

# Kick-start takes you to the movies: Kick-start takes you to the movies, Part 2

Finishing a personal movie database application, based on PHP and DB2

Skill Level: Introductory

[Ian D. M. Hakes \(ihakes@ca.ibm.com\)](mailto:ihakes@ca.ibm.com)  
Software Developer  
EMC

29 Nov 2006

Explore PHP and XML development using the [Eclipse IDE](#), [IBM® DB2® Express-C 9](#), and [Websphere® Application Server Community Edition](#). Learn how to configure these applications, part of a program designed to [kick-start your application development](#), to develop a Web-based movie information database. This is part two of a [two part tutorial](#), covering the primary PHP code development and DB2 database configuration and data retrieval. [Part 1](#) covered the installation and configuration of the tools, along with some basic proof-of-concept code development.

## Section 1. Welcome back!

Here at the studio, we're glad you're back to help finish off this project. The producers have given the green light to go ahead with the rest of the production, we have some A-list stars on board, and the special effects wizards have fired up the green screen. But time is money in this business, so let's get right to it.

### About this tutorial

Building on your application installation, setup, and configuration steps from

"Kick-start takes you to the movies, Part 1", this sequel continues exploring the Eclipse IDE, DB2 Express-C 9, and PHP to create a Web-based movie database. You'll take advantage of the Eclipse integrated development environment, the PHP Hypertext Preprocessor, and the new XML capabilities of DB2 9 to build your PHP interface, search for and retrieve movie information from the Web, and save that information as an XML document in DB2. Finally, you'll tie up the loose ends, wrap the project, and start writing your acceptance speech.

## Objectives

After completing this tutorial, you will understand how to store and retrieve XML data through PHP and a back end database. The applications and skills we'll cover in this specific example can be easily extrapolated into many other software development ideas for business processes.

## Prerequisites

This tutorial is written for software developers and analysts whose skills and experience are at an intermediate level. You should have a general familiarity with using an integrated development environment, and a basic knowledge of PHP Web page creation, SQL, and XML.

## System requirements

To complete the steps in this tutorial, you need:

- Workstation for the [Eclipse IDE](#)
- [PHP](#) Web server (enabled with either Apache or [WebSphere Application Server Community Edition](#) with the new alphaWorks [PHP extension](#))
- [DB2 Express-C 9](#) database server

The tasks can be handled by three different computers, or you can set up all the applications on just one system.

The details of how to install these applications are covered in much more depth and detail elsewhere in developerWorks and on these applications' respective Web sites. See the [Resources](#) for links.

The specific product installation instructions for this tutorial were covered in Part 1. I will try to be as platform independent as possible. My Web server, DB2 database, and PHP installation are all on a Linux®-based (Red Hat Enterprise 4) system, but

my Eclipse IDE is installed on a Microsoft® Windows® XP system. Of course, the setup is entirely up to you, as all of the kick-start applications are available for both Windows and Linux operating systems.

If you want to follow along with the interface creation steps, you can [download the set of PHP files](#) included with this tutorial, then use the Import function of the Eclipse IDE to bring the files into your Movie Database workspace. The alternative is to create and code them as you go along.

---

## Section 2. Production design

In the early stages of a movie's development cycle, the production designer works with the director and cinematographer to decide the visual style and feel of the movie. They decide how the characters will look, including the clothes and hairstyles, what sets will be built and what locations will be used, and what special graphic effects can be implemented.

In this section, you'll quickly look at how you want your movie database interface to look, how it will take and return data about the movies, and even how to include some special effects.

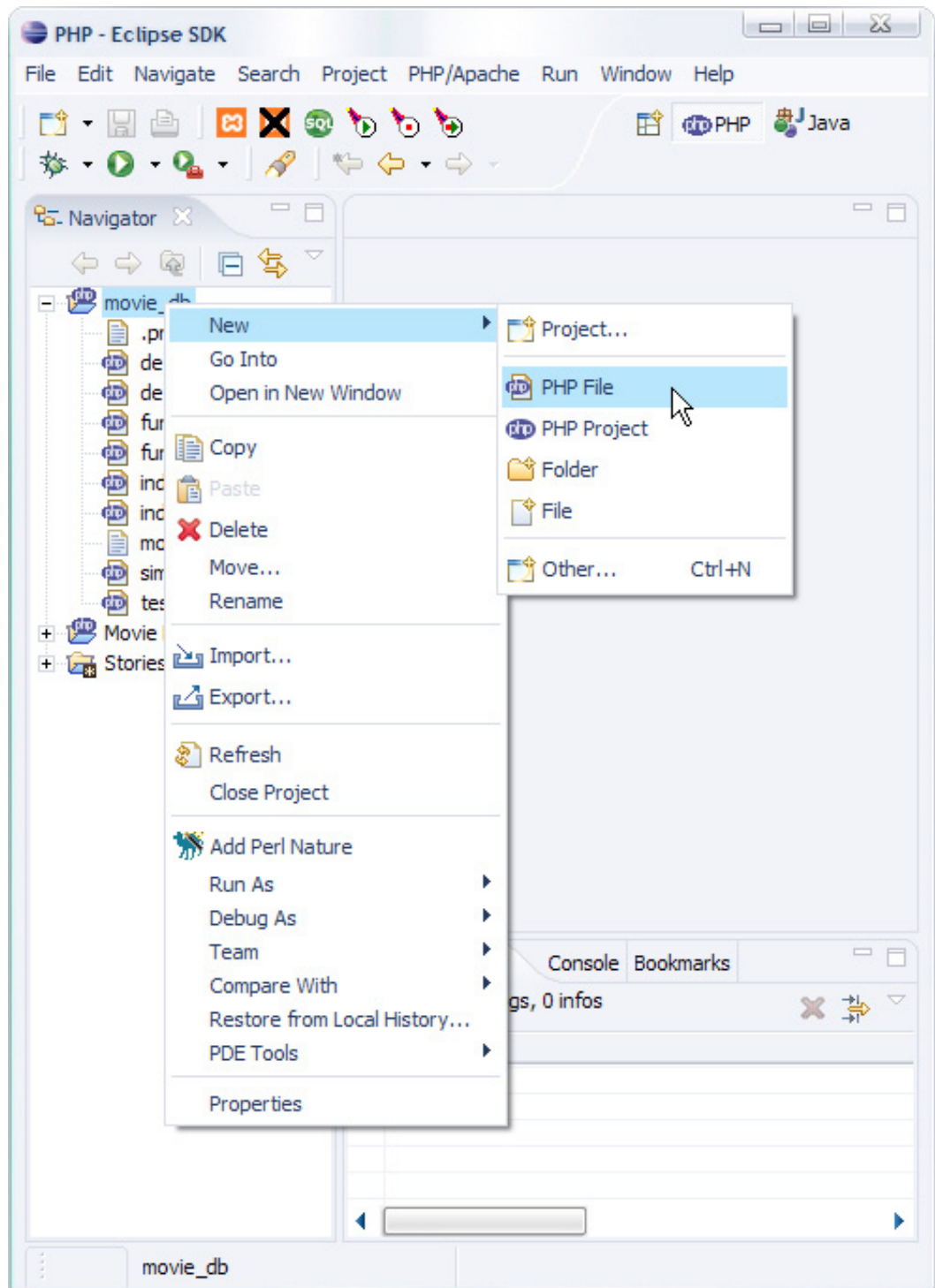
### Look and feel

At this stage of the project, focus on making things as simple as possible (the addition of extravagant GUI interfaces is beyond the scope of this tutorial). The key pieces you need are:

- A text area for typing in what you want to search on
- A choice of suitable search categories
- Basic positioning and some graphic effects

Let's create a new PHP file in your Eclipse project called index.php. Web servers will look for a file named "index" as the base page in displaying the contents of any Web directory, so you might as well follow convention in this case.

1. In Eclipse, right-click on the **movie\_db** project name, then click **New > PHP File**, as shown in Figure 1.  
**Figure 1. Create new PHP file**



2. Name the file `index.php` and click **Finish**.

In this file, you'll now create the basic form for your search:

```
$head = "<html><head><link rel='stylesheet' href='movie.css' type='text/css' \
      media='screen'><title>Movie Vault</title></head><body>";
$form = "<h3>Welcome to the Movie Vault<h3>";
$tail = "</body></html>";

echo $head;
echo $form;
echo $tail;
```

If you save the file and reload the file in your Web browser, you'll see the greeting on the base `index.php` file.

In a quick digression, let's think about those `$head` and `$tail` values. For almost every page in your Web site, now and in the future, you'll want to have that HTML, so let's put those values into just one place that you can refer to from every page. If you want to change a common element, rather than updating it within every single page you only have to change it in one spot.

Create another new PHP file, named `defaults.php`, and cut the `$head` and `$tail` declarations from the `index.php` page and paste them into `defaults.php`. The next step, to pull those declarations into your working page, is to add an include call that references `defaults.php`. Do this by adding the following line of code at the top of the `index.php` page:

```
include("defaults.php");
```

It's that easy. Now you can call any of the declarations inside the `defaults.php` file as if they were in your `index.php` page. If you didn't have the defaults in the same location on the Web server, you'd have to put the relative location in the include call (for example, `include("/templates/defaults.php");`), but for simplicity's sake, keep all your files in the same directory on the server.

Let's move on and create the basic HTML form to get data back from the user. A form needs input fields and a place to return that data for use in submitting queries or updates. In this case, you're going to ask for some text to search on, and a category. Again for the sake of simplicity, you'll return the information to the same page, rather than creating a new page to accept the data.

The coding basics of form creation are pretty simple. You need a `<form>` tag set, where you define an action (where you send the submitted information) and a method (how you send this information). Inside the form tags you have a separate `<input>` tag for each piece of information that gets passed through.

```
$thispage = $_SERVER["PHP_SELF"];
if (isset($_HTTP_GET_VARS["qstring"])) {
```

```
        $string = $_HTTP_GET_VARS["qstring"];
    } else { $string = ""; }

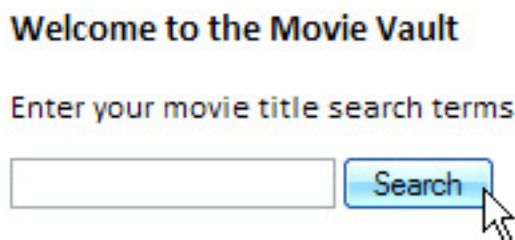
    $form = <<<TEXT
    <h3>Welcome to the Movie Vault</h3>
    <p>Enter your movie title search terms</p>
    <form name='movie' action='$thispage' method='get'>
    <input type='text' name='qstring' maxlength='64' />
    <input type='Submit' value='Search' />
    </form>
    TEXT;
```

There are a couple of tricks to note here. You've brought the page name into the PHP script as a new variable, `$thispage`, which gets the information about the page name from the PHP global server array. The second line tells your script to take the string value `qstring` from the `$_HTTP_GET_VARS` array passed to the page if it exists, or otherwise to give it an empty value. This prevents the script from returning errors that the `qstring` variable is undefined. You'll use this again later for other variables passed back to the script.

You've used PHP's "heredoc" functions in defining the `$form` variable, so you can include a large block of text without having to worry about escaping the quote characters in the form tags. It's very important to make sure you have the closing `TEXT;` of the heredoc block on a line all by itself, with no trailing white space, or you'll wind up with a parse error on the Web site about an unexpected '\$' value.

After a refresh, your Web site looks like a simple submission page, as shown in Figure 2.

**Figure 2. Initial application page**



Adding a title string to the text area and clicking **Search** on the page will now return a query string to your page, but the information is useless unless you let the PHP know that it needs to create a query from it. To tell the page that you've already made a selection, add a "flag" to the form, and some code that will recognize the flag and create a query out of your submitted information.

To add the flag, add a hidden input to your form:

```
<form name='movie' action='$thispage' method='get'>
<input type='hidden' name='sel' value='y' />
<input type='text' ...
```

To parse out a completed query, you also add an `if` structure to your main page so that submitted information is composed into a query. You can add this either above or below the `$form` variable declaration. I'm going to put it above, as it helps me to think of the PHP processing occurring from top to bottom on the page.

```
if ($sel == "y") {
    # compose and return DB2 query
}
```

For PHP security reasons, the `$sel` variable is not directly defined as the information is passed back through the form to the Web page. To retrieve the values from the form, you again have to bring in the `$HTTP_GET_VARS` global array, and extract the information inside. At the top of the page (below the `$thispage` declaration), add:

```
if (isset($HTTP_GET_VARS["sel"])) {
    $search = $HTTP_GET_VARS["sel"];
} else { $search = ""; }
```

and change the `if` statement to read:

```
if ($search == "y") {
    ...
```

## A user interface

Now that the basic code is in place, it's time to call in the production team to give this application something resembling a user interface. It is outside the scope of this tutorial to teach the details of creating a cascading style sheet. You can either leave the bare bones interface and skip the rest of this section, or extract the `movies.css` file from this tutorial's zip file and copy it to your Web directory. Then all you have to do is add this line:

```
<link rel='stylesheet' href='movies.css' type='text/css' media='screen'>
```

to the `$head` variable in the `defaults.php` file.

To take advantage of the visual display attributes of the CSS file, enclose your page in a table element. Add:

```
<table id='page' width='900px' height='100%' border='0' cellspacing='0' \
      cellpadding='0' align='center' bgcolor='#FFFFFF'>
  <tr><td height='*' valign='top' align='center'>
```

after the opening `<body>` tag in the `$head` variable, and add:

```
  </td></tr>
</table>
```

at the start of the `$tail` variable. Now if you refresh the Web page, you should have a more pleasant looking interface, with a blue backdrop and the search area centered in the white foreground.

If you want to take advantage of more of the available options in the `.css` file, by all means, go ahead. There are lots of tweaks in there, but space limitations here prevent us from exploring them in more detail.

A basic search form, and a usable graphic interface are a great start for the production department. For now, let's leave them to work on sharpening the look and feel and you'll start shooting the movie.

---

## Section 3. Principal cinematography

The actual shooting of the film usually takes a relatively short time of six to twelve weeks, but is, paradoxically, the most expensive part of the entire production. In a similar vein, your project needs to create a retrieval method for new movie information - a relatively short part of the entire application design, but likely to be the most complicated piece.

For your local movie database application you need a repository to draw information from. You need a large scale movie database that has information about thousands of existing movies. You're going to use the best source available, the Internet Movie Database ([imdb.com](http://imdb.com)). At this point, I must point out that the use of information from [imdb.com](http://imdb.com) is freely licensed for personal use only. This application cannot be turned into a commercial venture without contacting [imdb.com](http://imdb.com) directly and arranging a license for their information.

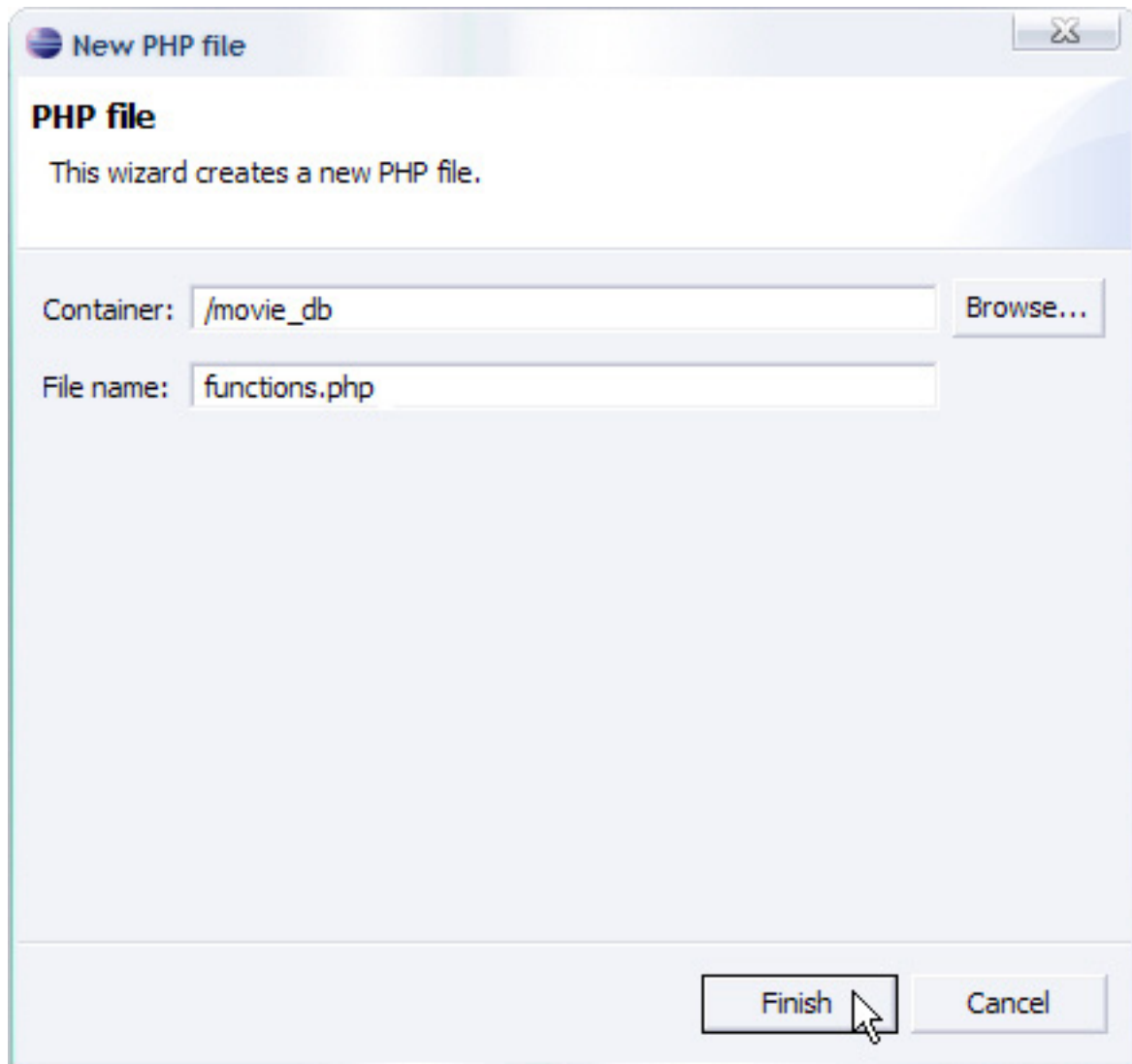
The goal of this application is to retrieve information from your local repository about the movies you have on hand. If the information isn't in your database, the next step is to visit [imdb.com](http://imdb.com), find a suitable match, then bring that information into your database for future use.

Fortunately for your purposes, a team from TRYNT Heavy Technologies ([www.trynt.com](http://www.trynt.com)) has written an API for accessing and retrieving IMDB information. Take the base search information from the API and use it to find the appropriate [imdb.com](http://imdb.com) entry, then call the API again to retrieve the detailed XML information about the movie.

Let's start by creating a new file called `functions.php` in Eclipse, which will be used to hold all the functions for your application in a single place. This helps facilitate debugging and keeps your main application files as clean and simple as possible.

In Eclipse, choose **File > New > PHP file**, and call it `functions.php` under your **movie\_db** container, as shown in Figure 3. Click **Finish**.

### Figure 3. Create a new `functions.php` file



To use these functions in your application, use a PHP inclusion statement in the `index.php` file, as you did for the `defaults.php` file. However, because you cannot run the application without these functions, we'll use PHP's `require()` function instead of the more easy-going `include()` function. Simply put, if the file defined by the `require()` function cannot be found, your entire script is halted, whereas the `include` function just throws a warning and lets the script continue. Add this line to your `index.php` file:

```
require("functions.php");
```

To create a new function in the `functions.php` file:

```
function getimdb_choice($searchfor) {
```

```
}

```

In this function, you'll:

1. Pass in the search string
2. Retrieve the available imdb.com id values (using the TRYNT API) that match your string
3. Return the available options as an array

To get the result array, inside the `if (search=='y')` control structure in the `index.php` page you create a new array and call this function, passing the search string as an argument to the function:

```
$imdbcodes = array();
$imdbcodes = getimdb_choice($string);

```

Here is the PHP code for the function:

```
function getimdb_choice($searchfor) {
    $ids = $codes = $titles = $years = array();
    $url = "http://www.trynt.com/movie-imdb-api/v2/?t=";
    $url .= urlencode($searchfor) . "&fo=xml&f=0";
    $result = file_get_contents("$url");
    $pattern = "/\<id\>(.*?)\</id\>/";
    preg_match_all($pattern,$result,$ids,PREG_SET_ORDER);
    $no_ids = count($ids);
    if ($no_ids < 5 ) { $no_ids = 5; } # Return at most 5 results
    # Get the ttxxxxxxx codes from the API return
    for($i=0; $i<$no_ids; $i++) {
        $tt = $ids[$i][1];
        if (preg_match('/tt\d{1,7}/',$tt)) {
            array_push($codes,$tt);
        }
    }
    # Get the matching titles from the API return
    $mainarray = array();
    foreach($codes as $one) {
        $thisxml = getimdb_xml($one);
        $pattern1 = "/\<title\>(.*?)\</title\>/";
        preg_match($pattern1,$thisxml,$titles);
        $pattern2 = "/\<year\>(.*?)\</year\>/";
        preg_match($pattern2,$thisxml,$years);
        $fulltitle = $titles[1] . " (" . $years[1] . ")";
        $mainarray["$one"] = "$fulltitle";
    }

    asort($mainarray);
    return $mainarray;
}

```

The first line declares and initializes four arrays that you're going to use in the function, while the next three lines create the URL for the TRYNT API and retrieve the XML data. Take the search string and pass it through PHP's `urlencode()` function, to replace spaces and other special characters with their allowed URL equivalents. Then take the resulting complete URL and, using the `file_get_contents()` function, retrieve the XML search data from `imdb.com` (from the API) into a string called `$result`.

The XML that gets returned consists mainly of `<id>` tags that identify the result from `imdb.com` that match the search string, sorted according to relevance. Next, use the `preg_match_all()` function to match all instances of those tags (using a regular expression), and put the results into a new array called `$id`.

Remember, `imdb.com` returned results based on search relevance, so one of the top five results should match your query if the search terms were good enough.

Now comes a bit of a hack in the code because the returned result set can contain literally hundreds of results, and you don't need to dig through all that data. Instead, take the first five `id` values and parse out the `imdb.com` codes from the result set (`imdb.com` uses a generated code that looks like `'ttxxxxxxx'` where `'x'` is a numeric character). These codes then are pushed onto the `$codes` array.

The next step is to declare another array that will hold your final values that will be passed back out of the function, an array called `$mainarray`. Let's step through the top five codes from the result set and dig deeper to retrieve more detailed data.

The first line in this for each control structure:

```
$thisxml = getimdb_xml($one);
```

is a call to another function that you'll define in a moment, called `getimdb_xml()`. This function, like `getimdb_choice()`, is a call to the TRYNT API, except in this case you use a specific code value and it returns the detailed XML from `imdb.com`. For now, let's assume that the `$thisxml` string now contains a complete XML record for your `imdb.com` code choice. Again, use PHP's regular expression function to pull out the title and year information from the XML data. This information is then concatenated, and you can start to populate your `$mainarray` array using the unique `imdb.com` codes as array keys, and the title (with year) as the array values. For example:

```
$mainarray["tt0435625"] = "The Descent (2005)"
```

This form of array is a little more manageable than using an associative array, and is

akin to a "hash" type in Perl.

The final step in the `getimdb_choice` array is to alphabetically sort the `$mainarray` so users will get their search listing returned in a more readable format.

Now that you have one of your major retrieval functions complete, let's revisit the other function, `getimdb_xml()`. This function is very similar to `getimdb_choice` in that it uses the TRYNT API to return XML information about a movie in the imdb.com database. The difference is that you are getting the complete XML record about a specific movie, rather than the id and title information about a search result. The coding for this function is similar. Create a new function in `functions.php` called `getimdb_xml`, passing a string variable that we'll call `$code`.

```
function getimdb_xml($code) {  
}
```

The code of the function is:

```
$url = "http://www.trynt.com/movie-imdb-api/v2/?i=";  
$url .= urlencode($code) . "&fo=xml&f=0";  
$xml = file_get_contents("$url");  
  
#Clean up the XML  
$xml = urldecode($xml);  
$xml = str_replace("Hong Kong>", "Hong-Kong>", $xml);  
$xml = str_replace("New Zealand>", "New-Zealand>", $xml);  
$xml = str_replace("South Korea>", "South-Korea>", $xml);  
$xml = str_replace("West Germany>", "West-Germany>", $xml);  
$xml = str_replace("South Africa>", "South-Africa>", $xml);  
$xml = str_replace("Czech Republic>", "Czech-Republic>", $xml);  
  
$patterns = '/\'/';  
$replaces = '&apos;';  
$xml = preg_replace($patterns, $replaces, $xml);  
  
return $xml;
```

Again, begin with creating a URL that points to the API with your passed `imdb.com` code as the unique element in the API. Then use the `file_get_contents()` function to retrieve that information and pass it back as a string to the `$xml` variable.

However, you have to clean up the returned XML a bit to be compliant. First of all, strip out the HTML specific elements using the `urldecode()` function, and then replace various country names with non-spaced equivalents. Finally, take out any single quotes and apostrophes from the XML so that it won't mess up DB2 when you add the code to an insert statement.

Having completed the clean-up of the XML data, you can pass it back as a string

from the function.

As a final step, do a little house cleaning and take the `$dbconn` database connection code that you put together in Part 1 of this tutorial and move it into the `functions.php` file:

```
function $connect () {
    $database = 'MOVIEDB2';
    $user = 'db2inst1';
    $password = 'mypassw0rd';

    $dbconn = db2_connect($database, $user, $password);
    # This creates a database handle to use for
    # subsequent connections to the database
    if ($dbconn) {
        # if the handle returns true, then we've successfully connected
        echo "Connection succeeded.";
        return ($dbconn);
    } else {
        # if the handle returns false, then something has gone wrong
        echo "Connection failed.";
        return 0;
    }
}
```

In this function, you're pointing to the `MOVIEDB2` database, and the function returns the database handle `$dbconn` if the connection is successful, and false (0) otherwise.

Now that you have all of your movie information available, your principal photography on the project is complete. It's time to pay off the actors and shooting crew, and move into the editing suite to start the post-production process.

---

## Section 4. Post-production

Taking the hours and hours of footage and converting it into a coherent two hour movie that still maintains the director's vision is a challenging task, requiring a skillful film editor and plenty of patience. Now that you have a means to retrieve all the XML information about a movie, you need to edit and organize the results in much the same way to get the right data stored in your database.

Think of populating your movie database in terms of stages. The initial stage is to ask the user for a title to search against. Stage two is to pass that query to `imdb.com` and return a list of possible choices to the user. The user then picks which movie to add to the local library. This choice is then passed to the script for stage three,

retrieving and storing the XML data in the local library.

You've already completed most of the work for stages one and two, but you need to add some code to your `index.php` page to help the script identify which stage to work on. First, add another hidden `<input>` field in the form to identify which stage you're working on. In the `$form` declaration, inside the `<form>` tags, add this line:

```
<input type='hidden' name='stage' value='2' />
```

When the user enters a title to search for, and selects **Search**, the PHP script will know to enter the second stage of the script.

Add this code at the top of the page to pull out the stage value after it is passed back to the PHP page:

```
if (isset($_HTTP_GET_VARS["stage"])) {  
    $stage = $_HTTP_GET_VARS["stage"];  
} else { $stage = ""; }
```

Inside the `if ("search" == y)` structure, add an `if..elseif...else` structure to break out the three stages:

```
if ($search == "y") {  
    # compose and return DB2 query to local DB  
    if ($local) {  
        # Found the movie in the local DB  
    } elseif ($stage == "2") {  
        # Get the choices from imdb.com  
    } elseif ($stage == "3") {  
        # Get the detailed XML from imdb.com  
    }  
}
```

(If the `$local` variable is true, then you found the movie locally -- more on this in the [next section](#).)

In the `$stage == "2"` clause, you want to invoke your `getimdb_choice()` function using the `$string` variable as the search string, and return the result to an array called `$imdbchoices`:

```
} elseif ($stage == "2") {  
    # Get the choices from imdb.com  
    $imdbchoices = getimdb_choice($string);
```

Once you have this array, you create another form for the page, showing the available choices to the user and recording which movie to add, by adding a new string named `$secondform`:

```
$secondform = <<<MULTIFORM
<p>Which movie were you looking to add?</p>
<form name='addchoice' action='$thispage' method='get'>
<input type='hidden' name='sel' value='y' />
<input type='hidden' name='stage' value='3'>
MULTIFORM;
```

Again, make sure that the final `MULTIFORM` portion of the heredoc declaration has no trailing white space. This form has a different name (`addchoice`) to differentiate it from the first form, and also includes a new "stage" value so the choice is passed to the appropriate stage in your `if...elseif...else` structure.

Notice that the form isn't complete. It has no `</form>` tag, because you need to add all the movie choices from your array. To do so, you add the code block:

```
foreach ($imdbchoices as $key=>$value) {
    #Ask the user to pick which movie is the one they are searching for
    $value = html_entity_decode($value);
    $secondform .= "<input type='radio' name='choice' \
        value='$key'>&nbsp;<a href='http://www.imdb.com/title/$key'> \
        $value</a></input><br />";
}
```

This code iterates through the array, taking each title and code and putting that information into a separate radio button choice for the user to pick from. You also have to use the PHP `html_entity_decode()` function to convert the special entities returned in the values to proper text. Finally, complete the form with this piece of code:

```
$secondform .= "<br/><input class='submit' type='Submit' \
    value='Add to Library' /></form>";
```

To wrap up the use of this form, declare it at the top of the page and then echo it at the end of the page:

```
$thispage = $_SERVER["PHP_SELF"];
$secondform = "";

echo $form;
echo $secondform;
echo $tail;
```

If you add a search term in the page, then click **Submit**, the page will return with five (or fewer) movie choices. Each will have a link to the corresponding imdb.com page for the movie for confirming the choice, if desired.

After selecting the appropriate movie and clicking **Add to Library**, you'll see that the URL now has `stage=3&choice=ttxxxxxxx` with `ttxxxxxxx` corresponding to the imdb.com code for that movie.

To retrieve and store the full XML information for that movie, we'll now move to the `elseif ($stage == "3")` clause. First, if you're going to be accessing the database, you need a database handle for the script to use. Let's declare one using your `connect()` function, and put it near the top (just above the `if ("select" == "y")` clause):

```
$dbhandle = connect();
```

And add in a closing call at the bottom (just before the `echo $head` statement):

```
db2_close($dbhandle);
```

Add to your global declarations to retrieve the `$code` variable from the GET array passed back to the page:

```
if (isset($_HTTP_GET_VARS["choice"])) {  
    $choice = $_HTTP_GET_VARS["choice"];  
} else { $choice = ""; }
```

And, to create a text string to hold your results from this third stage:

```
$thirdform = "";
```

Now you can add this code inside the `elseif ($stage == "3")` clause:

```
$imdbxml = getimdb_xml($choice);  
$sql_insert = "INSERT INTO db2inst1.movies (movieid, info) \  
VALUES (default, XMLPARSE(document '$imdbxml'))";  
$db2result = db2_exec($dbhandle, $sql_insert);  
if ($db2result) {  
    $thirdform .= "<h4>Successfully added the title.</h4>";  
} else {  
    $thirdform .= "<h3>Add failed: " . db2_stmt_errormsg() . db2_stmt_error() . "</h3>";  
}
```

The first line calls the `getimdb_xml()` function you defined earlier to call the TRYNT API using the unique `imdb.com` value for the chosen movie, returning the full XML information as a string to the `$imdbxml` variable.

Then you start to get into the fun DB2 parts. The `$sql_insert` variable is a regular SQL INSERT statement, with one exception: the DB2 `XMLPARSE` function is called to parse the `$imdbxml` string as an XML document. This makes sure that what gets passed through to your `info` column in the `movies` table is a well-formed XML document.

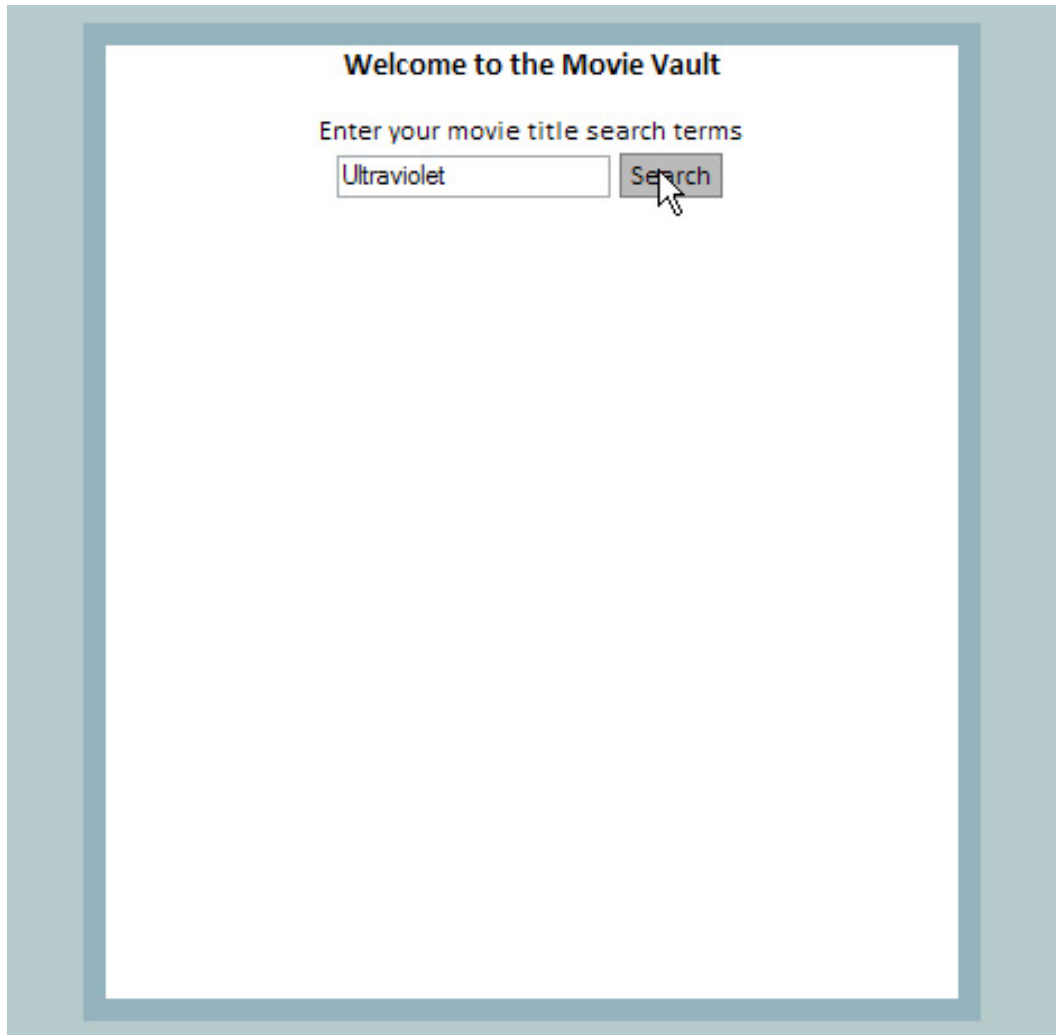
Next you'll create a call to the `db2_exec()` function that is part of the `ibm_db2` PECL extension you set up in Part 1. This prepares and executes your INSERT statement against the `MOVIEDB2` database. If it returns success or failure, you can return that information to the Web page through the `$thirdform` text string. Add an `echo` call to the bottom of the page:

```
echo $secondform;  
echo $thirdform;  
echo $tail;
```

Let's step through the interface now to see how things look, shall we?

1. The default page, with a suitably choice movie to add entered into your text area.

**Figure 4. Step one of adding a movie**



2. The alphabetized return set, showing that each title is a link to the imdb.com page.

**Figure 5. Step two of adding a movie**

**Welcome to the Movie Vault**

Enter your movie title search terms

Which movie were you looking to add?

"Ultraviolet" (1998)

Beyond Ultra Violence: Uneasy Listening by Merzbow (1998)

Starz on the Set: Ultraviolet (2006)

Ultraviolet (1992)

Ultraviolet (2006)

3. Successfully added the title to the database.
- Figure 6. Step three of adding a movie**



To verify, do a quick `db2 select * from db2inst1.movies` using your DB2 CLP. This will spit back a result set with a movieid of 10000, along with an accompanying XML document full of information about your chosen movie, "Ultraviolet." To quote Klaatu in the 1951 classic, "The Day The Earth Stood Still," you do not claim to have achieved perfection, but you have a system, and it works!

You've got the project all wrapped up now, eh? The master cut has been printed and shipped to the theaters, so nothing more to do but put your feet up and let the Oscar nominations roll in, right? Wrong! You've got to get out there and promote, promote, promote! No one's going to see your surefire hit unless you tell them it's a surefire hit. Media blitzes! Headline stars schmoozing on Regis and Kelly! A mention in Letterman's top 10 list! You won't make this a success until you're opening big at the box office.

---

## Section 5. The Premiere

Now that you have your movie information stored in your database, you have to get it out there for everyone to see and marvel at. There are myriad ways to show off this information, but in the interest of brevity, go for the easiest way.

To show off your movie information, query the XML column in the database, extract a few key pieces, and wrap it up in a nice tabular format that can appear on every page.

First off, let's put together the list of what you want to retrieve from the XML. How about:

1. Title
2. Year
3. URL for the imdb.com record
4. imdb.com user rating
5. URL for the cover art image
6. Brief plot description

Armed with those choices, you can start to put together the DB2 query against the XML column. In Part 1 of this tutorial, you queried XML information using the embedded SQL query function `db2-fn:sqlquery`. While it is certainly possible to execute the same sort of query in this case, it's a little easier (and more educational) if you instead explore the idea of using an XPath query to retrieve the information.

This table of existing movie information is going to be called on every page, so let's create it as a new function in the `functions.php` file:

```
function existing($dbh) {
    $table = "";
    $db2array = retrieve_existing($dbh);
    if (!$db2array) {
        $table .= "<h3>Local search failed: " . \
            db2_stmt_errormsg() . db2_stmt_error() . "</h3>";
        return $table;
    } else {
        $table = "<h3>Here are the current movies \
            in your library</h3><table>";
        $row_level = 1;
    }
}
```

```

while (db2_fetch_row($db2array)) {
    $num = bccmod($row_level,2);
    if ($num == 1) {
        $table .= "<tr>";
    }
    $title = db2_result($db2array, 0);
    $year = db2_result($db2array,1);
    $url = db2_result($db2array, 2);
    $rating = db2_result($db2array,3);
    $cover = db2_result($db2array,4);
    $plot = db2_result($db2array,5);
    $table .= "<td><a href='$url'><img src='$cover'></a></td>" .
        "<td><a href='$url'>$title ($year)</a><br />Rating:
$rating</td>" . "<td>$plot</td>";
    if ($num == 0) {
        $table .= "</tr>";
    }
    $row_level++;
}
$table .= "</table>";
return ($table);
}

```

You'll also create another function called `retrieve_existing()` that contains the XML query:

```

function retrieve_existing($dbh) {
    $xml_query = "select t.* from db2inst1.movies m,
xmltable('\$c/trynt/movie-imdb' \
    passing m.info as \"c\" columns TITLE varchar(64) path 'title',
YEAR char(4) \
    path 'year', URL varchar(64) path 'url',
RATING char(4) path 'user-rating', \
    COVER varchar(128) path 'picture-url',
PLOT varchar(512) path 'plot') as t";
    $db2result = db2_exec($dbh,$xml_query);
    return $db2result;
}

```

As you can see, these aren't the simplest functions in the world, but they're not that complicated. They can be broken down into:

- Creating the XML query
- Executing the query and returning the result set
- Checking the result set for error
- Creating the table
- Filling in the table rows
- Returning the table

The key here is the XML query in the `retrieve_existing` function. It's rather long

and nasty-looking, but again, breaking it down into constituent parts makes it easier to understand.

You can see that you're going to be selecting all the columnar data from a table designated as `t`:

```
select t.* from db2inst1.movies m, xmltable( ... ) as t
```

This `t` table is actually created by the `XMLTABLE` function, using the data passed to it from the `m.info` column of the `db2inst1.movies` table (which has been declared as table `m`):

```
... xmltable('\$c/trynt/movie-imdb' passing m.info as "c" ...
```

Now, since you're interested in information inside the `<trynt><movie-imdb>` node of the XML data (you can see this if you query the data using the DB2 CLP to see the parent to child relationship at the top of the XML information), you tell the `XMLTABLE` function to drill into the `/trynt/movie-imdb/` level of the `m.info` column for your data:

```
... xmltable('\$c/trynt/movie-imdb' passing m.info as "c" ...
```

And the `XMLTABLE` function is going to create a pseudo-rational table with columns defined as follows:

```
columns
  TITLE varchar(64) path 'title',
  YEAR char(4) path 'year',
  URL varchar(64) path 'url',
  RATING char(4) path 'user-rating',
  COVER varchar(128) path 'picture-url',
  PLOT varchar(512) path 'plot'
```

Each of the path values correspond to a node inside the `<trynt><movie-imdb>` level in the XML column data.

The result set of this query can be treated just like a regular SQL table query as far as PHP is concerned. After you do a quick check to see that there is actually a result set (returning the error messages back to the main page if there isn't), you can use the result set array to populate your existing movie database.

The `$row_level = 1, $num = bccmod($row_level,2)` and `$row_level++`

declarations and uses are merely a means of splitting the table into two columns. If you find this confusing, or prefer your output showing one movie per row, feel free to ignore these.

```
while (db2_fetch_row($db2result)) {
    $num = bcmmod($row_level,2);
    if ($num == 1) {
        $table .= "<tr>";
    }
    $title = db2_result($db2result, 0);
    $year = db2_result($db2result,1);
    $url = db2_result($db2result, 2);
    $rating = db2_result($db2result,3);
    $cover = db2_result($db2result,4);
    $plot = db2_result($db2result,5);
    $table .= "<td><a href='$url'><img src='$cover'></a></td>" .
        "<td><a href='$url'>$title ($year)</a><br />Rating: $rating</td>" .
        "<td>$plot</td>";
    if ($num == 0) {
        $table .= "</tr>";
    }
    $row_level++;
}
$table .= "</table>";
return ($table);
```

For each row of the result set returned from DB2, you retrieve the title, year, URL, rating, cover, and plot results straight from the XML query, then compose a table data record for each one. When the result set returns empty, close up the table and return the string back to the page that called the function.

Speaking of which, to get this information out on your index.php page, all you need to do now is add this line:

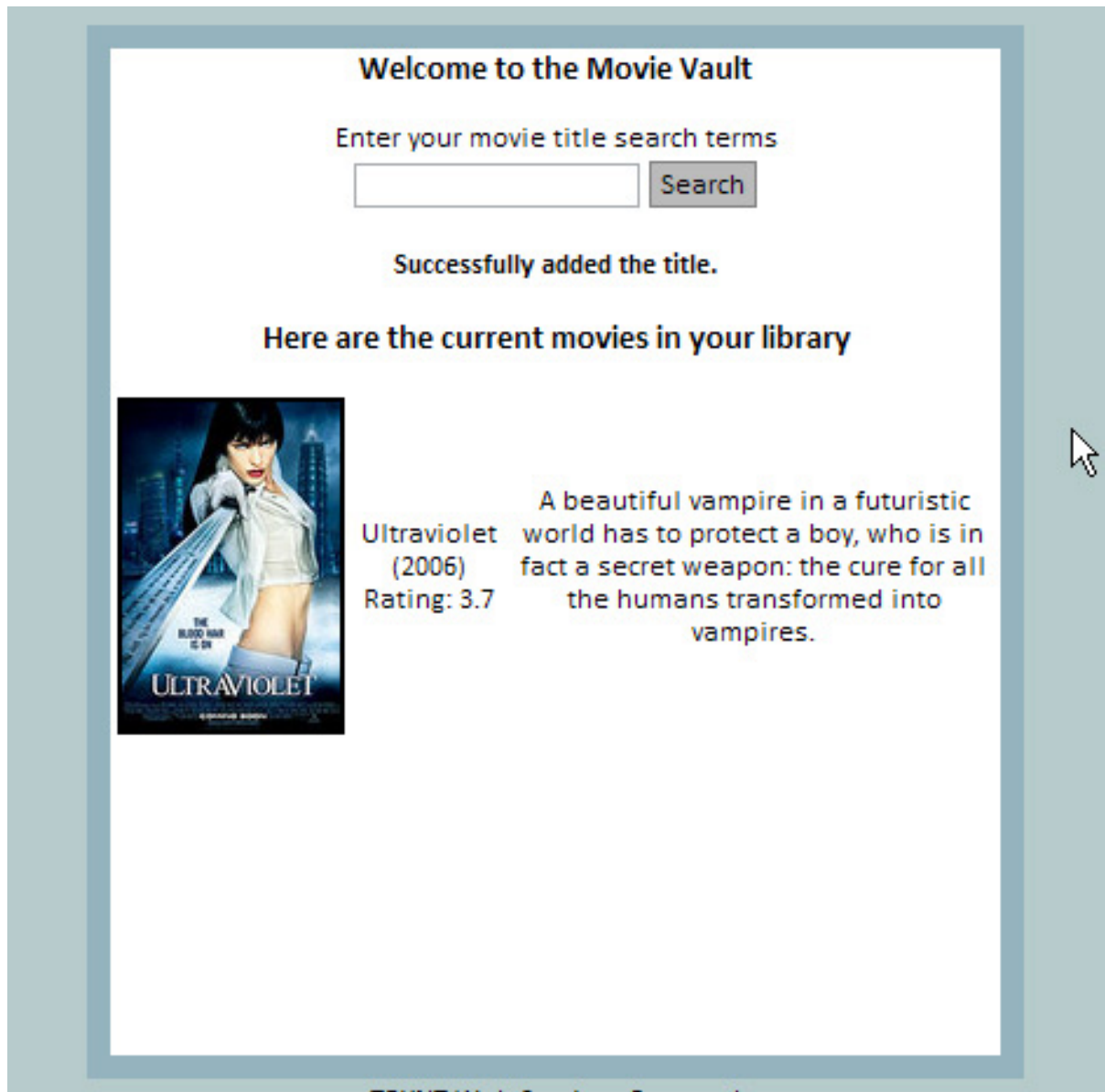
```
$exist_table = existing($dbhandle);
```

anywhere on the page between the `$dbhandle = connect()` and `db2_close($dbhandle)` declarations, then echo that string before the `$tail`:

```
echo $thirdform;
echo $exist_table;
echo $tail;
```

Now if you go back to your base index.php page in your browser, you'll see the movie you added in the last section, as shown in Figure 7. (Don't refresh the one with the GET information in the URL, or you'll add the same movie to your database again.)

### Figure 7. The interface with local movie information



Impressive.

Well, you're almost there. All you need to do now is make sure you're not duplicating your efforts by adding a movie that already exists in your database.

Add another text string called `$check` at the top:

```
$thirdform = "";  
$check = "";
```

When you enter the `if ($search == "y")` clause, you can run this check first:

```

$etitles = array();
$local = 0;
$db2search = retrieve_existing($dbhhandle);
if ($db2search) {
    while (db2_fetch_row($db2search)) {
        $title = db2_result($db2search, 0);
        $title = strtolower($title);
        $url = db2_result($db2search, 2);
        $etitles["$url"] = "$title";
    }
} else {
    $check .= "<h3>Local search failed: " . db2_stmt_errormsg() . db2_stmt_error() . "</h3>";
}

```

This saves the titles and unique URL values stored in an array called `$etitles` that you can then compare against the user's search string to see if it already exists in the database:

```

# Title comparison
$string = strtolower($string);
$local = 0;
# Search the array of existing titles to see if the search string is already in there
$found = array_search($string,$etitles);
if ($found) {
    $winner = $etitles["$found"];
    $winner = ucwords($winner);
    $check .= "<h4>$winner is already in the local database.</h4>";
    $local = 1;
}

```

Add the echo of the `$check` variable after the one for the `$form`:

```

echo $form;
echo $check;
echo $secondform;

```

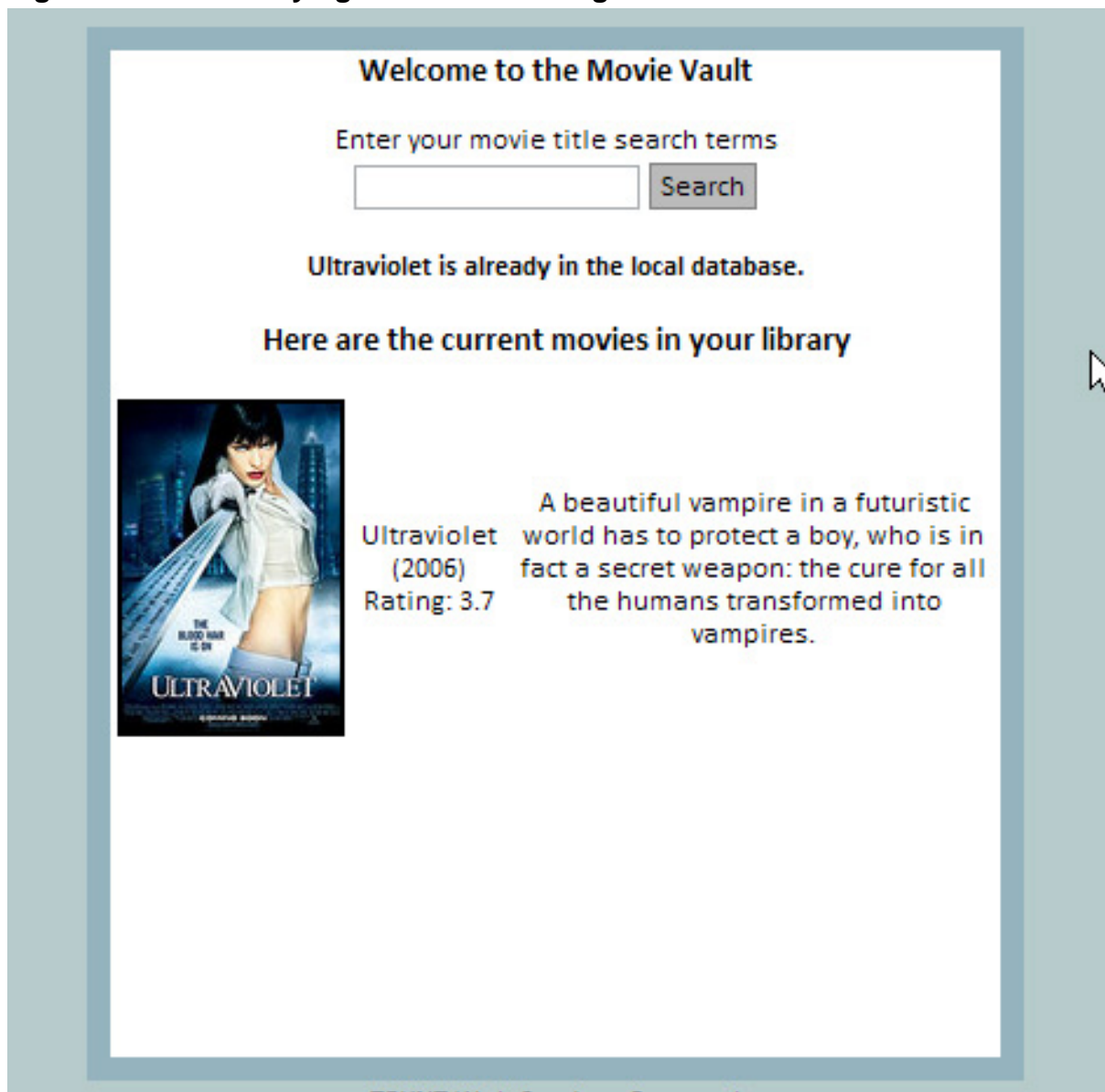
To keep the PHP script from going into the API calls (which are unnecessary if the movie is already in the database), add an additional condition to the `if..elseif` structures:

```

if ($stage == "2" && $local != 1) {
    ...
} elseif ($stage == "3" && $local !=1) {
    ...
}

```

So now, if you enter a search string that matches an existing movie title, rather than doing a full query search the interface returns a simple result, as shown in Figure 8, telling you that the movie is already in the local database.

**Figure 8. Result of trying to add an existing movie**

And that's a wrap for this project! Like any creative endeavor, this application isn't perfect: it could use additional error handling, some more search functions, and there's lots of room for improvement in the look and feel of the graphic interface. But, like Fozzie the Bear said in *The Great Muppet Caper*, "Just let him go. If you hold on too long, he'll just give you warts."

So, until next time, "Th-th-that's all folks!"

## Section 6. Acknowledgements

This article could not have been written without the assistance of the following individuals: Paul Yeung, Salvador Ledezma, Cynthia Saracco, Leon Katsnelson, Rav Ahuja, Ryan Chase, and Raul Chong. And of course my wife, Julie, and son, Ronan, who put up with many late nights of typing.

## Downloads

Description	Name	Size	Download method
Code files for Part 2	part2_code.zip	5KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Learn more about the [DB2 Command Editor](#).
- Find out how to use the [DB2 Command Line Processor](#).
- "Use DB2 native XML with PHP" (developerWorks) offers an in-depth tutorial of XML usage in DB2 9.
- Find [PHP and DB2 setup](#) information from the DB2 Information Center.
- developerWorks [Kickstart program](#).
- developerWorks [Information Management](#).
- Find out more about the new DB2 [XML technologies](#).
- Visit the [Internet Movie Database](#).
- The [TRYNT Heavy Technologies](#) IMDB Web API helped greatly with the IMDB information retrieval.

## Get products and technologies

- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.
- [DB2 Express-C](#), IBM's no charge database server.
- Download [WebSphere Application Server Community Edition](#).
- The [PHP Integration Kit](#) for WebSphere Application Server Community Edition.
- Download the [PHP](#) code base.
- The [DB2 native extensions](#) for PHP.
- The PECL [DB2 extensions](#) for PHP.
- Get the [Eclipse software development kit](#) from eclipse.org.

## Discuss

- [Participate in the discussion forum for this content](#).
- Participate in the [WebSphere Application Server Community Edition](#) forum.

## About the author

Ian D. M. Hakes



Ian Hakes is one of the DB2 Express-C community facilitators bringing the Express-C product to the world. He has worked at IBM since 1999, in a variety of roles: technical writer, developer, team lead, and technical marketing. Ian holds a B.Sc.H. degree in Physics and a B.A. in English, both from Queen's University. In his spare time, Ian enjoys reading and writing novels, and playing road hockey.