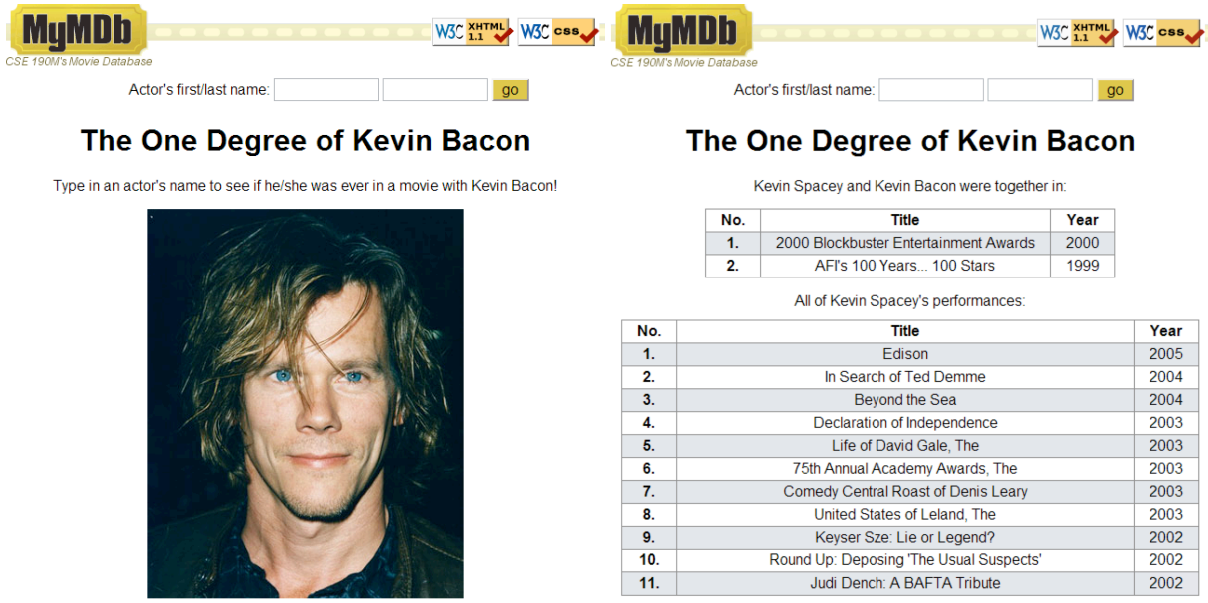
University of Washington, CSE 190 M
Homework Assignment 8: Kevin Bacon

This assignment focuses on interacting with relational databases in PHP using SQL, as well as tying together all the concepts taught throughout this course. You will write the following pages:



Background Information:

The **Six Degrees of Kevin Bacon** is a game based upon the claim that every film actor can be connected to fellow actor Kevin Bacon by a chain of movies no more than 6 in length—or, in other words, that all actors are removed from Kevin Bacon by only six degrees of movie-separation. (In reality, according to *oracleofbacon.org*, 12% of all actors cannot be connected to Kevin at all—but of those who can, 99.98% are within six degrees.)

Your task for this assignment is to write the HTML and PHP code for a web site called **MyMDb** that mimics part of the popular IMDb movie database site. Your site will show two lists of movies: (1) the movies in which a particular actor has appeared with Kevin Bacon, and (2) *all* career movies in which that actor has appeared.

Both the front page **mymdb.php** and the search page **search.php** have a form where the user can type an actor's name. When the form is submitted, results are shown on **search.php**. Turn in the following files:

- **mymdb.php**, the front page
- **bacon.css**, the CSS styles for both pages
- **bacon.js**, the JavaScript code for both pages
- **search.php**, the search results page
- **common.php**, all common code that is shared between pages
- **hw8.zip**, a ZIP archive containing any additional files (optional, if needed)

Since this is the last assignment of the quarter, and one of its goals is to tie together much of the material you have learned, we are not providing you with any skeleton code to get started—you will need write *all* HTML, CSS, JavaScript, PHP, and SQL code for this assignment. You're on your own!

IMPORTANT: To assist with grading, every PHP/HTML page must have a `script` tag that links to the following instructor-provided JavaScript code. (For now this file will be blank.)

- **http://www.cs.washington.edu/education/courses/cse190m/11su/homework/8/provided.js**

Appearance Constraints (both pages):

Your **mymdb.php** and **search.php** must **both** match certain appearance criteria. Both pages must have:

- The same **title**, and links to the **same CSS and JavaScript** resources.
- A **"favorites icon"**. (If you like, you may use the provided **mymdb_icon.gif** from the course web site.)
- A prominently displayed **MyMDb logo**. (If you like, you may use the provided **mymdb.png**.)
- The standard **W3C validator and JSLint buttons**, linked to their corresponding sites.
- A **form** to type an actor's first and last names to search for matching actors in the database.
- A central area of **results** and info. In **mymdb.php** this area contains content of your choice, but it must include at least one image of Kevin Bacon and general text about the site. In **search.php** this area contains a list of the movies in which the actor appeared with Kevin Bacon, as well as a list of all that actor's movies.
- A common **stylistic theme**, and a non-trivial number of styles such as fonts, colors, borders, and layout.

Beyond these constraints (and any other appearance constraints listed in the sections below), all other aspects of the page are left to you to design. We'd prefer for your solution to **not** look exactly like ours — so be creative! Any images other than the ones provided (which you don't have to use) should be uploaded to your Webster space. All images should be linked using absolute URLs, not relative URLs.

With so much HTML code identical between the two pages, it is important to avoid redundancy. You should use the PHP `include` function with a shared common file included by both pages, as you did in HW4 (NerdLuv), to factor out all redundant HTML code, as well as any PHP functions that you call in both PHP files.

Front Page, mymdb.php, and common search form:

The initial page, **mymdb.php**, contains an image of Kevin Bacon and any general information you like. It also has the common search form (which appears on both pages) that allows the user to search for an actor to match with Kevin Bacon. The form must contain two text boxes, one for the first name and one for the last name.

The end goal is to submit this form to **search.php** to show movie results for that actor. However, before the form is submitted you will need to convert the partial text the user entered into a known actor. It is quite possible that the name the user types (such as "Will Smith") will match more than one actor — or that the name will match no one at all in the database. As such we will need a way of figuring out which unique actor in the database (if any) the user is referring to.

To do this you will be making use of an **actorid.php** web service we've set up that will give you search results for whatever (partial or complete) first and last names the user enters. The full behavior of this web service is documented in the next section, but it returns XML containing the full name and unique ID of a matching actor in the database.

Thus when the user clicks the “go” button, you will need to **pause** the form submission while you contact the web service (using an **Ajax request**) to find out the ID of the actor (if any) that matches the first and last name entered. Once you've received information back from **actorid.php** about the name entered, you can then **resume** the form submission, so that **search.php** is sent the unambiguous ID and correct name of the actor to display.

The information flow is roughly the following:

1. **mymdb.php** User types a first and/or last name into the form and clicks the "go" button.
2. **bacon.js** JavaScript code **pauses** the form submission, and initiates an Ajax request to look up the actor ID and full name for the entered terms.
3. **actorid.php** Looks up the actor with that first/last name and outputs them as XML.
4. **bacon.js** Processes the XML that comes back to discover the proper actor ID and full name, and ensures this new data will be passed to **search.php**.
5. **bacon.js** **Resumes** the form submission to **search.php**, now including the ID and full name.
6. **search.php** Shows the movie results for the actor of the provided ID.

Note that you should still use the form itself to convey the additional information to **search.php**, rather than using JavaScript to compose a query string, and then navigating to a new URL. *(Hint: Look at the HTML Forms lecture slides for ways of including information in a form that isn't visible to the user.)*

Pausing/resuming the form submission: If your "go" button is a `submit` button inside a `form` tag, any click on it will attempt to immediately submit the form. You don't want this to happen right away, since your JavaScript code needs to do an intermediate Ajax request to look up the actor's ID before the form can actually be submitted. To "pause" the form submission, you can listen for the form's `submit` event and call `.stop()` on that event to prevent it from happening right away. Then, when you're ready to actually submit the form, you can cause it to be submitted by calling `.submit()` on the DOM object representing the `form` tag.

If the **actorid.php** web service is unable to find anyone matching the provided search terms, your code should inject an error message into the page indicating this to the user, and halt submission of the form altogether.

Using the provided actor ID web service, actorid.php:

parameter name	value
`first_name`	actor's (partial or complete) first name as a string, such as "will"
`last_name`	actor's (partial or complete) last name as a string, such as "shatner"
`imdb`	when set to any truthy value, will cause the service to use the full `imdb` database rather than the `imdb_small` database used for testing

The provided **actorid.php** service provides IDs and full names for (partial or complete) actor names given to it. For example, to search for the ID of an actor with partial names "rad" and "pit":

https://webster.cs.washington.edu/cse190m/homework/hw8/actorid.php?first_name=rad&last_name=pit

This query would return the following XML output, containing the actor ID and full name of the matching actor:

```
<actor id="376249" first_name="Brad" last_name="Pitt" />
```

The service will issue an HTTP error code of `404 Not Found` if no actor is found in the database that matches the given search terms. By default, to guard against inefficient queries bogging down the database server, the web service submits its queries to the smaller `imdb_small` database. Once you are satisfied that your queries are efficient, you should set the `imdb` parameter to any truthy value to switch the service to use the full `imdb` database. **You must submit your finished assignment with this parameter set.**

Movie Search Page, search.php:

The **search.php** page performs two queries on one of the IMDb databases on Webster to show a given actor's movies. Query the database using PHP's `mysql_` functions. Connect to the database using your UWNetID as your user name, and the MySQL password that was emailed to you. (If you lost your password, email us.)

The host to connect to should be `localhost`, since you will be running your PHP code on Webster. The database you connect to should be `imdb_small` while you're developing your program; when you are satisfied that your queries are efficient, you should switch to using the full `imdb` database.

If your **search.php** page accepts any of the following values as parameters, they **must be named as follows**:

- `actor_id` for an actor's ID
- `first_name` for an actor's first name
- `last_name` for an actor's last name
- `full_name` for an actor's full name

The database has the following relevant tables. (The `roles` table connects actors to movies.)

table	columns
`actors`	`id`, `first_name`, `last_name`, `gender`
`movies`	`id`, `name`, `year`
`roles`	`actor_id`, `movie_id`, `role`

Your page displays the following output. For each database query, you must **perform a join** between several database tables.

1. List of movies with this actor and Kevin Bacon: First display a list of all movies in which the given actor performed with Kevin Bacon. These movies should be displayed as an HTML table, with the styling described below. This is the harder of the two queries and **should be done last**, although it is displayed first on the page.

If the actor has not been in any movies with Kevin Bacon, you should not show a table. Instead show a message such as, "Borat Sagdiyev hasn't been in any films with Kevin Bacon."

This query is larger and more difficult because you must link a pair of performances, one by the submitted actor and one by Kevin Bacon, that occurred in the same film. Although you will be using the provided actor's ID, which is being passed as a parameter, you should not directly write Kevin Bacon's actor ID anywhere in your PHP code. For example, don't write a line like:

```
$bacon_id = 22591;   # Kevin Bacon's actor id  (BAD, don't do this)
```

Instead, since Kevin Bacon's name is unique in the database, you must format your query to use his name.

Hint: You must connect two actors (the given actor and Bacon) to their respective roles, and those roles to a movie they both appeared in. Our query joins 5 tables in the `FROM` clause and contains 3 conditions in its `WHERE` clause. (Note that it's possible to eliminate one of the joined tables, but this isn't required.)

2. List of all the actor's movies: Secondly, display a list of *all* movies the given actor appeared in in their entire career. This is the easier of the two queries, and **should be done first**. These movies should also be displayed in an HTML table, as described below. (You may assume that any actor in `actors` has been in at least one movie.)

Hint: Join `actors`, `movies`, and `roles`, then retain rows where IDs from the tables (actor ID, movie ID) match each other and the ID of your actor. Our solution joins 3 tables in the `FROM` clause and has one test in its `WHERE` clause. (Again, it's possible to eliminate one of the joined tables, but not required.)

Table data: The data in both tables should be sorted by year in descending (reverse-chronological) order. Ties should be broken by sorting the movie title in ascending (alphabetical) order. The tables have three columns: A number starting at 1, the title, and the year. The headings of each column must be styled differently in some way (bolded, for example). The rows must have **alternating background colors**, called "zebra striping." (Use PHP to apply styles to alternating rows.)

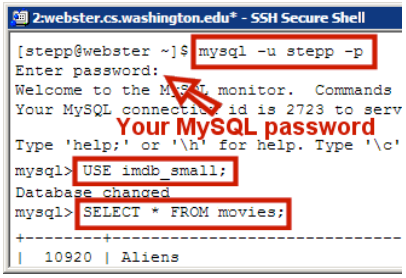
No.	Title	Year
1.	Private War, A	2005
2.	Reefer Madness	2004
3.	Blind Horizon	2004
4.	Churchill: The Hollywood	2004

It is not acceptable to filter your query results using PHP code: you must use SQL to filter the data down to the relevant rows and columns. For example, a very bad algorithm would be to write a query to fetch *all* of the given actor's movies, then another query to fetch *all* of Bacon's movies, and then use PHP to loop over the two looking for matches. Each of the two major operations described above should be done using a single SQL query.

If the user enters Kevin Bacon himself as the actor to match with, the behavior of the page is undefined.

Developing Your Program:

Because the database server is shared by all students, an inefficient query to the full `imdb` database (which is very large) can hurt performance for everyone. To remedy this, we've set up a smaller database `imdb_small` with a much smaller set of movie information for you to use during development. When you're satisfied that your queries are efficient, switch your PHP and JavaScript code to query against the full `imdb`. If you slow down the server for everyone by submitting an inefficient query to the full `imdb` database, we reserve the right add you to a "wall of shame" on the course web site!



Use the **MySQL console** to develop your queries before moving your queries to PHP. For development purposes, some good test actor IDs to use are `376249` (Brad Pitt) and `770247` (Julia Roberts). If your query ever takes too long (more than a second), press **Ctrl-C** to abort it immediately.

When you're ready to move your queries to PHP, test the results returned by the PHP `mysql_` functions to

troubleshoot errors (the functions all return `false` if they fail). If your query is not working properly or giving you the wrong results, be sure to check the *exact text* of the query that's being sent to the MySQL server from your PHP code — it may be different than you think! Often times little SQL syntax errors are introduced when you put your query in PHP, so it helps to see *precisely* the text that's being sent to the database server. *Hint: Save the text of your SQL query in a variable, then print that variable inside* `pre` *tags.*

Implementation and Grading:

Submit your work from the course web site. For reference, our **search.php** contains roughly 95 lines (65 "substantive"); we have roughly 90 lines (60 substantive) of common shared code; our **mymdb.php** is very short, only around 15 lines; our **bacon.js** is roughly 50 lines (35 substantive); and **bacon.css** is 90 lines (57 substantive).

All HTML output sent by your PHP code should pass the W3C XHTML validator. Your CSS code should pass the W3C CSS validator and should avoid redundant or poorly written rules. Your JavaScript code should pass the **JSLint** tool.

Your code should follow **style guidelines** similar to those on our past homework specs. Minimize redundant HTML/PHP code. Use functions and include files to break up code. Avoid global variables. Place descriptive **comments** at the top of each file, each function, and on complex code. Place a comment next to every SQL query you that explains what the query is looking for. Use parameters and return values properly.

For full credit, you must write your JS code using **unobtrusive JavaScript**, so that no JavaScript code, `onclick` handlers, etc. are embedded into the XHTML code. Properly use the XHTML DOM to manipulate page content.

Show proper separation of content, presentation, and behavior between HTML, CSS, and JS/PHP. As much as possible, your JS code should use CSS classes and rules rather than manually setting style properties in the JS.

Format your code similarly to the examples from class. Properly use whitespace and indentation. Use good variable and method names. Avoid lines of code and comments more than 100 characters wide.

Do not place a solution to this assignment on a public web site. Upload your files to the **Webster** server at:

https://webster.cs.washington.edu/*your_uwnetid*/hw8/