Project A: Extending Red, White, and Blue

In this first project, you will work on understanding the implementation of a small map-based web application, Red, White, and Blue, which we will refer to as RWB, and extending it to add additional functionality.

The project can be done individually or in groups of two or three. The goal here is for you to understand, top-to-bottom how a database-backed web application works.

Before you start

Read the handout "Using Databases in the Web Of Things Environment". This will explain how to log in to the server, how to configure your environment, and how to access oracle using SQL*Plus. Make sure that your environment is working correctly.

Getting and installing RWB

To install RWB, log into your account on the server and do the following:

```
cd ~/www
tar xvfz ~pdinda/339/HANDOUT/rwb/rwb.tgz
cd rwb
more README
```

The README file will give you detailed instructions on how to configure RWB and verify that it is working. You should be able to visit your RWB instance via `https://murphy.wot.eecs.northwestern.edu/~you/rwb/rwb.pl`. It is important to use https - Google Maps will not work correctly without it.

Note that you will need to change the `rwb.pl` code to reflect your Oracle account, and your Google Maps key. If you don't have a relevant Google Cloud account to get a Google Maps key, you can easily create one and the default account credit should be more than sufficient for this lab. You will probably want to restrict the use of your key to our host machine.

Once you have it installed, what you will see is a map view centered on your current location, which is indicated with an additional marker. The map will update as you move, and you can zoom and scroll the map manually as well. In addition to the marker signifying you, you will also see additional markers. These represent the locations of political committees (political parties, election committees, PACs, SuperPACs, etc, for example) that lawfully spend money to affect elections.

By default, you are logged into the application as an anonymous user, and can only view the map and log in as a registered user. A registered user typically has more functions, which appear below the map. As initially configured, the only registered user is the "root" user, which has all functions available, including the ability to register new users, and to grant and revoke their permissions.

What does RWB do?

The goal of RWB is to provide a user with a view of the political activity in his/her vicinity that affects elections for national offices, a view that combines the following sources of information.

- Federal Election Commission (FEC) data. The FEC is responsible for implementing federal campaign finance laws that apply to all entities involved with fundraising, spending, advertising, etc, in elections for national offices. As part of this effort, the FEC makes available disclosures, for every election cycle, of committees, candidates, and individual contributors. We have imported this data into an Oracle database for use with RWB, and other parts of the class The database currently covers election cycles (1980-2018) and contains over 96,000 candidate records, over 218,000 committee records, millions of committee-to-candidate transactions, and 10s of millions of individual contributions.
- Geolocation data. The FEC identifies the addresses of the entities it tracks. Where possible, we have used an open source geolocation database to translate these addresses into map coordinates, specifically latitude and longitude. The geolocation database is also available to you.
- Crowdsourced data. RWB users will be able to provide additional information, outside of the scope of the FEC data, such as an opinion of the politics of a location, and (maybe) improved geolocation information for FEC entities. Such data will be incorporated into the view that other RWB users see.

Out-of-the-box RWB simply shows the FEC committee data for the current election cycle on the map. Over the course of the project, you will extend it to do the following:

- Include candidate, individual, and opinion data in the map view alongside the committee data. The user will be able to select what information to show.
- Aggregate the FEC information to show the total political funds, and funds by party, for the current map view. This information will be shown as both dollar amounts and as a color, the "color of the money", ranging from red to blue.
- Aggregate the opinion information to an overall opinion of the "color of the opinion" of the current map view, also ranging from red to blue.
- Have the ability for existing registered users to invite new users.
- Collect opinion data from registered users.

How does RWB work?

The entire state of RWB exists in the Oracle database. There are two parts to the schema. The FEC and geolocation data are kept in a read-only database in the CS339 account, while the parts of the schema that need to be written by students (users, opinions, user-supplied geolocations, etc) are kept in their own accounts. The file `rwb.sql` describes the non-shared application data. This is where you may make changes.

The shared FEC data model is described in `~pdinda/339/HANDOUT/rwb/fec`. Note that there is a lot of complexity to this, but for the most part, the example code in RWB illustrates the relevant tables and columns for this project. Also, the tables we use for the shared FEC data follow the data schemas given on the FEC web site as closely as possible (http://www.fec.gov/finance/disclosure/ftpdet.shtml). They are described there

as "data dictionaries". The corresponding SQL for these tables is in `~pdinda/339/HANDOUT/rwb/fec/ora`. There are two primary changes that we have made. First, the information about a given class of entities (e.g., committees) from all election cycles is kept in the same table, and a new column indicates the specific election cycle for a record. For example, the `committee_master` table contains committee records from all cycles. If you want to find committee records for a specific cycle, you need to query the table specifically for that cycle. The second change is that there are additional tables that we have created that map specific entities to their geographical coordinates. For example, the `cmte_id_to_geo` table, maps from committee records in the `committee_master` table to their latitude and longitude. Similarly, the `cand_id_to_geo` table, maps from candidate records in the `cand_master` table to their latitude and longitude. These mappings are reasonably accurate since the FEC provides detailed address data. The `ind_to_geo` table maps from individual contribution records in the `individual` table to their latitude and longitude, but these are very inaccurate since the FEC only provides zip code information (at best). These geolocation tables are defined in `~pdinda/339/HANDOUT/rwb/geo`. Note that both the `fec` and the `geo` directories contain scripts and other tools that we use to import and construct data. You only need to consider the SQL files.

Outside of the shared FEC and geolocation data, the state of RWB consists of the following tables, which are defined in the `rwb.sql` file. The `rwb_users` table contains the username, password[1], and email address of each RWB user, as well as the name of the user who referred him. The `rwb_actions` table contains the names of all the possible actions that a user may take in RWB. The `rwb_permissions` table maps from RWB usernames to the actions that they have permission to take. An action should only be taken if the action exists in the `rwb_actions` table, the user exists in the `rwb_users` table, and the user has permission for the action in the `rwb_permissions` table. By default, the following actions are made available:

- query-fec-data – can query the FEC data
- query-opinion-data – can query the crowdsourced opinion data
- query-cs-ind-data – can query the crowdsourced individual geolocation data
- give-opinion-data – can provide opinion data
- give-cs-ind-data – can provide individual geolocation data
- invite-users – can invite a new user to the system
- add-users – can explicitly add a new user to the system
- manage-users – can add/remove users and their permissions.

Two users are added by default: root, with password rootroot, who can do anything, and anon, with password anonanon, who can only query the FEC and opinion data. A typical user that is registered or invited by root will be able to query all data, give opinion data, and invite or add new users. The actions that an invited or added user can take is a subset of the actions the inviting or adding user can take. Notice that users form a tree, with each user pointing to the user who invited or added him, and having a subset of that user's abilities.

[1] The password is stored in clear text in the database, which is highly insecure. This is for pedagogical purposes only – a real system should not do this.

The `rwb_opinions` table provides a mapping from a geolocation and user to a "color" represented by a number in the range -1 (red) through 0 (white) to +1 (blue). The purpose of this table is to provide a starting point for the crowdsourced opinion component of the project. This is a starting point. You are welcome to change the representation as you choose.

The `rwb_cs_ind_to_geo` table provides a mapping between an individual contribution and the geolocation of its origin. The idea here is to improve the information provided by FEC for individual contributions through crowdsourcing. A record in the table also points to its requesting user, the submitting user, and third user who has validated the submission. The purpose of this table is to provide a starting point for the crowdsourced individual geolocation extra credit. You can ignore this table if you don't want to do this extra credit.

RWB has a notion of user sessions. Most of the state of a user session exists within the client, namely the HTML and JavaScript data that the user's web browser display and use. However, a portion of the state, namely the login credentials, is shared between the client and the server. This sharing is done in the form of a cookie named `RWBSession`. When users log into RWB they are given a cookie that contains their login name and password in cleartext (this is not secure!) and is set to expire in about one hour. Without a cookie, the user is treated as the "anon" user. When presented with a cookie, RWB uses its contents to authenticate the user and check to see if he has the necessary permissions (authorization) to do what he wants.

RWB is implemented in a "three tier" style. The client or "presentation tier" consists of HTML and JavaScript that is running in the user's web browser. The HTML is dynamically generated by `rwb.pl`, while the relevant JavaScript is in the file `rwb.js`. The RWB client code in `rwb.js` also makes use of the Google Maps API to show and decorate the map. The client code uses the browser's geolocation API to determine its current location and when the location changes, and it registers a callback so that is informed whenever the user scrolls or zooms the map. The client code may fetch more information and manipulate the map based on it. It does so by issuing a request and having the callback dump the results into a hidden portion of the HTML. More complicated user interactions are done with web forms.

The middle tier or "logic tier" consists of `rwb.pl`, a Perl CGI script that is run by the web server. There are basically three kinds of interaction modes between the client and server. First, `rwb.pl` generates the entire HTML and JavaScript for the client. Second, `rwb.pl` handles forms submitted by the client. Finally, `rwb.pl` can return raw data to the client-side JavaScript. The code is designed so that every invocation of `rwb.pl` is essentially handled as an HTML form submission. The script's activity is governed by the cookie, and the act and run parameters. A special `debug` parameter, when set to 1 will add information to the HTML about the parameters, cookies, SQL commands being run, and their outputs.

The third tier, or "data tier" consists of the database, which combines the `rwb.sql` and shared FEC data schemas. The logic tier (`rwb.pl`) speaks to the data tier using Perl's

DBI interface, which is an example of a "dynamic SQL" interface. In response to an invocation, `rwb.pl` will generate and run one or more statements of SQL, and then generate HTML based on their results.

The following is a more detailed description of an RWB session:

- The user visits https://murphy.wot.eecs.northwestern.edu/~you/rwb/rwb.pl
- `rwb.pl` notices there is no cookie, and thus assumes that the user is anonymous. Since no action is given, it also assumes the action is "base", which means to generate the baseline client page. It therefore generates a default HTML page, with embedded references to `rwb.js` and other components.
- The user's browser parses and displays the HTML. As part of this, it executes the associated JavaScript in `rwb.js`.
  - `rwb.js` issues a request for the current location, registering a callback function.
  - Once the location is determined, the callback is executed, and it edits the HTML to add a new map object, centered at that location. It also registers callbacks for any change in the bounds, center, or zoom-level of the map, and a callback for any change in position of the user.
- Once the map is created, the bounds callback fires. The callback function then determines the current map center and bounds, and issues a new request back to the server, invoking `rwb.pl` again, but this time with the "near" action, which will return data about committees within the view. This request includes another callback function.
- `rwb.pl` is invoked again, with the "near" action and the bounds. It then issues a database query to find the relevant committees, and returns them as a very simple HTML document.[2]
- When the document has made its way back to the client, the request callback function (in `rwb.js`) is fired. It takes the data and places it into a hidden division in the HTML, caching it locally. It then parses the data and uses it to add markers to the map for each of the committees.
- Whenever the map view changes or the user's location changes, the callback functions that are fired essentially repeat the previous three steps.

Beyond map interaction, which is mediated by `rwb.js`, the user can also invoke forms. The following description is what happens for a login form:

- If the user presses the login link, `rwb.pl` is invoked again, with the "login" action, and "run" set to zero. As a result, the page `rwb.pl` generates will consist of a login form.
- The user fills out the login form and hits submit.
- `rwb.pl` is invoked with the "act" parameter set to "login", and the "run" parameter set to 1.

[2] This is even simpler than JSON, for those who are familiar with that. The intent of the RWB is to be entirely self-contained except for the maps and the database. A web app framework is **not** used because we want you to see everything between the database and the user.

- `rwb.pl` extracts the "user" and "password" parameters provided by the form, and does a SQL query to see if the combination exists in the database.
- If the combination exists, it creates a cookie with the combination and passes it back to the user's browser. Next, it displays the base page again. On generating the base page, for every action the user could take, it will produce a link. At minimum, this will now include a "logout" link. Logout is handled by expiring the client cookie.

Debug Functionality

If you invoke `rwb.pl` with the debug flag, for example https://murphy.wot.eecs.northwestern.edu/~you/rwb/rwb.pl?debug=1, it will provide a detailed view of the cookies, parameters, and SQL operations that are performed, as well as their results.

Syntax errors in rwb.pl can be annoying, since they will be reported by your browser as internal server errors. If you run `rwb.pl` from the command-line on murphy, it will show any syntax errors in a cleaner way.

Sometimes it is essential to see what is happening on the server side on a script execution. ***We have made the server side logs visible to you for such debugging. The logs are in /var/log/httpd/*, with access_log and error_log being the most important.***

JavaScript (`rwb.js`) and HTML is best debugged in the browser. If you are using Chrome, you can select Tools->Developer Tools from the menu. This will open a set of very useful tools. The "Elements" tab will show you a breakdown of your HTML, as Chrome has parsed it. Any JavaScript or related errors will show up in the Console tab, and if you click on an error, it will show you the specific line of code that's failing. The Sources tab gives you access to the JavaScript Debugger. The Network tab will show you the network requests that are being made. Safari and Firefox have similar tools. We generally use Firefox or Chrome.

Project Steps

In this project, you will extend RWB to provide the following functionality. Each extension is marked with a percentage that is intended to reflect its perceived difficulty level. You may do the extensions in any order, although we believe the order presented will be the easiest. We believe the best approach to building each extension is the following:

1. Read and understand `rwb.sql`, the fec and geo SQL files, `rwb.pl`, and `rwb.js`. You may find it very useful to use the debugging tools described above to help you.
2. Design the SQL statements that are needed
3. Test the SQL statements using SQL*Plus.
4. Embed the SQL statements into Perl functions (see UserAdd(), for example)
5. Write the Perl logic to call the functions based on form parameters
6. Write the JavaScript (if needed) to use the data returned by `rwb.pl`

You should avoid becoming bogged down in the Perl, and, especially, the JavaScript and HTML, parts of the code. As this is a databases course, you will be graded on correctness, not the appearance of the final product. Just ask us for help in person or in the discussion group if you can't figure something out.

Extending queries (40%) : As handed out, only committee data for the map region and the current election cycle is included in the map view. Extend this to include candidate and individual data for a selection of election cycles. The user should be able to select (using checkboxes, for example) which FEC data they want to see (any combination of committee, candidate, and individual) and for which election cycles (any combination of election cycles). In your implementation, all filtering of the data by the selections needs to happen in SQL. That is, the only data that should be transferred from `rwb.pl` to your client is the data that will be shown. The most straightforward way to achieve this is to extend the "near" action with additional parameters and have the code for the action construct the appropriate SQL queries based on those parameters. Also, your implementation must determine the available election cycles dynamically, by querying the database. In this way, your code will work correctly for future election cycles.

This part of the project will teach you, among other things, about basic SQL SELECT FROM WHERE queries, as well as JOINs, since the geolocation and FEC data are in distinct tables. It will also teach you about dynamic query construction (writing code that writes the code of the query).

Crowdsourcing opinions (30%) : Here, you will add the ability for users to extend the database with opinions. As the starting point of implementing this functionality, you will need to add the ability to invite users. Out of the box, RWB really only lets root explicitly add users. If a user has the "invite-users" permission, the application should give him a form in which he gives the email address of a user he'd like the invite. The system should then send mail to that user with a special, one-time-use link they can use to create an account. The created account should have a subset of the permissions of the inviter.

Any user with the "give-opinion-data" permission should see a simple interface where they can assign a political "color" to their current location. This assignment will then be sent back to `rwb.pl`, which will put it into the `rwb_opinion` table. In turn, your query interface should be extended to include these opinions along with the FEC data. Any user with the "query-opinion-data" permission will be able to see the opinion markers on their map.

This part of the project will teach you, among other things, about SQL INSERT, DELETE, and UPDATE, which are used to manipulate the data in the database in a controlled manner.

Aggregated view (30%) : In this part of the project, you will add the display of ***summary statistics*** about the information shown in the map. This summary information needs to be computed by the database – that is, you will further extend the "near" action to generate SQL queries that compute the summary information and return it. We want you to compute the following summaries:

- If the user has "committees" selected, then compute the total amount of money involved by the committees in the current view. The `cs339.comm_to_cand` and `cs339.comm_to_comm` tables contain such information. Color the background of this summary with a color from blue to red based on the difference between contributions to the Democratic and Republican parties.
- If the user has "individuals" selected, then compute a similar display for individual data.
- If the user has "opinions" selected, include the average and standard deviation of the "colors" in the map region. Color the background of this summary from blue to red, as above.

An important issue is how to determine the aggregate view when there are few or no data points in the region specified by the query. In this case, your code should use queries to progressively larger regions, centered around the user, until you find enough data to produce an aggregate view.

It is possible to simply use the database to collect the relevant data, send that to your JavaScript, and then compute the summary on the client side. This may be much easier to do in some cases and is OK as a starting point, but you ultimately need to compute your aggregates in the database using SQL for this part of the project.

The purpose of this part of the project is to get you to think about the data, the summaries, and how to use JOINs and GROUP BY to do the work of computing summaries over data.

Where to go for help

The goal of this project is to learn how a database-backed web application works. Don't fall into the trap of spending lots of time generating pretty HTML, cool JavaScript, or particularly elegant Perl. Get the SQL right first and make the Perl return what you need. We don't want you to get stuck on the details of Perl, CGI, JavaScript, or HTML, so please use the following resources:

- ⇨ Use the built-in debug functionality, logs, and the developer features of your browser, as described earlier.
- ⇨ The discussion group as described on the course web page. Don't forget to help others with problems that you've already overcome.
- ⇨ Office hours. Make sure to use the office hours made available by the instructor and the TAs.
- ⇨ Handouts: we have prepared handouts on using databases in the WOT environment, a high-level introduction to the Perl programming language, and a high-level introduction to the JavaScript programming model.
- ⇨ Web content. You will find many examples of Perl, JavaScript, and SQL programming on the web. We have links to several particularly noteworthy resources on the web site.
- ⇨ Additional examples. You will find additional examples on Perl, SQL, etc, in `~pdinda/339/HANDOUT`

Hand-in

To hand in the project, you will do the following:

- ⇨ Get a copy of your extended RWB up and running in your account on the server.
- ⇨ Email the instructor and the TAs with a copy of all the files that you have modified or added. At minimum, this will be `rwb.pl, rwb.js, rwb.sql`. In the email, also supply a URL for your running app and the root account name and password. Be sure to note your partners' name(s), if you have partners.
- ⇨ We may provide more detailed hand-in instructions later.

Extra Credit (30% Maximum)

You can gain extra credit by trying the following extensions. If you are interested in these, please talk to us first so that we can determine the possible credit.

Geolocating individual contributions through a game with a purpose: The FEC data, for various reasons, only locates individual contributions to the granularity of zip codes. In our data, we have geolocated them to "10 main street, city, state, zip". The result is that all their markers will appear at this location. Improve this by creating a "game with a purpose". The idea here is that a user can request a more accurate geolocation of a particular contribution. These requested geolocations will then be randomly assigned to users in the general vicinity who have the "give-cs-ind-data" permission. These requests will just pop up during their normal use of the application. If two randomly selected users (the submitter and the validator) give roughly the same geolocation data, this data is then considered good, and provided as a more accurate information for any user who has the "query-cs-ind-data" permission.

Security: RWB is horrendously insecure because passwords are stored in cleartext in the database and cookie file, and could be sent in cleartext over the network if the user does not use https. Read about more secure approaches to session management and implement one of them.

iPhone App or Android App. Improve the mobile client experience by writing an app.

Deeper analysis queries. Examine the FEC data model and develop additional queries that characterize the monetary side of electioneering for national offices.

Integrating voting data. Our data does not indicate the specific elections, or who won each election. Find a source of data that does and integrate it into RWB. A starting point might be the Congress's statistical reports on congressional elections from 1920 to the present: https://history.house.gov/Institution/Election-Statistics. An initial version of this functionality would simply allow the user to see election contests in the map view. A more sophisticated version would correlate committee expenditures and election contest outcomes to let a user see how important political contributions are to deciding outcomes within the current map view.

Better geolocation. Many of the entries in the FEC data, particularly for individual contributions, do not have any geolocation data at all, and others have poor data. We use a free public domain geocoding database (geocode.us). Find and recode the FEC data with a better one.

Precomputed or cached summaries. A weakness of the "Aggregated View" part of the project is that whether you compute summaries in the database or on the client, you may be repeating work. This is particularly important for summaries since they are expensive to compute. But in an urban area like Chicago, an app like RWB may have many users making similar or identical summary queries. The FEC data is static (it only changes once per year). We could either precompute the answers to common queries in advance or cache query results for reuse. For extra credit, implement one or both of these ideas.

Persistent database connections. RWB encapsulates its interaction with the database in the ExecSQL function in order to try to make things clearer for the student. Each time ExecSQL is invoked, a new database connection is made. This is very inefficient. A more scalable (but less clear) design would reuse the connection within one invocation of `rwl.pl`, and, even better, across invocations of it. Learn how to use the database interface more efficiently, and then improve `rwb.pl`.

Improved query throttling. Query throttling is important for scaling an application like RWB. The current JavaScript frontend of RWB throttles requests going to the backend in two ways: queries are limited to a geographic bounding box, and queries are rate-limited so that a long chain of queries (say from user moves or scrolling the map) reduces to a small number of queries. Ideally, however, query throttling would be implemented in the logic tier (`rwb.pl`) and in the data tier (Oracle), so that users could not avoid it (and UI programmers could not get it wrong). For extra credit, implement such query throttling.