Symfony: Getting Started

Where to get Symfony?

You can get Symfony from symfony-project.com. To start quickly, you should download the Symfony Sandbox. This is a ready-made empty Symfony project that you can start with.

Directory Structure

Important directories:

1. Apps – contains PHP source code and templates of the application.
2. Cache – contains cache files generated from PHP templates and source code. Sometimes you may need to clear this to fix some issues.
3. Config – Keeps the database schema and other configuration files.
4. Web – contains controller files, image files, CSS and JavaScript files.

What is an application?

An application is a part of the site. Most simple sites will only have 1 or 2 applications. For example, a site might have a 'frontend' application for normal users and a 'backend' application for administration. Applications can have modules, and each modules can share a common template file. Applications are stored in the /apps directory. If there is an application called 'frontend' it will be stored in /apps/frontend

What is a module?

A module is a group of pages, this can be a section of a site or a group of pages with similar function. Modules exist inside applications. Modules are stored in the modules directory of an application. If we had a module called 'products' inside the 'frontend' application, it will be at /apps/frontend/modules/products

Symfony command line tool

Symfony has an easy command to do many things, including creating applications and modules. When the server is using Linux, you have to enter the directory and type './symfony' to use that command. For Windows, you can type 'symfony'. If you don't specify an argument, the command will return all possible symfony actions:

clear-cache	clear cached information
clear-controllers	clear controllers
disable	disables an application in a given environment
downgrade	downgrade to a previous symfony release
enable	enables an application in a given environment
fix-perms	fix directories permissions
freeze	freeze symfony libraries
init-app	initialize a new symfony application
init-batch	initialize a new symfony batch script
init-controller	initialize a new symfony controller script
init-module	initialize a new symfony module
init-project	initialize a new symfony project
log-purge	purges an applications log files
log-rotate	rotates an applications log files
plugin-install	install a new plugin
plugin-list	list installed plugins
plugin-uninstall	uninstall a plugin
plugin-upgrade	upgrade a plugin
propel-build-all	generate propel model and sql and initialize database
propel-build-all-load	generate propel model and sql and initialize database, and load data
propel-build-db	create database for current model
propel-build-model	create classes for current model
propel-build-schema	create schema.xml from existing database
propel-build-sql	create sql for current model
propel-convert-xml-schema	create schema.yml from schema.xml
propel-convert-yml-schema	create schema.xml from schema.yml
propel-dump-data	dump data to fixtures directory
propel-generate-crud	generate a new propel CRUD module
propel-init-admin	initialize a new propel admin module
propel-init-crud	initialize a new propel CRUD module
propel-insert-sql	insert sql for current model
propel-load-data	load data from fixtures directory
sync	synchronise project with another machine
test-all	launch all tests
test-functional	launch functional tests for an application
test-unit	launch unit tests
unfreeze	unfreeze symfony libraries
upgrade	upgrade to a new symfony release

Task aliases:

app	pake init-app
batch	pake init-batch
cc	pake clear-cache
controller	pake init-controller
module	pake init-module
new	pake init-project

How to create an application?

It is difficult to create all the files and directories needed to create an application, so symfony has a command that can generate it for you. The command to create an application called 'myapp' is './symfony init-app myapp'. You can give any name to your application.

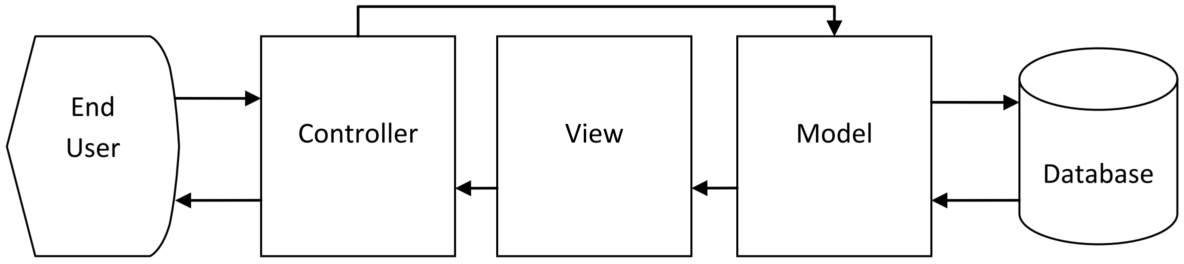
How to create a module inside an application?

Your application cannot be used unless it has modules, because all the pages in your application are inside modules. To create a module, use this command: './symfony init-module myapp mymodule' Where 'mymodule' is the name of your module. This name will show in the page URL when you access your page. If you have 'mymodule' the URL will normally be like this:

http://myserver/mycontroller.php/myapp/mymodule/mypage

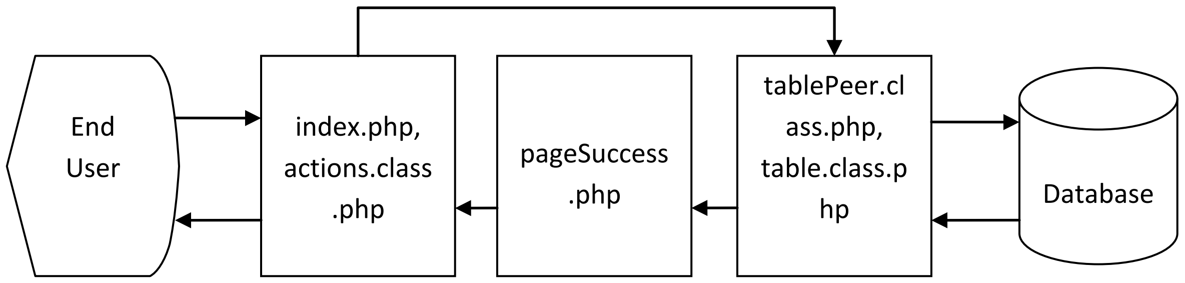
Understanding the MVC architecture in Symfony

Model-View-Controller (MVC) is a pattern used in object oriented programming (but can also be used in non-OOP). It is especially popular in the web development world. The basic idea is to separate the application into 3 parts – the model, view and controller. Here is what the MVC architecture looks like:



1. First, the end user sends a request for a page on the server. The request is accepted by the controller. The controller takes care of any user interaction, like AJAX requests or form handling or just clicking a link and requesting a new page.
2. The controller might then invoke the model and the model can read/write to the database.
3. Then the model will be used to generate the view (HTML/JS/CSS, etc.) and that view is sent back to the end user through the controller.

In Symfony, the diagram is something like this:


End User
index.php, actions.class .php
pageSuccess .php
tablePeer.class.php, table.class.php
Database

The controller is split into 2 parts, one handles the basic HTTP requests, finds the right files to include, etc. That is automatically created by Symfony by using this command: './symfony init-controller myapp prod' That will create a controller file called 'myapp_prod.php' in the 'web' directory. To access a page, you go to 'http://myserver/myapp_prod.php/module/page' if you rename myapp_prod.php to index.php, you can use 'http://myserver/module/page' The last argument of the command can be 'test', 'dev' or 'prod' They are for testing, development and production. If you say 'dev' then it will enable the symfony debugger and you can easily check which queries were run in this page, how long did the page take to load, etc. If you say 'prod' there will be no special features and PHP error messages will be off. If you say 'test' PHP errors will be on but there will be no debugging.

The second part of the controller is different for every module. When you create a module, and go to the module directory at 'apps/myapp/modules/mymodule/actions' you will find the file 'actions.class.php' This file will contain all the logic (except database queries) for your page. By default, it will have only one function: executeIndex(). This is run when you go to the module's default page: 'http://myserver/mycontroller.php/module' Here is an example executeIndex function:

```
public function executeIndex()  {
  $this->myVar = 'hello world';
}
```

It is setting a variable called 'myVar' that can now be used in the template (view) file. The view file for the default index page of the module is at 'apps/myapp/modules/mymodule/templates' called indexSuccess.php. Here is an example indexSuccess.php:

```
<p><?php echo $myVar; ?></p>
```

The result of this page (http://myserver/mycontroller.php/module) will be like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta name="robots" content="index, follow" />
<title>symfony project</title>
<link rel="shortcut icon" href="/favicon.ico" />
</head>
<body>
<p>hello world</p>
</body>
</html>
```

You must be wondering, where did all that extra HTML come from? Your template/view file had only one line. The answer is each app also has its own shared template. Normally this will include the header, navigation and the footer. The file that stores this global template is 'apps/myapp/templates/layout.php'.

You will notice that there is some PHP in this file:

<?php include_http_metas() ?> - outputs <meta> tags with HTTP information like file encoding (example ISO-8858-1 or UTF-8)

<?php include_metas() ?> - outputs <meta> tags with other information like keywords, robots, etc. This is used by crawlers and search engines.

<?php include_title() ?> - outputs the page <title> tag. The default title is set in 'apps/myapp/config/view.yml'

actions.class.php Explained

The file actions.class.php generally contains all the processing for the page. You can have the processing in the template file also, but that is not recommended because it will not separate the template from the code and then there is not much point of using the MVC pattern. Whenever you create a new template file, you will not be able to access it unless you create an execute function for that new file. For example, if you created a file called 'myPageSuccess.php' inside the module 'mymodule' (directory apps\myapp\modules\mymodule\templates) then you must also create a method called executeMyPage() inside the action class in apps\myapp\modules\mymodule\actions\actions.class.php. Note that whenever you create a template, you will append the word 'Success' to it. There are other cases in which the file name will be different, like 'myPageError.php', but that is only done when you want to perform some validation before the page loads and if you want to load a different template if validation fails.

Here is an example of a simple actions.class.php:

```
class AdminUsersActions extends sfActions
{
 public function executeIndex()
 {
  // do something
 }

 public function executeList()
 {
  // do something
 }
}
```

For that, you need 2 template files, indexSuccess.php and listSuccess.php. But if you had a more complicated actions.class.php file with validation, it might look like this:

```
class AdminUsersActions extends sfActions
{
 public function validateIndex()
 {
  // do validation and return true or false
 }
 public function handleErrorIndex()
 {
  // do something if validation returns false, will use indexError.php template
 }
 public function executeIndex()
 {
  // will be run if validation method returned true, will use indexSuccess.php template
 }

 public function executeList()
 {
  // do something
 }
}
```

In this case, we need 3 files – listSuccess.php, indexSuccess.php and indexError.php. indexSuccess will be used when the validation function returns true (no validation errors), otherwise, indexError will be used. Inside the actions class, you have access to many Symfony methods and properties that might be useful. They are discussed below.

HTTP Request Handling

In normal PHP, we use the super-global variables $_GET, $_POST, and $_REQUEST to read the request parameters. But Symfony does not use the normal http://server/webpage/?param=value syntax. When you create a link like that, Symfony will convert it to something like http://server/webpage/param/value. This is done automatically by the link_to() helper, which is used to create links. An example is:

```
link_to('mymodule/mypage?id=1', 'Click here');
```

Will be converted to:

```
 <a href="mycontroller.php/mymodule/mypage/id/1">Click here</a>
```

You are probably wondering why we need the link_to helper and how we are now going to read the ID from the URL. The reason we need the link_to helper is to add the controller, because we don't know which controller is going to call our script. In addition to that, it also handles the conversion of URL parameters as we just saw. To read these parameters, Symfony provides the getRequestParameter() method. For example, if you wanted the id, you would write the following code:

```
function executeIndex() {
  $id = $this->getRequestParameter('id'); // id is now 1
}
```

You can also use the getRequestParameter() method to read values sent to your page by HTTP POST (form submissions). So if you had a login form, instead of $_POST['username'], you can simply use $this->getRequestParameter('username') with the same result.

Forwarding, Changing Templates

Sometimes you might want to forward the user to another page inside your action, or use another template file instead of the default.

To forward, Symfony provides the forward() method. This takes the module name and the page name. You can call it in your execute method like this:

```
$this->forward('mymodule', 'mypage');
```

Note that this may be case sensitive depending on the server OS. Sometimes you might also want to forward to a 404 page depending on whether or not some condition is met. The syntax for that is:

```
$this->forward404Unless($condition);
```

If $condition is true, it will *not* forward. If it is false, it will forward to the default 404 page.

In Symfony, when you use the forward functions, it is as if that page will be included instead of the current page. This is more like PHP's include() rather than PHP's header('Location: …'), so the URL will not change to mymodule/mypage if you used the forward method. However, if you want the URL to change, you can use the following method:

```
$this->redirect('RegisteredUsers/list');
```

If you want to use another template instead of a template with the same name as your function, you can use the setTemplate() method:

```
function executePage() {
  $this->setTemplate('anotherPage'); // use anotherPageSuccess.php instead of pageSuccess.php
}
```

Dynamically Setting the Page Title

Sometimes, you will want to set the page title dynamically – maybe from the database, or maybe on a page by page instead of module by module or site-wide default. You can do this by calling the following in your action:

```
$this->getResponse()->setTitle('My Random Title: '.rand(0, 100));
```

getResponse() contains the response document being generated by Symfony. It will also contain HTTP headers, and other things being sent to the browser.

Symfony & Databases

Perhaps one of the most important ways in which the Symfony framework is different is in how it handles databases. Symfony bundles a database abstraction layer called Creole. It also uses an ORM (Object Relation Mapping) library called Propel. Propel can generate all your database (model) classes automatically and even add functions that handle the relationships between your tables. A database abstraction layer is a layer between the database drivers and your application that allows your application to access any database using the same code. So instead of writing SQL, you use the methods provided by Creole and Propel, and these will generate the SQL that is optimized for the database that you have configured it for. While all this sounds good, it also means that you have to learn a different syntax to interact with the database.

Symfony Database Configuration

Before you can start working with the database, you must configure Symfony and tell it which DBMS (database management system), which server, which username, password and database you want to use. You need to set this configuration in 2 files. The first file is /config/databases.yml:

```
all:
  propel:
    class:     sfPropelDatabase
    param:
      dsn: mysql://mysqluser:mysqlpassword@mysqlserver/mysqldatabase
```

That is the normal format for the file. You need to change the DSN (Data Source Name) line to your database. For example if you wanted to use MySQL with user admin and password 111111 on the server mysqlserv and database symfonytest, you would change the DSN to:

```
mysql://admin:111111@mysqlserv/symfonytest
```

The second file you need to change is /config/propel.ini There are 2 attributes in the file that you need to change - propel.database.createUrl and propel.database.url:

```
propel.database.createUrl  = mysql://admin:111111@mysqlserv/
propel.database.url        = mysql://admin:111111@mysqlserv/symfonytest
```

You use the same format as the DSN in databases.yml, except for createUrl, you don't include the database name.

YAML Syntax

YAML (pronounced Yamel) is a data definition language similar in syntax to JSON and similar in purpose to XML. We will not go into the details of the YAML syntax because it is not necessary for a quick start guide. You just need to know that in YAML, the basic syntax is like this:

```
root:
  firstchild:
    lastchild: value
```

To represent the same in XML, you would use this syntax:

```
<root>
  <firstchild>
    <lastchild>value</lastchild>
  </firstchild>
</root>
```

Notice that the YAML is shorter. That is one of the reasons YAML was selected for Symfony over XML. Note that in YAML, you should *never* use tabs. If you use a tab, it will result in a parse error and the YAML will be unreadable. For indentation, you should use 2 spaces. Another thing to remember is that in YAML, the indentation has meaning. For example, in XML you can do this:

```
<root><firstchild>
        <lastchild>value</lastchild></firstchild>
              </root>
```

And even though it looks like nonsense to humans, XML parsers can understand it. But if you change the indentation in YAML, it will stop working. So this is invalid:

```
root:  firstchild:  lastchild: value
```

Unlike XML, YAML does not need a root node. You can have many top level nodes:

```
toplevel1:
  child1: value
```

```
toplevel2:
  child2: value
```

The schema.yml File

Every Symfony project that is going to use the database has one schema.yml file in the /config directory. Since different databases use different SQL conventions, for example if you used the TEXT data type in MySQL, in Oracle you would use LONG or VARCHAR2. Because of this, we need a standard that can be converted by Propel to any database format. This is the purpose of the schema.yml file. Since it can take a long time to explain everything and can also be boring, let's start with an example schema.yml file instead (see next page):

```
propel:

 weblog_post:
  _attributes: { phpName: Post }
  id:
   type: INTEGER
   autoIncrement: true
   primaryKey: true
   required: true
  title:
   type: VARCHAR
   size: 255
  excerpt:
   type:  LONGVARCHAR
  body:
   type:  LONGVARCHAR
  created_at:
  _uniques:
   unique_title:
    - title
 weblog_comment:
  _attributes: { phpName: Comment }
  id:
   type: INTEGER
   autoIncrement: true
   primaryKey: true
   required: true
  post_id:
   type: INTEGER
   required: true
   foreignTable: weblog_post
   foreignReference: id
   onDelete: CASCADE
   onUpdate: CASCADE
  author:
   type: VARCHAR
   size: 255
  email:
   type: VARCHAR
   size: 255
  body:
   longvarchar
  created_at:
  _indexes:
   index_create_date:
    - created_at
```

The first line is always ‘propel:’. The second level lines ‘weblog_post’ and ‘weblog_comment’ are tables. Propel will generate the database structure and the PHP class files for every table. Below is a table explaining the attributes for a table:

Special Table Attribute Name	**Purpose**
_attributes	Generally, you will not use this except for the phpName attribute. The phpName attribute tells Propel what to name the PHP class that is generated. In this case, the class that will contain the methods for reading and writing to the weblog_comment table will be called Comment.

_indexes	This allows you to create indexes for the table. You should create indexes properly so that the database performance is good. Hereis an example index:
	`  _indexes:`
	`    myIndexName:`
	`      - myIndexCol`
	`    myIndexName2:`
	`      - myIndexCol1`
	`      - myIndexCol1`
	
	The first index has 1 column, the second index covers 2 columns.

_uniques	This is same as _indexes, except it creates unique keys.

You will notice that individual fields inside a table also have attributes. The table below explains the attributes for fields:

Attribute Name	Purpose
type	This is the data type of the field. Note that some data types may not be supported here, such as TEXT. For TEXT, use LONGVARCHAR instead. Most common data types are supported: INTEGER, BIGINT, VARCHAR, TINYINT, CHAR
size	This is the size of the data type. In normal SQL it is the number in brackets after the data type. For example if you had VARCHAR(255), you would set the type to VARCHAR and the size to 255.
required	This sets the NULL/NOT NULL constraint. The value can either be true for NOT NULL constraint or false, to allow NULL. By default, this is set to false, so fields can be NULL.
default	This sets the default value of the field in case no value was provided during an INSERT statement.
foreignTable	If the field is a foreign key from another table, this is set to the parent table's name.
foreignReference	If the field is a foreign key, this is set to the name of the parent field in the parent table.
onUpdate	If the field is a foreign key, this defines which action to perform when an update is performed on the parent column. The default action is NO ACTION. Other valid values are CASCADE (update the child row if the parent is updated) and SET NULL.
onDelete	If the field is a foreign key, this defines which action to perform when a row in the parent table is deleted. The default action is NO ACTION. Other valid values are CASCADE (delete the child row if the parent is deleted) and SET NULL.
primaryKey	This can be set to true or false and determines whether the row is a primary key or not.
autoIncrement	For primary keys, this allows you the table to generate its own IDs.

Generating the Database & PHP Propel Classes

To generate the database and PHP propel classes, simply run the following command:

```
./symfony propel-build-all
```

It will automatically generate the database, tables, all foreign keys and other constraints and then generate the PHP classes. Whenever you do this in a real project, make sure you backup the database first! Otherwise you might lose all your data!

Querying the Database

Now that we have the database set up, we can write code to query the database and do useful things with it. If you go to the '/lib/model' directory, you will see the classes that were generated by Propel when you ran the ‘propel-build-all’ command. However, when you open a file, you will see that it is an empty class:

```
<?php
/**
 * Subclass for representing a row from the 'weblog_comment' table.
 *
 *
 *
 * @package lib.model
 */
class Comment extends BaseComment
{
}
```

Normally you will add your own functions to this class, and then when you modify your database and generate your Propel models again, it will not overwrite your file. But you notice that it is inheriting from the BaseComment class. You can find that file in the ‘lib/model/om’ directory. If you open that directory, you will see 2 files with similar names – BaseComment.php and BaseCommentPeer.php. BaseComment.php contains the setter and getter methods for all the columns in that table. It also contains functions to insert, update, and delete. You will never use most of these functions. The most important functions that you need to know are:

getColumnName() – get the value of a column

setColumnName($value) – set the value of a column (usually before an insert or update)

save([$connection]) – INSERT if the row is new, UPDATE if the primary key exists

delete([$connection]) – DELETE a row

Note that the getter and setter methods follow a special naming convention. They will convert under_score syntax to lowerCamelCase syntax So if you have a column called post_id, the get and set will be getPostId() and setPostId(). The BaseComment object is normally returned after you do a select query. Otherwise you can always use it by initializing the Comment() class like this: $comment = new Comment();

The other file, BaseCommentPeer.php is a class that you will normally call statically. This means you will not instantiate it with a new Class(), but rather use the Class::method() syntax. Below are noteworthy functions inside the BaseCommentPeer class:

doCount($criteria[, $distinct[, $connection]]) – count the number of rows using the given criteria

doSelect($criteria[, $connection]) – select all rows matching the given criteria. This returns an array of Comment objects.

doSelectOne($criteria[, $connection]) – selects one row matching the given criteria. This is essentially like LIMIT 1. This returns one Comment object.

doSelectJoinPost($criteria[, $connection]) – selects and joins with the post table with criteria. Since we specified that the post table is related to this table, Propel automatically made this function for us.

doCountJoinAll($criteria[, $connection]) – joins with all tables and does a count matching the given criteria.

The arguments in square brackets [] are not required, so you don't need to worry about them. You are probably wondering what $criteria is. We will get to that soon. Another thing you might notice is that the class has many constants that are named after the columns. The following constants are defined:

ID
POST_ID
AUTHOR
EMAIL
BODY
CREATED_AT

These are useful when you want to reference a column in a criteria object. Next, let's look at an example so you can understand how it all works:

Below is some sample code for an action method to query the database (in actions.class.php):

```
 public function executeShowComments()
 {
        // Search for a distributor
        $postId = $this->getRequestParameter('id');
        $criteria = new Criteria();
        $criteria->add(CommentPeer::POST_ID, $postId);
        $criteria->addAscendingOrderByColumn(CommentPeer::CREATED_AT);
        $this->comments = CommentPeer::doSelect($criteria);
 }
```

And then in the template file:

```
<?php
foreach ($comments as $currComment) { ?>
        <div><?php echo esc_entities($currComment->getBody()); ?></div>
        <br />
<?php
} ?>
```

$currComment is an object of the class Comment and that is why we use the getter method on it. So now that we basically know how to query the database, let’s look at the Criteria object in more detail. Below are commonly used methods in the Criteria class:

Method Name	**Purpose**
addAsColumn($name, $clause)	This will add a new column with an alias. In SQL, this looks like SELECT col+1 AS myAlias … The code for that will be: $criteria->addAsColumn(‘myAlias’, ‘col+1’)
add($column, $value[, $operator])	This will add a new condition to the WHERE clause. If you call it more than one time, it will automatically use AND to combine the conditions that you add. For example: $criteria->add(CommentPeer::POST_ID, $postId); will generate this SQL: WHERE \`post_id\` = $postId $criteria->add(CommentPeer::POST_ID, $postId); $criteria->add(CommentPeer::AUTHOR, ‘Bob’); Will generate: WHERE \`post_id\` = $postId AND \`author\` = ‘Bob’
addAscendingOrderByColumn($column)	Example: $criteria->addAscendingOrderByColumn(CommentPeer::CREATED_AT); SQL: ORDER BY \`created_at\` ASC
addDescendingOrderByColumn($column)	Example: $criteria->addAscendingOrderByColumn(CommentPeer::CREATED_AT);

	SQL: ORDER BY `created_at` DESC
setLimit($limit)	Sets the maximum number of rows to be returned by the query. Example: $cirteria->setLimit(2); SQL: LIMIT 2
setOffset($offset)	Sets the number of rows to skip in the result set. For example, if a query has 20 results and offset is 1, the first row will be skipped. SQL: LIMIT $offset, $limit
addJoin($col1, $col2[, $jointype])	Adds a table join to the query. You can specify the join type. By default, INNER JOIN is used. The join types are: Criteria::LEFT_JOIN Criteria::RIGHT_JOIN Criteria::INNER_JOIN Example: $criteria->addjoin(CommentPeer::POST_ID, PostPeer::ID); # inner join $criteria->addjoin(CommentPeer::POST_ID, PostPeer::ID, Criteria::LEFT_JOIN);
addAlias($alias, $table)	Add an alias for a table that we are joining with. Note that you cannot add an alias for the main table, only a table you are joining with.

You should now be able to create most queries that you will need to use. You might still wonder then, what about complex WHERE clauses with many conditions? Or how about when you don’t want any conditions?

If you don't want any conditions, just use new Criteria() instead of $criteria, like this:

```
PostPeer::doSelect(new Criteria()); // select all posts, SELECT * FROM weblog_post
```

If you want complex conditions, for example:

```
WHERE ((`post_id`=1 AND `post_id`=0) OR `post_id`=2)
```

That does sound like non-sense, because it will always be false, but let's just suppose it is what we want. The following criteria will generate that for us:

```
$c = new Criteria();
/* creates new criterion */
$d = $c->getNewCriterion(CommentPeer::POST_ID,1);
/* creates new criterion */
$d1 = $c->getNewCriterion(CommentPeer:: POST _ID,0);
$d->addAnd($d1); /* match two criterions */
 /* add created connected criterions */
$c->Add($d);
$c->AddOr(CommentPeer:: POST _ID,2);
$result = CommentPeer::doSelect($c);
```

However, you will rarely use this kind of complicated syntax, so you usually don't have to worry about it. Now you can create almost any kind of query that you can imagine.

Sessions and Authentication

Symfony tries to automate session handling, authentication and credential checking for you. To access the session in your actions class, the code is something like this:

```
$this->getUser()->setAttribute('myvariable', 'myvalue');
```

In normal PHP, this is equal to:

```
$_SESSION['myvariable'] = 'myvalue';
```

Similarly, when you want to get a value, you simply use:

```
$this->getUser()->getAttribute('myvariable');
```

But unlike in PHP, in Symfony you don't have to worry about starting the session or handling the session ID.There are also other methods to clear the session variable or clear a variable in the session:

```
$this->getUser()->getAttributeHolder()->clear(); // same as $_SESSION = array();
```

```
$this->getUser()->getAttributeHolder()->remove('myvariable'); // same as
unset($_SESSION['myvariable']);
```

Logging In

After your login form is submitted, you query the database, check the username and password. If it matches, then you can tell Symfony that the login was successful by using the following code:

```
// query db..

// if the password matches, then:

$this->getUser()->setAuthenticated(true);
```

This will log the user in.

Logging Out

Similarly, if you want to log out, you can set the same method to false:

```
$this->getUser()->setAuthenticated(true);
```

That will log the user out. You might also want to clear the session:

```
$this->getUser()->getAttributeHolder()->clear();
```

Enforcing Authentication

There are 2 ways to check and enforce authentication. You can tell Symfony which pages require logins by editing some YAML file, or you can do it yourself programmatically.

If you want to require login for a page, the best way to do it is to use the security.yml file. This file can be set for each application, each module or each page. If you want to set it for the whole application, go to /apps/myapp/config/security.yml By default, you will see:

```
default:
  is_secure: off
```

If you want to require login for all pages in the application by default, then change it to this:

```
default:
  is_secure: on
```

All pages will automatically display the login page now. Suppose you wanted to require login only for one module, you can set this to off, and go to /apps/myapp/modules/mymodule/config and create a file called security.yml (if it does not exist) and enter the same into it:

```
default:
  is_secure: on
```

That module is now secure. If you want only a single page in the module to be secure for example only the index page, then use the following code:

```
index:
  is_secure: on
```

This is a very easy way to enforce login, but sometimes you want the page to change depending on whether or not the user is logged in instead of redirecting the user to a login page. For this you need to check programmatically by yourself whether the user is logged in. You can do this in the actions class by the following code:

```
if ($this->getUser()->isAuthenticated()) {
		// user is logged in
}
```

That's it! You should have gathered enough knowledge about Symfony to start using it for simple projects. If you have any more questions, you can always check at http://symfony-project.com/ or the official Symfony book at http://www.symfony-project.com/book/trunk The book is also available in the RarePlay library in print.