Overview: This assignment asks you to build a database-backed Web site based on the simple application you built in Homework 1. The goal is to gain experience in setting up a Web server and programming the interfaces of the Web server, your application code, and the DBMS. Functionally, the Web application in this assignment is very similar to the earlier console application. The Web interface must allow users to invoke all the functions of that assignment (e.g., connect, createTables, findRoom, addFacilities) and view their results. The semantics of the functions remain unchanged. The main difference is that user interaction will use HTTP and HTML instead of the stdin-stdout interface.

Software: We will use *PostgreSQL* as the DBMS, *Apache httpd* as the Web server, and *CGI* as the interface to the Web server, all running on *aturing*. You may use a programming language and libraries of your choice, *but you must check with me first*. All code you use must be *clearly and prominently* acknowledged in your submission (both in the packaged code you submit and on the Web pages of your application produced using your submission). No matter what resources you use, you must be able to explain, in detail, how everything works. If you prefer using different (or additional) software, you may do so, again provided you check with me first. (For example, students in previous semesters have used Tomcat, PHP, JSP, and a host of other software.) The only catch is that you will be responsible for making sure all the software you use works with the submission procedure below, implying your Makefile must result in the installation and configuration of any extra software you need. **Please budget enough time** to learn how to set up a Web server, how to generate dynamic Web pages, and how to use the interface between the Web server and your application process. These tasks are the major components of this assignment.

Web Interface: The entry point to your application is a page that we shall call the *application home page*. It should include the following:

Identifying information: A title (COS 480 Fall 2019 HW02) and your name, preferred email address, aturing login ID, and PostgreSQL login ID. Do **not** include sensitive information such as passwords and student-IDs. The title and your name should anchor links to the class Web page and your personal Web page, respectively. (The contents of your personal Web page are irrelevant for grading as long as it is clearly your own page. If you don't have one, now is a good time to create it. Putting a file `index.html` in the `public_html` subdirectory of your home directory on aturing is a quick way.)

Function links: For each function, there must be a link that points to its *function page*, described below. The link's anchor-text must be the name of the function. The function links must be in the order the functions are listed in the earlier assignment.

All the HTML pages used by your application (e.g., the function pages described below) must include a **link to the application home page** somewhere near the top of the page.

Function Pages: There is one function page corresponding to each of the user functions of the earlier assignment. Each function page should include the following:

Identifying information: This information consists of all the identifying information on the application home page followed by the name of the function (e.g., *addReservation*).

Input fields for function arguments: For each of the function's arguments, there should be one text-input box. These boxes should be stacked vertically and should include descriptive labels. For example, use labels such as *start time*, *end time*, *building*, *room*, and *name of person making reservation* for the arguments of the *addReservation* function.

***Proceed* button:** The above should be followed by a form-submission button labeled *Proceed*. When this button is activated, the appropriate function should be invoked with the provided arguments. The result of the function, as described in the earlier assignment, should be presented as a *result page*, described next.

Result Pages: These pages are used to display the result (output) of a function invocation. Each result page should include the following:

Identifying information: This information consists of all the identifying information on the function page followed by the arguments used in the function invocation. For example, for the function *findRoom* invoked with argument *Neville*, the page should indicate *findRoom(Neville)*. You are free to use an alternate formatting of this information (e.g., HTML tables), especially for longer, string-valued arguments. However, the information should be easy to discern.

Database Results: The output for each function should be in a tabular format reminiscent of the output format described in the earlier earlier assignment. Instead of using separator strings (space-colon-space) and newlines as column and record separators, you should use HTML tables with descriptive column headings. For example, the result page for a function invocation *findRoom(Neville)* should contain an HTML table with three columns, and as many rows as there are result tuples (one for each matching room). In each row, the columns should list, in order, the room's identifier, the building name, and room number. Suitable column headings are *ID*, *building*, and *room*, respectively. The output should be sorted as described in the earlier assignment.

Wherever a room identifier is listed (e.g., in the first column of the output of *findRoom*), that identifier should **anchor a link** leading to the details of the corresponding room. That is, clicking on that link should result in the invocation of the *describeRoom* function with that name as argument, in turn resulting in the display of the corresponding results page. Further, wherever a building name is listed (e.g., in the second column of the output of *findRoom*), it should **anchor a link** leading to the invocation of *findRoom* with that building name as argument. That is, clicking on the link anchored at the building name should result in the invocation of the *findRoom* function with that argument, in turn resulting in the display of the corresponding results page.

Similarly, wherever identifiers of facilities and reservations are displayed in the output, they should anchor links to the result pages obtained by invoking the functions *describeFacilities* and *describeReservations*, respectively, with those identifiers as arguments.

Diagnostics: An area below the table should be reserved for diagnostic messages. You are encouraged to catch as many errors as possible. At the very least, you must catch errors arising from missing inputs, invalid inputs (e.g., a non-number in a field that requires numeric input), failed authentication (invalid user-name or password), and server unavailability. You must also produce an informative message when there is no data matching the search conditions (e.g., "there are no rooms matching Nevillle" when no items match an invocation of *findRoom*). While you are not required to provide thorough error recovery in this assignment (meaning you can earn the full score by catching only the above errors), you are strongly encouraged to do so because you will need something similar when you implement your team project.

HTML Design and Validation: You are encouraged to create interesting designs for your Web pages subject to the above constraints. However, please make sure that all the relevant information is easily legible. Do *not* use (client-side) active features such as Java applets and Javascript. Further, all HTML should be valid, as tested using the W3C service at `http://validator.w3.org/`.

Extra Credit: For extra credit, implement the following feature: At the bottom of each HTML page generated by your application, there must be a link labeled *validate*. Following that link should present the output generated by the W3C validator with that page as input. Make sure your Web pages generate no errors and no warnings.

Submission: You should use a packaging and submission similar to Homework 1, replacing `hw01` with `hw02` where appropriate. Unpacking your submission should create a directory *M*`-hw02` (as a subdirectory of the working directory), where *M* is your last name. Typing `make` in the *M*`-hw02` directory should result in the complete compilation of your program, producing two final executable files (machine code, shell script, Perl script, etc.): `startserver` and `stopserver`. Obviously, you will need to include a Makefile. You should also include a short README file describing the files in your submission; it should also include instructions for compiling and running your program. Please test very carefully that this unpacking and compilation procedure works with your submission. Your score will suffer greatly if it does not, or if your submission contains object files or machine code.[1]

The program startserver will be invoked with a port number (e.g., 4159) as the single command-line argument. This action should result in everything required to set up and start your application's Web server. The output of startserver should be a single line containing the URL of your application (e.g., `http://aturing.umcs.maine.edu:4159/`. The program

[1] As a general guide, your submission should only contain files whose contents you created "by hand." Anything produced as the output of a compiler, interpreter, or similar tool, should not be included; instead, the source file should be included along with proper Makefile commands to generate the desired target files.

stopserver will be invoked with no arguments. It must result in the termination of all the HTTP server processes used by your application. Be careful not to assume that your application will be the only one running on the test machine. If the port number supplied to startserver is unavailable, it should print out an appropriate error message and quit gracefully. (A better strategy would be for startserver to find and use an available port in the vicinity of the given one; you are encouraged, but not required to implement this feature for this assignment.)

Cautionary note on running HTTP servers: Please be very, very careful about how you run Apache (or another HTTP server) when you are debugging and testing your program. It is very easy to overload a machine with runaway `httpd` processes. Make sure you read, understand, edit, and use a `httpd.conf` file that does not overload the system. Pay special attention to the `MinSpareServers` and `MaxSpareServers` directives. Make sure you understand how to shut down the `httpd` processes by killing the proper (parent) process. Check your processes frequently using `ps -u yourusername`, especially before you log out (when you should kill all remaining httpd processes). You should read the manual pages for at least the `ps`, `top`, and `kill` commands. You should be very careful in this regard, mainly to be nice to other users but, as additional motivation, if you are found to be reckless, you will lose a lot of points.

Budget enough time for learning: Students often find this assignment a lot harder than it first appears. *Many of the difficulties will not become apparent until you start working.* Please start very early and expect to learn a lot of new material. If you are not very familiar with Unix software and conventions (e.g., if you had trouble with environment variables or shell scripts), you should budget even more time.