: [DEPRECATED]  Optional DataTables callback function which will be called
        immediately after each time the table is drawn.  This attribute only applies to the
        deprecated dataTables searchPresentation.  You can specify either the name of a
        JavaScript function which you've defined in another JavaScript include, or define an
        anonymous function right in the YAML file as long as you follow proper YAML syntax and
        remain mindful of the affect of indentation on the YAML parser. For documentation on
        this function, look for fnDrawCallback on this page: `DataTables - Usage
        <http://www.datatables.net/usage/callbacks>`_.
    fnServerData: [DEPRECATED]  Optional DataTables callback function which will be called
        immediately before making a search request to the server.  This allows you to alter the
        search URL, possibly adding/changing request parameters based on current state.  This
        attribute only applies to the deprecated dataTables searchPresentation.  You can specify
        either the name of a JavaScript function which you've defined in another JavaScript
        include, or define an anonymous function right in the YAML file as long as you follow
        proper YAML syntax and remain mindful of the affect of indentation on the YAML parser.
        For documentation on this function, look for fnServerData on this page: `DataTables -
        Usage <http://www.datatables.net/usage/callbacks>`_.
  (popup search name):
    ...
```

5.2.5 Loaders

The loaders section of the YAML file looks like this:

```
loaders:
  (loader name):
    searchCommand: Optional value of the 'command' request parameter to pass when making
```

```
    server-side search requests to satisfy this search; if omitted, this will default to
    "load<loader name>" with the first character of loader name uppercased.
outputPath: Output path for loader includes, relative to the html directory; defaults to
    'include/search'.
docRootPath: Path back to the document root from the output path; defaults to an
    upward-pointing relative path based on outputPath.
idColumn: The column name of the unique identifying column.
idColumnPSType: The PreparedStatement type of the id column; one of: boolean, int, float,
    double, string, match, or binary; defaults to int.
loaderTemplate: Filename of loder template to use; always in templates/search; defaults to
    load.include.php.
phpClasses:
  \[NOTE: this section is optional, since the generated code uses a class autoloader.]
  (PHP class name): { path: (path to PHP class file) }
  (PHP class name): { path: (path to PHP class file) }
  ...
phpIncludes:
  -: (path to PHP include file)
  -: (path to PHP include file)
  ...
andWhere: SQL "where" subclause which will be ANDed with the remainder of the "where"
    clause; the main table is always aliased as 'pri'.
andWhereAssignments:
  -:
    expression: PHP expression of value to put into PreparedStatement placeholder.
    psType: PreparedStatement data type. }
relations:
  (object attribute name to hold related data):
    table: The table name of the referenced table.
    useDAO: Yes to load related rows using the related table's DAO; No to load them as plain
        old PHP objects.  This is especially useful in combination with sqlQuery, for
        returning the results of an arbitrary join select.
    relationType: many|one; determines whether an array of rows will be fetched, or a single
        row.
    offset: The number of initial rows to skip.
    limit: The maximum number of rows to return; hint: if relationType is 'one', set this to
        1 for more efficient queries.
    sqlQuery: Optional direct SQL query to use for fetching the rows.
    sqlQueryAssignments:
      -:
        expression: PHP expression for the value to be assigned to the next ? placeholder in
            sqlQuery.
        psType: PreparedStatement data type for the expression; one of: boolean, int, float,
            double, string, match, binary; defaults to string; \[NOTE: match is the string
            with leading and trailing % characters for doing like queries)].
      -:
        ...
    queryOperator: One of the allowed query operators recognized by the DAO for the
        referenced column in the referenced table; defaults to '='.
    local: Column name of the referring column in the main/referring table.
    foreign: Column name of the referenced column in the referenced table.
    orderBy: SQL 'order by' clause for the query; must be valid for querying the referenced
        table.
  (object attribute name to hold related data):
    ...
forbiddenColumns: [ (comma-separated list of columns NOT to be sent to the client) ]
```

For a relation, if sqlQuery is specified, then sqlQueryAssignments is required and queryOperator, local, for-

eign and orderBy are ignored. This means that you can use only one or the other mechanism: either the sqlQuery/sqlQueryAssignments mechanism, or the local/foreign/orderBy mechanism.

Every entry in the loaders section generates a resulting server-side loader include which can then be included in CRUD pages or other pages, to enable the user to load individual rows from the corresponding table. The name of the generated file is loaderName_load.include.php, where the loaderName portion is equal to the (loader name) entry. Each file looks for a request parameter named 'command', which is equal to 'load(LoaderName)', where (LoaderName) is the value of (loader name) parameter for that loader, with the first character uppercased. This way, multiple loader includes can be included in the same page, and each can be triggered by passing different values into the 'command' request parameter.

It is possible to write loaders by hand, using a generated loader as an example. This is only recommended if there is no other way to accomplish the desired task. If you do this, you should put the file under include/customSearch to keep it separate from generated searches and loaders.

5.2.6 CRUDs

The cruds section of the YAML file looks like this:

```
cruds:
  (CRUD page name, excluding .php extension):
    outputPath: Output path for CRUD page and includes, relative to the html directory; defaults
        to empty.
    docRootPath: Path back to the document root from the output path; defaults to an
        upward-pointing relative path based on outputPath.
    phpClasses:
      \[NOTE: this section is optional, since the generated code uses a class autoloader.]
      (class name): { path: (path to PHP class file, relative to the directory where the CRUD
          page will go ) }
      ...
    postInitPHPIncludes:
      (path to PHP include file, relative to the directory where the CRUD page will go )
      ...
    javaScriptFiles: [ (comma-separated list of JavaScript files which will be included in the
        page header, or one JavaScript file per line, beginning on the next line, indented two
        spaces in from this line) ]
    cssFiles: [ (comma-separated list of CSS files which will be included in the page header, or
        one CSS file per line, beginning on the next line, indented two spaces in from this
        line) ]
    allowAddSimilar: Yes to create an "Add Similar" link for each row in the search table,
        allowing the user to easily "clone" rows; No to omit this feature.  Optional.  Defaults
        to No.
    crudSearch:
      likePopupSearch: Name of popupSearch entry to use as a prototype.  This entry is optional.
          If it exists, A popupSearch entry of the specified name must exist in the same YAML
          file; it will be used as a starting point.  Any attributes which are defined in the
          crudSearch entry will override attributes of the same names in the popupSearch, but
          only for the purpose of generating the crudSearch.  This makes it easy to define a
          popupSearch entry for a table, then use it either as-is or customize it for use as the
          CRUD search for the same table.

      \[NOTE: The remaining parameters in this section are identical to the popupSearch section.
          Please see that section for details.]
    crudLoad:
      loadCommand: Loader command which will be used to load an existing row for populating the
          entry form; defaults to loadTableName, where TableName is the name of the main table
          with the first character uppercased.
```

```
formFields:
  id:
    title: The title, or prompt, for the form field
    placeholder: The placeholder text for text, password and textarea input types
    inputType: The HTML input type; one of: hidden, text, password, textarea, select,
        file, checkbox, radio, htmlFragment, tabs, tab, tabsClose, accordion,
        accordionGroup, accordionClose
    onclick: optional JavaScript code to execute when the field is clicked with the mouse;
        only applicable to certain input types; see
        `here <http://www.w3schools.com/jsref/event_onclick.asp>`_ for details.  Use of
        this attribute is not the preferred method.  When possible, please use the jQuery
        click() function to hook event handlers to input components.  You can hook up
        event handlers in the preInitHook() JavaScript callback.
    onchange: optional JavaScript code to execute when the field's value changes; only
        applicable to certain input types; see `here
        <http://www.w3schools.com/jsref/event_onchange.asp>`_ for details.  Use of this
        attribute is not the preferred method.  When possible, please use the jQuery
        change() function to hook event handlers to input components.  You can hook up
        event handlers in the preInitHook() JavaScript callback.
    ajaxAutocompleteCommand: command parameter for AJAX command to handle autocomplete
        requests for this field.  If specified, must match the command for an included
        autocomplete search include file.  This is used for simple autocomplete inside a
        text field.
    ajaxAutocompleteMinLength: minimum number of characters which must be typed into the
        field before autocomplete turns on.  This is used for simple autocomplete inside a
        text field.
    autocompleteSingleRowSelector:
      (option name): (option value)
        \[NOTE: The presence of this (sub)section causes a normal input component
         (typically inputType text) to behave as a searchable, drop-down list with
         server-side autocomplete functionality, for the puprose of selecting a row from a
         related table and submitting its id or primary key to the server when the form is
         submitted.  This functionality is implemented using a combination of the included
         ajaxComboBox component and a server-side autocomplete include (generated by the
         autocompletes section in the YAML file).  The options here are passed into the
         hookAutocompleteSingleRowSelectorToInput(options) function in
         html/jax/js/specialFieldFeatures.js.  Refer to that file for more details.
         Recognized options for this section are listed immediately below.]
      autocompleteCommand: The autocomplete command to use for searching rows on the
          server. Required.
      idColumn: The name of the primary key/unique identifying column for the table being
          searched. Optional.  Defaults to 'id'.
      idIsString: Yes if the primary key/unique identifying column is a string (character)
          column, No if it is an integer column. Optional. Defaults to No.
      minimumInputLength: The minimum number of characters the user must enter in order to
          be able to search rows in the table. Optional.
      allowClear: Yes to display a clear button when the select box has a selection. The
          button, when clicked, resets the value of the select box back to the
          placeholder, thus this option is only available when selectPlaceholder is
          specified as a non-empty placeholder (or left at its default, non-empty
          placeholder string).  Optional.  Defaults to No.
      selectPlaceholder: The placeholder text to be put into the select box if the id is
          zero (integer id) or empty (string id).  Optional.
      notFoundMessage: The text to be put into the select box if the currently selected id
          is invalid, but not zero (integer id) or empty (string id).  Optional.
      maxRowsPerPage: The maximum number of rows to retrieve at a time from the server
          (rows per page).  Optional.
    onPopupSearch: optional JavaScript to be executed when the popup search icon is
```

```
            clicked; this should show/hide an inline popup search
        descriptionField: optional name/id for a description text input which will be readonly
            and disabled, and will exist only to display a description for the identifying
            value of a related row whcih is entered into the main input component or selected
            using a popup search
        descriptionFieldSize: optional size for the description field; defaults to 40
        descriptionFieldMaxLength: optional maxlength for the description field; zero means no
            maxlength; defaults to zero size: size attribute for the input component;
            applicable only to text, password and select input types
        maxlength: maximum input length for the input component; applicable only to text and
            password input types
        cssClass: any additional CSS classes you wish to apply to the input element
        readonly: (No|Yes); whether to set the input component to readonly; applicable only to
            text, password, textarea and file input types
        disabled: No|Yes; whether to set the input component to disabled
        rows: number of visible rows of text for textarea input types
        cols: number of visible columns of text for textarea input types
        accept: comma-separated list of MIME types to accept for file input types (see the
            accept attribute of the HTML <file> tag for details)
        multiple: No|Yes; whether to allow multiple items to be selected for select input
            types
        options:
          (option value): { title: (option text) }
          ...
        optionsFromAssociativeArray: for select input types, this can be a PHP expression
            which returns a PHP associative array where the array keys are the option values
            and the array values are the option texts to be populated into the select
            component whenenever the view is output; if this parameter is set, the options
            parameter is ignored
        value: for checkbox input types, this is the value to return when the checkbox is
            checked.  Optional.  Defaults to 1.
        html: for htmlfragment input types, this is HTML code to be inserted at this point in
            the form.  For normal layout mode (div+CSS), this should be a set of nested <div>
            elements in the form of <div class="form-row"><div class="form-cell">(label
            content goes here)</div><div class="form-cell">(input content goes here)</div>.
            When using tables to layout the forms (not recommended), the outer tag in this
            content must be a table row (<tr>) element which fits into the two-column table
            which is used to layout the form.  The first column is generally used for field
            labels, and the second column is generally used for input elements.  However, you
            can use this to put any arbitrary HTML content into the form, as long as you wrap
            it inside the appropriate tags.  However, when using div+CSS, the content must
            always follow the two-column model UNLESS the inLayout attribute is set to No.
            The content can even include PHP code blocks by wrapping them in the standard
            <?php ... ?%gt; tags.

    filters:
      (unique column name to be filtered):
        (unique filter identifier within column; unused):
          class: Filter class name, e.g.: TrimFilter.
          include: Relative path to filter class, e.g.:
              jax/classes/crud/filter/TrimFilter.class.php.
            \[NOTE: the include entry is optional, since the generated code uses a class
                autoloader.]
          params: { (filter-specific parameters) }
        ...
      ...

    validators:
```

```
  (unique column name to be validated):
    (unique validator identifier within column; unused):
      class: Validator class name, e.g.: NotEmptyValidator.
      include: Relative path to validator class, e.g.:
          jax/classes/crud/validator/NotEmptyValidator.class.php.
        \[NOTE: the include entry is optional, since the generated code uses a class
            autoloader.]
      phpCondition: An optional PHP expression which evaluates to true if we need to run
          this validator, or false if not; this allows you to only run the validator when
          a certain set of conditions are met.
      params: { (validator-specific parameters) }
      ...
    ...

addFocusField: Optional field name to receive the input focus when going into add mode.
editFocusField: Optional field name to receive the input focus when going into edit mode.
onlyUpdateColumns: [ (comma-separated list of columns in the main table which are allowed
    to be updated when saving an updated row in edit mode; if this is set, only columns in
    this list will be updated) ]
neverUpdateColumns: [ (comma-separated list of columns in the main table which should
    never be updated when saving an updated row in edit mode) ]
```

Tabs and Accordions

The CRUD generator supports tabbed panes and accordions, including nested tabbed panes and accordions (tabbed panes or accordions inside of tabs or accordion groups), through the following inputTypes attributes for elements under the formFields subsection: tabs, tab, tabsClose, accordion, accordionGroup, accordionClose.

Tabs

The tabs input type begins a tabbed pane. A tabbed pane must contain one or more tab input types, followed by a tabsClose input type to close the tabbed pane. Each tab input type can be followed by one or more other inputs of any type, so that the form can be divided into logical sections which are grouped into tabs.

The tabs input type honors an optional "tabsPosition" parameter, which can be one of: top, left, right, or bottom. This controls the placement of the tabs in relation to their contained content. If the tabsPosition parameter is omitted, the tabs will be placed on the top (above their contained content).

The tab input type begins a tab within the most recently opened (and not-yet-closed) tabbed pane. A tabbed pane must contain one or more tab input types, followed by a tabsClose input type to close the tabbed pane. A tab can contain one or more tabbed panes or accordions (nesting). A tab contains all subsequent input elements until the next tab or tabsClose. A tab can have a title attribute, which is the text which will be displayed in the tab itself.

The tabsClose input type closes the most recently opened tabbed pane. For each tabs input type, there must be a corresponding tabsClose input type. Tabbed panes can be nested by placing a tabs ... tab ... [tab ...] tabsClose sequence of input elements inside a tab input type.

Accordions

The accordion input type begins an accordion. An accordion must contain one or more accordionGroup input types, followed by an accordionClose input type to close the accordion. Each accordionGroup input type can be followed by one or more other inputs of any type, so that the form can be divided into logical sections which are grouped into accordion groups.

The accordionGroup input type begins a group within the most recently opened (and not-yet-closed) accordion. An accordion must contain one or more accordionGroup input types, followed by an accordionClose input type to close the accordion. An accordionGroup can contain one or more tabbed panes or accordions (nesting). An accordionGroup contains all subsequent input elements until the next n accordionGropu or accordionClose. An accordionGroup can have a title attribute, which is the text which will be displayed in the heading of the group. When the heading of an accordion group is clicked, that group is expanded or collapsed.

The accordionClose input type closes the most recently opened accordion. For each accordion input type, there must be a corresponding accordionClose input type. Accordions can be nested by placing an accordionOpen ... accordion ... [accordion ...] accordionClose sequence of input elements inside an accordionGroup input type.

5.3 Default Generator Configuration File

In order to get a search, loader and CRUD page up and running quickly for a new table, Jax Framework ™ includes a command which will look at a table's schema and generate a starting-point YAML file which will create a search, a loader and a CRUD page for the table.

To generate the initial generator configuration YAML file, run the following command:

```
gen/makegencfg <tableName>
```

Where <tableName> is the name of the table. If a generator configuration YAML file already exists for the table you specify, the command will report that fact, and will exit without overwriting the existing file.

To get help with command line options for the generator config YAML file maker utility, run this:

```
gen/makegencfg -help
```

After the initial generator configuration YAML file has been created for a table, you are free to customize the YAML file to make everything work the way you want it to work. Remember to re-run the search and CRUD generators after you edit a generator configuration YAML file.

5.4 Search Generator

The search generator is a command-line PHP application. To run it, do something like this (from the Jax Framework ™ root directory):

```
gen/searchgen <tableName>
```

Where <tableName> is the name of the table for which you want to generate searches, loaders and popup searches. If you omit the table name, the search generator will (re-)generate searches, loaders and popup searches for all tables.

To get help with command line options for the search generator, run this:

```
gen/searchgen -help
```

By default, the search generator gets its schema information from the schema files in the ddl directory. You can make it retrieve its schema directly from the database by using the -ddlfromdb, -dbclass, -dbhost, -dbuser, -dbpassword and -dbdatabase command-line options. See the command-line help for more information.

If you want the document root for the output files to be something other than the html directory off of the Jax Framework ™ root directory, you can use the -docroot command-line option to change the document root for the search generator, causing it to output its files into a different directory.

5.5 CRUD Generator

The CRUD generator is a command-line PHP application. To run it, do something like this (from the Jax Framework ™ root directory):

```
gen/crudgen <tableName>
```

Where <tableName> is the name of the table for which you want to generate CRUD pages. If you omit the table name, the CRUD generator will (re-)generate CRUD pages for all tables.

To get help with command line options for the CRUD generator, run this:

```
gen/crudgen -help
```

By default, the CRUD generator gets its schema information from the schema files in the ddl directory. You can make it retrieve its schema directly from the database by using the -ddlfromdb, -dbclass, -dbhost, -dbuser, -dbpassword and -dbdatabase command-line options. See the command-line help for more information.

If you want the document root for the output files to be something other than the html directory off of the Jax Framework ™ root directory, you can use the -docroot command-line option to change the document root for the CRUD generator, causing it to output its files into a different directory.

5.6 CRUD Callbacks

5.6.1 Introduction

Once generated, the CRUD pages can be further customized by either tweaking the YAML file and re-generating, or creating special PHP (server-side model and view) and/or JavaScript (client-side controller) files which follow a specific naming convention, and defining certain callback functions within those files. These are called "hooks", and they allow the developer to add custom functionality and/or modify the default behavior of the generated CRUD pages. By using this approach, we can get CRUD pages working quickly by simply creating a YAML file and generating the searches, loaders, popup searches and CRUD pages. The default behavior is the most commonly used behavior. When the developer wants to extend or modify the default behavior, he/she simply creates the appropriate include file(s) and callback functions within them, along with any custom functions, to perform the customization. By using this approach, we avoid the need to edit any generated files.

5.6.2 Model Callbacks

The main PHP file for a CRUD page, which handles all database access, is considered part of the model in the MVC architecture as implemented in Jax Framework ™. This is because the JavaScript controller, which runs in the browser, is orchestrating everything once the page is initially loaded.

For a given CRUD page, <crudName>.php, there can (optionally) exist a file named <crudName>_hooks.include.php which contains callback functions which will be called automatically when certain events occur. Each callback function is optional; if it does not exist, it will not be called.

The following hook functions can be defined inside the <crudName>_hooks.include.php file. The comments above them describe how they work. Note that some functions declare specific global variables. These global variables are (some of) the global variables which would be relevant at the time the function is called. There may be others, and you are free to define and use your own global variables. It is even possible to define a global variable which is initialized by the preUpdateHook() function, and read by the postUpdateHook() function, for example.

```
// Put code here to perform any custom initialization.
function initHook() {
    global $params, $command;
    ...
}

// Add custom pre-processing on $row.
// At this point, $row is an instance of stdClass, cast from the $_POST array.
function preFilterHook() {
    global $db, $row, $result;
    ...
}

// Add custom validation here.
// At this point, $row is an instance of stdClass, cast from the $_POST array.
// Any errors which result from validation must be put into the $result error message container.
function validationHook() {
    global $db, $row, $result;
    ...
}

// Add any special processing here which must occur before the columns are copied from the new
// row ($row) to the old row ($oldRow) and the old row is updated in the table.
// At this point, $row is an instance of stdClass, cast from the $_POST array, and $oldRow is
// the original data object of the correct class, retrieved from the table.
// If there are specific columns which should not be copied from $row to $oldRow before $oldRow
// is saved back to the table, add those column names to the $neverUpdateColumns array.
// Generally, these would be configured in the CRUD generator YAML configuration file, but
// sometimes it is conditional whether to allow a column to be updated.
function preUpdateHook() {
    global $db, $row, $oldRow, $neverUpdateColumns;
    ...
}

// Add any special processing here which must occur after the old row ($oldRow) has been updated
// in the table, but before the transaction is committed.
// At this point, $row is an instance of stdClass, cast from the $_POST array, and $oldRow is
// the original data object of the correct class, retrieved from the table and specific columns
// updated from $row.
// Setting $success to false (it will be true when this function is called) will cause the
// transaction to be rolled back.
function postUpdateHook() {
    global $db, $row, $oldRow, $success;
    ...
}

// Add any special processing here which must occur before the new row ($newRow) is inserted.
// At this point, $row is an instance of stdClass, cast from the $_POST array, and $newRow is
// a data object of the correct class, created from the values in $row.
function preInsertHook() {
    global $db, $row, $newRow;
    ...
}

// Add any special processing here which must occur after the old row has been inserted
// in the table, but before the transaction is committed.
// At this point, $row is an instance of stdClass, cast from the $_POST array, and $newRow is
// a data object of the correct class, created from the values in $row.
```

```
// Setting $success to false (it will be true when this function is called) will cause the
// transaction to be rolled back.
function postInsertHook() {
    global $db, $row, $newRow, $success;
    ...
}

// Add delete check validation here.  The id which the user is attempting to delete is in $id.
// Any errors which would prevent deletion must be put into the $result error message container.
function deleteCheckHook() {
    global $db, $id, $result;
    ...
}

// Add any special processing here which must occur before the row (identified by $id) is deleted.
function preDeleteHook() {
    global $db, $id;
    ...
}

// Add any special processing here which must occur after the row (identified by $id) is deleted
// but before the transaction is committed.
function postDeleteHook() {
    global $db, $id;
    ...
}

// Put code here to execute just before the view is output.
// This may include processing of unhandled commands in $command.  To handle a command which
// requires a POST, first confirm that ($_SERVER['REQUEST_METHOD'] == 'POST').
function preViewOutputHook() {
    ...
}
```

5.6.3 View Callbacks

For a given view include page, <crudName>_view.include.php, there can (optionally) exist a file named <crudName>_view_hooks.include.php which contains callback functions which will be called automatically between specific blocks of HTML within the view. Each callback function is optional; if it does not exist, it will not be called.

One common use of view callbacks is to output HTML and/or JavaScript blocks which are custom to the page and are needed in order to extend the default CRUD functionality. One example would be outputting custom blocks of HTML in order to add functionality to the input form. Another example would be a script block which outputs JavaScript variables which contain arrays of data or constants which come from server-side PHP code, encoded using json_encode().

The following hook functions can be defined inside the <crudName>_view_hooks.include.php file. The comments above them describe how they work.

```
// Put code here to output any custom content after the page header template has been output.
function afterHeaderViewHook() {
    ...
}

// Put code here to output any custom content before the heading on the search page.
function searchBlock1ViewHook() {
    ...
```

```
}

// Put code here to output any custom content before the add link on the search page.
function searchBlock2ViewHook() {
    ...
}

// Put code here to output any custom content before the data table on the search page.
function searchBlock3ViewHook() {
    ...
}

// Put code here to output any custom content afer the data tabe on the search page.
function searchBlock4ViewHook() {
    ...
}

// Put code here to output any custom content before the page footer template is output.
function beforeFooterViewHook() {
    ...
}
```

5.6.4 Controller Callbacks

For a given controller JavaScript file, <crudName>_controller.js, there can (optionally) exist a file named <crudName>_controller_hooks.js which contains callback functions which will be called automatically when specific events occur within the controller. These are client-side functions, and are written in JavaScript. Each callback function is optional; if it does not exist, it will not be called.

The following hook functions can be defined inside the <crudName>_controller_hooks.js file. The comments above them describe how they work.

```
// This function is called after the page is loaded but before everything else is initialized.
// You can put custom initialization code here.
function preInitHook() {
    ...
}

// This function is called after the page is loaded and everything else is initialized.
// You can put custom initialization code here.
function postInitHook() {
    ...
}

// This function is called at the beginning of the attachSpecialFieldFeatures() function, which
// attaches special features to input elements with specific CSS classes.  For example, input
// elements with date and datetime classes get a date picker icon, and also get their change
// event hooked to filter and re-format their values.  Input elements with numeric-scale0
// through numeric-scale10 classes get their change event hooked to filter and re-format their
// numeric values.
function preAttachSpecialFieldFeatures() {
    ...
}

// This function is called at the end of the attachSpecialFieldFeatures() function.  See the
// preAttachSpecialFieldFeatures() function (above) for more details.
function postAttachSpecialFieldFeatures() {
```

```
    ...
}

// This function is called once per row in the search data table.
// You can customize this function to return HTML for any links you want to appear on the current
// row BEFORE the default Edit, Delete and View links.  These links can trigger custom actions
// for this row.
function getAdditionalRowSearchActionLinksPre(id, rowData) {
    ...
    return '';
}

// This function is called once per row in the search data table.
// You can customize this function to return HTML for any links you want to appear on the current
// row AFTER the default Edit, Delete and View links.  These links can trigger custom actions
// for this row.
function getAdditionalRowSearchActionLinksPost(id, rowData) {
    ...
    return '';
}

// This function is called once per row in the search data table.
// You can customize this function to enable/disable editing of specific rows on a per-row basis,
// or globally.  This function returns true if a specific row may be edited; false if not.
function canEditRow(id, rowData) {
    ...
    return true;
}

// This function is called once per row in the search data table.
// You can customize this function to enable/disable deleting of specific rows on a per-row basis,
// or globally.  This function returns true if a specific row may be deleted; false if not.
function canDeleteRow(id, rowData) {
    ...
    return true;
}

// This function is called once per row in the search data table.
// You can customize this function to enable/disable viewing of specific rows on a per-row basis,
// or globally.  This function returns true if a specific row may be viewed; false if not.
function canViewRow(id, rowData) {
    ...
    return true;
}

// This function is called by the setMode() function before it does any of its own work.
// You can put custom mode set code here.
function preSetModeHook(newMode) {
    ...
}

// This function is called by the setMode() function after it has done all of its own work.
// You can put custom mode set code here.
function postSetModeHook(newMode) {
    ...
}

// This function is called by the load*IntoForm() function before it does any of its own work.
// You can put custom form loading code here.
```

```
function preLoadFormHook(id, newMode, allowEditing) {
    ...
}

// This function is called by the load*IntoForm() function after it has retrieved the row from
// the server, but before it has loaded the data into the form.
// You can put custom form loading code here.  You can also add/change/delete attributes of
// row to feed values into specific form fields.
function midLoadFormHook(id, newMode, allowEditing, row) {
    ...
}

// This function is called by the load*IntoForm() function after it has retrieved the row from
// the server and loaded the data into the form.
// You can put custom form loading code here.  You can also add/change/delete attributes of
// row to feed values into specific form fields.
function mid2LoadFormHook(id, newMode, allowEditing, row) {
    ...
}

// This function is called by the load*IntoForm() function after it has done all of its own work.
// You can put custom form loading code here.
function postLoadFormHook(id, newMode, allowEditing, row) {
    ...
}

// This function is called by the save*() function before it has done any of its own work.
// If this function returns boolean false, the save will not be executed.
// It is NOT required that this function return anything, as long as it is okay to proceed
// with the save operation.
// The mode global variable determines which type of save operation is occurring.
function preSaveHook() {
    ...
}

// This function is called immediately after a row has been inserted, updated or deleted.
// The mode global variable determines which type of save operation has just occurred.
// If this function returns boolean false, the current mode will not be changed back to
// SEARCH_MODE following the return of this function.  This allows implementation of
// custom features such as "Save and Continue Editing".
// When this function is called, the justInsertedRowId JavaScript variable is set to the
// id of the row which was just inserted, or to zero (0) if the save operation which just
// occurred was not an insert.  This allows implementation of custom features such as
// "Add New ... and Continue Editing".
function postSaveHook() {
    ...
}

// This function can be customized to add additional parameters onto AJAX URLs.
// Care should be taken to determine whether there is already a ? in the URL, and use the
// appropriate separator for the first query parameter that is appended to the URL.
// Note that for (deprecated) dataTables search components, this is only called once when
// the datatable component is created, rather than every time the datatable refreshes its
// data from the server.
function fixupAJAXURL(url) {
    ...
    return url;
}
```

CHAPTER

SIX

FILTERS AND VALIDATORS FOR FORMS

6.1 Introduction

When a row is saved in add mode or update mode, the form is posted back to the server-side PHP page (the main page, part of the model). A special, hidden form field named "command" is first populated with a command telling the PHP page to save a database row. Because the data which is coming in is user-entered data, and because it is easy to "fake" a form post using many different software utilities, we have no real clue as to the validity of the data which is coming in. If we simply accept everything that comes our way, we will eventually end up with a lot of garbage in our database tables.

The solution to this problem is filtering and validation. If a field is supposed to be an integer in the range of 5 to 37 inclusive, we can first filter the value to ensure that it indeed is an integer, tossing away any characters which don't belong. If the field is not a properly formatted number, this would result in an integer value of zero. Then we can apply validation to confirm that the filtered value does not fall outside the allowed numeric range for that field. If it does, we can generate an error message which is specific to that field, and is displayed in red above the field on the form. Additionally, we can refuse to take action on the form until all fields pass their validation criteria.

Jax Framework ™ provides a variety of filtering and validation classes. These exist, by default, in the html/jax/classes/crud/filter and html/jax/classes/crud/validator directories under the Jax Framework ™ root directory. If necessary, you can move this directory elsewhere and adjust the include paths in the YAML files where these classes are loaded, in order to accommodate the move. However, for ease of updates, it is recommended to keep the Jax Framework ™ files in their original locations.

6.2 Filters

6.2.1 The Filter Abstract Class

The Filter abstract class is very simple. It takes an array of parameters for its constructor. These come from the params attribute of the filter entry in the YAML file. Each individual filter accepts a different set of parameters.

```
abstract class Filter {
    public function __construct($params = array()) {
    }

    public abstract function filter($db, &$row);
    }
}
```

The following excerpt from gencfg/user.yaml illustrates the use of the TrimFilter and LowerFilter filter classes to trim leading and trailing whitespace from user_name column and convert it to lowercase:

```
filters:
  user_name:
    trim:
      class: TrimFilter
      params: { valueName: user_name }
    lower:
      class: LowerFilter
      params: { valueName: user_name }
```

6.2.2 Filters and their Parameters

Class: IntFilter

Functionality: Converts the value to a PHP integer.

IntFilter Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be filtered. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
convertZeroToNULL	Boolean true to convert a zero value to NULL; false to not. Defaults to false.

Class: DecimalFilter

Functionality: Converts the value to a PHP double and optionally rounds it to a specified number of fractional digits.

DecimalFilter Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be filtered. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
fractionalDigits	An optional nnumber of fractional digits to use when rounding the value. This may be positive, zero or negative. If this parameter is not present, no rounding will occur. Refer to the PHP round() function for an in-depth description of how rounding occurs.
convertZeroToNULL	Boolean true to convert a zero value to NULL; false to not. Defaults to false.

Class: LowerFilter

Functionality: Converts the value to lowercase.

LowerFilter Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be filtered. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
convertEmptyToNULL	Boolean true to convert an empty value to NULL; false to not. Defaults to false.

Class: UpperFilter

Functionality: Converts the value to uppercase.

UpperFilter Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be filtered. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
convertEmptyToNULL	Boolean true to convert an empty value to NULL; false to not. Defaults to false.

Class: TrimFilter

Functionality: Trims leading and trailing whitespace from the value.

TrimFilter Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be filtered. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
convertEmptyToNULL	Boolean true to convert an empty value to NULL; false to not. Defaults to false.

6.3 Validators

6.3.1 The Validator Abstract Class

The Validator abstract class is more complex than the Filter class. The listing below is simplified for brevity. The validation functionality is handled by the validate() function. The Validator class also takes an array of parameters for its constructor, just like the Filter class. These come from the params attribute of the validator entry in the YAML file. Each individual validator accepts a different set of parameters.

```
abstract class Validator {
    // Validate value(s) based on the rules of this validator.
    // $db is an open database Connection object.
    // $row is an object whose attributes contain at least the value(s) being validated by this
    // validator.
    // Returns a string containing the error message, or an empty string if no error.
    public abstract function validate($db, &$row);
}
```

The following excerpt from gencfg/user.yaml illustrates the use of the NotEmptyValidator, LengthValidator and NoDuplicatesValidator validator classes to validate that the user_name field is not empty, does not exceed a specified length, and is unique (not duplicated in another user row).

```
validators:
  user_name:
  notempty:
    class: NotEmptyValidator
    params: { valueName: user_name }
```

```
length:
  class: LengthValidator
  params: { valueName: user_name, maxLength: 32 }
noduplicates:
  class: NoDuplicatesValidator
  params:
    table: user
    fields:
      user_name: { field: user_name, type: string, queryOperator: = }
      id: { field: id, type: int, queryOperator: <> }
    errorMsg: The selected Username is already in use.
```

All validators contain default error messages which they emit when something doesn't validate correctly. However, you can override those default error messages for most (if not all) validators by specifying the errorMsg parameter for the validator. If you do this, then when a validation error occurs, the validator will emit the error message you specified instead of its own default error message.

6.3.2 Validators and their Parameters

Class: NotZeroValidator

Functionality: Validates that a numeric field is not zero.

NotZeroValidator Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be validated. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
type	The PHP type to convert the value to before checking for non-zero. Must be one of: int, double or boolean.
allowNULL	Boolean true to allow NULL values to pass through un-validated; false to not. Defaults to false.

Class: RangeValidator

Functionality: Validates that a numeric or string value is within a specified range.

RangeValidator Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be validated. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
type	The PHP type to convert the value to before checking for in-range. Must be one of: int, double, boolean or string. For numeric values, numeric comparisons are done. For string values, string comparisons are done which may be either case-sensitive or case-insenstive.
caseInsensitive	True if string comparisons should be case insensitive. Ignored if type is not string.
min	Optional lower-bound value to compare against. If omitted, no lower-bound comparison will be performed.
max	Optional upper-bound value to compare against. If omitted, no upper-bound omparison will be performed.
allowNULL	Boolean true to allow NULL values to pass through un-validated; false to not. Defaults to false.

Class: ListValidator

Functionality: Validates that a value is equal to one of the values contained in a list of valid values.

ListValidator Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be validated. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
type	The PHP type to convert the value to before checking for in-list. Must be one of: int, double, boolean or string. For numeric values, numeric comparisons are done. For string values, string comparisons are done which may be either case-sensitive or case-insenstive.
caseInsensitive	True if string comparisons should be case insensitive. Ignored if type is not
string.	
validValues	An array of accepted (valid) values for the field being validated.
allowNULL	Boolean true to allow NULL values to pass through un-validated; false to not. Defaults to false.

Class: NotEmptyValidator

Functionality: Validates that a string value is not empty.

NotEmptyValidator Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be validated. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
allowNULL	Boolean true to allow NULL values to pass through un-validated; false to not. Defaults to false.

Class: LengthValidator

Functionality: Validates that a string value's length is within a specified minimum and/or maximum length.

LengthValidator Parameters

Parameter Name	Description
valueName	The field name of the field or table column to be validated. This may or may not represent an actual column in the database table. This is optional. If omitted, it defaults to the grandparent key (the key under the "filters" key) which should generally be the name of the field.
minLength	Optional minimum allowed length for the value. If omitted, no minimum-length comparison will be performed.
maxLength	Optional maximum allowed length for the value. If omitted, no maximum-length comparison will be performed.
allowNULL	Boolean true to allow NULL values to pass through un-validated; false to not. Defaults to false.

Class: EmailAddressValidator

Functionality: Validates that a string value contains a properly-formatted email address.

EmailAddressValidator Parameters

Class: NoDuplicatesValidator

Functionality: Validates that a value or combination of values does not already exist in a database table.

NoDuplicatesValidator Parameters

NoDuplicatesValidator field entry attributes

Attribute Name	Description
field	The value name/database column name to check for duplicates.
type	The PreparedStatement data type for this field. Must be one of: int, float, double, boolean, string or binary.
queryOperator	The SQL query operator to use for the query. If not specified, defaults to '='.

Class: ForeignKeyValidator

Functionality: Validates that a value or combination of values references an existing row in a database table.

ForeignKeyValidator Parameters

ForeignKeyValidator foreignKeyMapping entry attributes

Attribute Name	Description
local	The value name from the form to match to a column in the referenced database table.
foreign	The referenced column in the referenced database table.
type	The PreparedStatement data type for this field. Must be one of: int, float, double, boolean, string or binary.

CHAPTER

SEVEN

PERMISSION SYSTEM

7.1 Introduction

The permission system in Jax Framework ™ is a role-based system which consists of users, roles and permissions. One or more permissions are assigned to a role, and one or more roles are assigned to a user. By using roles, we can group related permissions together and manage things much better. A large software application may include hundreds of different permissions, and it would be very tedious to sift through all of these permissions to select the correct ones each time a new user is added. Instead, we use roles. The list of permissions a user possesses is equal to the sum of all permissions in all roles to which the user belongs.

Individual permissions may be used to grant access to pages within the system, or to grant access to specific features within individual pages.

The files which comprise the core of the permission system are html/classes/Permissions.class.php, html/include/requireLogin.include.php, html/include/loginForm.include.php, and html/logout.php. The include/leftNav.include.php may also be considered part of the permission system because it performs permission checks to determine which pages are available to the current user based on the pages' permission requirements and the permissions the current user possesses.

There are three places where permissions checks are performed:

- The navigation menu determines which pages the user has access to, and only displays links to pages which the user is allowed to access.
- When a page which requires permissions is first loaded, it performs a secondary check to confirm that the user has permissions to access that page. If not, the page immediately shuts down with an error message.
- Within a page which the user is allowed to access, certain features of that page may require additional permissions beyond those required to access the page. This allows specific features to be turned on and off for individual users by simply assigning (or not assigning) that user to a group which has those permissions. For example, maybe normal users can view and add orders, but only a manager can change or delete orders – even though the same page is performing all of these functions.

7.2 Sessions and the Login/Logout Process

Any page which requires that the user be logged in before viewing the page, must simply include the include/requireLogin.include.php file at the top, before any actual page logic is executed. This include file checks whether there is a login session for the current user. If not, instead of allowing the page to execute, it emits a login form and exits.. The URL in the browser window does not change. This means that once the user is logged in, the page which was originally requested is rendered as it would have been if the user had been logged in in the first place. In other words, there is no redirect to a special login page, so ther is no need to keep track of which page to go back to afer successful login.

The include/requireLogin.include.php include file is also smart enough to handle "Keep me logged in" functionality, supporting a configurable (20 by default) number of keep-me-logged-in session per user. This enables a user to log in on several different computers, each with the "Keep me logged in" feature enabled. When the browser is closed, the cookie remains in the browser. The next time the user comes back to the site, if "Keep me logged in" was enabled, the user is automatically logged back in (assuming the keep-me-logged-in session has not expired).

The login form which is presented to the user is rendered by including include/requireLogin.include.php. You can customize this form within reason, but it is important that the input fields retain their value names unless you also modify requireLogin.include.php to accept different value names.

Jax Framework ™ was designed to allow you to either use its permission system as-is, or customize it to your needs. This may include using a different mechanism to represent users. Keep in mind that the Permissions class requires a unique integer identifier for every user. This is represented by $loggedInUser->userId. The CRUD generators allow you to specify a different PHP expression to retrieve this integer user identifier. Hand-written pages will need to be adapted as well, if you change this.

Customizations to the permission system may include:

- Providing your own tables or other storage/access methods for for users, roles or permissions
- Customizing the login form or login mechanism
- Changing how the session information is stored

Any customization which changes the storage/access methods for user, roles or permissions will also require modifications to the Permissions class, so that it will know how to access users, roles and permissions. Modifications may also be required to the other files previously mentioned.

The key thing to remember is that the login and permission system that is provided with Jax Framework ™ is a robust starting point. You are free to customize it to the needs of your application.

7.3 The Permissions Class

The Permissions class looks something like this (code removed for brevity):

```
class Permissions {
    public static function hasPermissions($user_id, $permissions) {
        ...
    }

    public static function getRequiredPermissionsForScript($scriptFilename) {
        ...
    }

    public static function hasPermissionsForScript($user_id, $scriptFilename) {
        return Permissions::hasPermissions(
            $user_id,
            Permissions::getRequiredPermissionsForScript($scriptFilename)
        );
    }

    public static function inScriptPermissionsCheck($user_id, $scriptFilename, $showMenuIfFailed) {
        ...
    }
}
```

The hasPermissions() function returns true if the specified user has all of the permissions listed in $permissions; false if the user is missing one or more of the permissions. The $permissions argument can be either an array of permissions

or a comma-separated list of permissions. This function is used everywhere to determine whether the current user has a specific set of permissions. This function can be used inside a PHP page to enable/disable certain features within the page based on whether the user has the required permissions for that feature.

The getRequiredPermissionsForScript() function takes one argument: the path to a PHP page (server-side/filesystem path). It returns an array containing all permissions required to run the page. This does NOT include permissions required to turn on optional features within the page.

As you can see from the code above, the hasPermissionsForScript() function takes an integer user identifier and the server-side filesystem path to a PHP page, and returns true if the user has permission to access that page.

The inScriptPermissionsCheck() function is called at the top of every page by doing something like this:

```
include dirname(__FILE__).'/classes/Permissions.class.php';
Permissions::inScriptPermissionsCheck($loggedInUser->id, __FILE__, true);
```

Note how the second argument is __FILE__, which gives the server-side path to the PHP file which is currently being executed. If the user has permission to access the page, this function returns without doing anything, allowing the page to operate normally. If the user does not have permission to access the page, this function presents an error message and exits the page by calling the exit() function. The third argument enables the normal header and footer includes (include/header.include.php and include/footer.include.php) which would result in the full header, navigation menu and footer being shown. If this argument is false, an empty HTML header will be emitted instead of header.include.php, and simple closing body and html tags will be emitted instead of footer.include.php; the end result being that the header, navigation menu and footer would not be shown.

7.4 Other Files in the Permission System

7.4.1 The requireLogin.include.php and loginForm.include.php Include Files

These files implement the login mechanism and the keep-me-logged-in functionality. By including requireLogin.include.php at the top of a page before any other page logic, the page automatically becomes a login-secured page. If the user is not logged in, a login form will be presented.

The requireLogin.include.php and loginForm.include.php file can be customized to suit your application's needs. If you need to integrate external or custom login functionality, this is the place to do it.

7.4.2 The logout.php Page

When this page is executed, the current login session is destroyed and any keep-me-logged-in cookie is deleted. The result is that the next page which is accessed by the browser which requires a login, will result in a login form being presented.

7.5 Highly Customizable

The login, permissions and navigation subsystems provided with Jax Framework ™ are tightly integrated. However, they may or may not be sufficient for your needs. Feel free to customize these subsystems, or replace them entirely. They are only provided as a starting point.