Free Machine Finder

Free Machine Finder – Draft Design and Implementation Proposal

Description

Provide a way for our students to easily locate free DICE desktops in AT. In addition provide something like the functionality in the student written MAPP demonstration system that helps students find out which labs their friends are working in.

Proposal

 ikely to see some amendments.

Yada Yada Yada

- About
- Contact
- Software

Recent Posts

- string_to_array behaviour with an empty string
- Hold the exact(ish) "CREATE VIEW" definition in the PostgreSQL System Catalog
- Mock REF Reviewer System
- Production Hadoop Service
- Free Machine Finder – Draft Design and Implementation Proposal

Categories

User Interface

We will provide an Android App as the primary user interface. If this is positively accepted we would then look at also providing an iOS equivalent. We will use a REST API so as not to preclude other interfaces, such as desktop web browser and command line, but will not necessarily implement any of these as part of this project.

The App will have a live home screen widget which simply displays the name of the AT lab that currently has the most available free machines. An additional icon will be shown alongside the lab name if there are "friends" logged onto machines in that lab (see details on
the "friends" feature below).

Opening the App will show one main window. This will contain an ordered scrollable list of AT lab names with those at the top of the list having the most available free machines and those at the bottom having the least available. An additional icon will be shown alongside each lab name where there are one or more "friends" logged onto machines in that particular lab.

The status of AT lab machine usage and logged in "friends" presented by the App will be refreshed automatically every minute while the App is running.

There will be a "Refresh" icon on the main App window which can be touched to force a manual refresh.

" icon on the main App window.
open the App "friends" window.

Archives

See details below.

- February 2009

Friends Feature

The “friends” feature is configured in the App “friends” window. At the top of the window will be an on/off “opt-in” slide toggle which will be off by default. Nothing else is shown on the window in this state. When this toggle is **off** (or switched to **off**) the users UUN is not available to other users to search for and invite to be a “friend” (and any previously made associations will then be hidden from those users, see below). Also the “friends” icon will no longer be shown alongside the AT lab name, irrespective of whether previously made associations would normally result in the icon being shown.

When the toggle is slid to **on** the following additional items are shown.

“Enter Friend UUN” which is an entry box where you can enter a UUN. Alongside this is a “Search” icon. Touch this (or touch Return on the onscreen keyboard) to search for the UUN among only those users that have chosen to “opt-in”. If the UUN is not found an error like “UUN not found (invalid) or UUN has not opted-in” will be displayed. If the UUN is found a modal will be shown with the message “UUN (Firstname Lastname) found”. This will have two buttons – “Invite” and “Cancel”. Touching “Invite” will close the modal and add the UUN to the list underneath (see below) and the “Enter Friend UUN” box will be cleared. Touching “Cancel” will close the prompt modal doing nothing

Below the “Enter Friend UUN” entry box will be a list of friends, each shown as “UUN (Firstname)”. For those waiting acceptance alongside is text saying “Invited”. In addition some will be separated at the top of the list which have an “Accept” button alongside. These are requests from other users to this one. Touch the button to confirm.

Swiping any friend item will display a modal with an option to delete the association or cancel. The entry does not have to be a confirmed association, swipe to delete can also reject an invitation or cancel an invitation they have made (in which case an orphaned acceptance would then be discarded on both sides). Once deleted the association will need to be requested again from either side and confirmed. There is no explicit notification on removal for the other user (the corresponding UUN will just drop off their list on the next refresh).

A “friend” association is always a two way pairing, it does not matter whether A invites and B accepts or B invites and A accepts, after this has happened A will have B listed as a friend and B will have A listed as a friend.

Once friend associations exist (which requires invitation AND acceptance) then when any of the associated UUN’s in the list are logged into machines this will be alerted to the user by displaying the “friends” icon alongside the corresponding AT lab name. This is except where a user has subsequently turned off “friends”, in which case their UUN will (silently) not be

although they will still be listed “friends” window) – possibly

with an “offline” status shown.

Security

For ease of use it would be impractical for the user to go through DICE authentication every time they wanted to use the App. So instead, on first opening the App the user will go through some kind of DICE authentication (possibly by just making an authenticated REST/API call although not entirely clear how this will work from a mobile app yet) for which they receive a unique token (random hash value). The token is stored on their phone and the token/UUN mapping is stored in the back end. Subsequently the App just uses this stored token for authentication (by passing it as an argument in every REST/API call).

Compromise by loss of phone is possible obviously, but if we are informed the relevant token/uun pair can be revoked by deletion of the record at the back end requiring the user to re-authenticate in order to use the App. Deletion of the token/uun pair at the back end would also delete the users “opt-in” status (reverting to the default which is **off**) and any existing “friend” associations. Increased security is possible, for example by constructing the hash based on UUN and phone UDID, but not sure that is really necessary here (and may cause some users concern over tracking).

Implementation

Machine Status Collection

student labs will have a cron job very minute. This command will

make an authenticated REST/API call to update the back end PostgreSQL database with the current status. This updates the corresponding machines record with the current console UUN (or empty if no user logged into the console). It need not wake up the machine to run during sleep – the last reported state should in general indicate no console user (unless they left themselves logged on).

An alternative (maybe preferable) to running the status update command every minute would be to just call the command directly from console login and console logout, this could be done through the PAM stack for example. Although it would be necessary to also run the command on machine reboot and GUI restart for example.

Authentication will use the machine hostclient principal. Using an LCFG spanning map and pgluser we will be able to automatically create a mapping role for each host principal on PostgreSQL for each AT lab machine as and when they go in and out of service. We can also use pgluser to grant the role permission to access the “status” table view (below).

On PostgreSQL these roles will use a view that constrains their access to only the record for the machine corresponding to the host principal. This can be trivially achieved since the SESSION_USER will be the “machine name” (derived from host principal) and by using a view with update rules based on SESSION_USER.

- the client will use hostclient/HOSTNAME.inf.ed.ac.uk
- a spanning map and pgluser will automatically maintain a corresponding “HOSTNAME” user in PostgreSQL for each hostclient principal
- similarly pgluser will grant suitable access to the “status” table view for each HOSTNAME role (by assigning the relevant access role to each HOSTNAME user role).
- when the hostclient principal connects the SESSION_USER will be “HOSTNAME”
- a “status” table in PostgreSQL will have two columns – “hostname” and “console_uun”
- a view will on the “status” table that is “select” and “update” constrained to only rows where “hostname” matches SESSION_USER – hence the given “HOSTNAME” role can only see and change the value of the row in the table which has a matching hostname value. A similar “insert” rule will be defined on the view so that only a row with matching hostname value can be added. The REST/API will use an “UPSERT” to insert or update the row it owns as appropriate.
- the “status” table will be unique on hostname to avoid any risk of double entry by accident or design flaw.

Obviously status updates will not work during online examinations since the connection will be firewalled. This is reasonable behaviour (all locked down labs will show as fully available during the exam).

Unfortunately it will probably not be possible at present .ed lab room bookings (for entral timetabling system as

remote programmatic access to that system was de-scoped in the original project. Consequently this service may show rooms having many free machines that are not actually available to students to use as there is a tutorial happening in the room at that time, for example.

Support Data

A feed of data will populate an “active_user” table from the Informatics School Database with known UUN’s and corresponding names. The known UUN’s will only be those that have a valid account entitlement (effectively the same data as provided via the Prometheus user view) . Once a UUN drops off that feed the authenticated REST/API can no longer be used to retrieve a token for that UUN and any existing token/uun pairing and associated data will be marked for deletion and will be fully purged after a short (glitch handling) delay. By using a FDW and TheonCoupler all the feed management can be done entirely in PostgreSQL and no supporting infrastructure should be required.

An additional feed of data will be required to populate a “room” table. This will contain machine to room mapping data. This data could come from LCFG or the inventory. Directly from the inventory would probably be preferable and can be achieved with a simple REST/API query against that and internalized as an FDW. Alternatively it may be possible this information could be extracted from clientreport or LCFG information on the client itself and returned in the

h the console login status.

REST/API

Initial Authentication and Connection. When the App has no token it will require DICE authentication. It can then make an authenticated connection to the REST/API as that UUN.

An authenticated URI to REST/API can request a token for the authenticated UUN (or a call to the authenticated URI will first bounce the user through an authentication step). Once the App has the token it no longer uses the authenticated URI.

All other connections to REST/API use the unauthenticated URI always passing the token as one of the arguments.

All REST/API calls return JSON.

The authenticated API simply returns the corresponding token. To do so it will perform an "UPSERT" operation on the "user" table. This table contains three columns: "uun", "token" and "friend_optin". The upsert will be view limited (for the REST/API credentials) so that it can only set the "uun" value – the token will be generated and set automatically and "friend_optin" will be left as NULL. The authenticated API will use credentials held on the server to perform this operation, but that is all these credentials will be able to do. The table will be unique on "uun", preventing double entry by accident or design flaw. Note that an attempt to re-authenticate (if the user loses their token by deleting the App or getting

te the existing record and re-

The unauthenticated REST/API provides the following calls:

- Get a list of labs and free machines – arguments are “token”. This returns an array of lab objects in order from most free to least free. The app will put the first entry in the homepage widget and display all the entries on the main App window. Each lab object contains the name and a flag indicating if friends are in it. Where the user has not turned on “opt-in” the friends flag will always be false. Where the supplied token does not match a stored token/uun pair the result will always be empty. Where it does the array of lab name objects will be returned. Where it does and the token/uun friend “opt-in” value is true then the relevant “friend” table will be scanned to set the friend flag appropriately. The “friend” table is used for this. It has two “uun” columns (invitor and invitee, although after this point the two are synonymous) and a “status” column which if true indicates the association has been confirmed (the invite has been accepted). When scanned the token/uun is looked for under both columns (i.e. find friends that the user invited and also friends that invited that user). Also has a “timestamp” column set on row creation, used later for automatic expiry of requests. The App makes this API call automatically every minute or when the “refresh” icon is touched.
- Enable friends – arguments are “token”. This is called to opt-in to the “friends” feature. Sets the relevant flag in the “user” table and returns confirmation message.

nts are “token” and “uun”. This match for the passed “uun”

value. A match is only returned if the UUN matches an entry in the “user” table where “status” is true. The returned match object includes the corresponding firstname and lastname from the “active_user” table.

- Invite friend (PUT) – arguments are “token” and “uun”. Adds a record in “friend” table with “uun,friend_uun” (where “uun” is that corresponding to the token and “friend_uun” is the “uun” passed directly as argument) and the default for status boolean (cannot be set via add api) which is NULL (meaning “request”). Only a UUN that would be returned as valid in the “get friend” call above will be added. Only one instance can be added (the table is unique on each uun+friend_uun combination).
- Approve friend request – arguments are “token” and “uun”. Updates the “friend” table by locating the row where “the uun corresponding to the token” = “friend_uun” and “uun” equals the passed “uun” (the requester) and status is NULL and sets status to TRUE.
- Decline friend (either an invite, a request for approval or a confirmed association) – arguments are “token” and “uun” (of friend). Searches “friend” table and deletes the row where uun corresponding to token equals “uun” and “friend_uun” equals passed “uun” or vice versa.
- Get friends – arguments are “token”. Returns data from “friend” table where current uun (corresponding to token) equals “friend_uun” or the “uun” columns. An ordered array of objects is returned suitable for direct display on the “friends” types will be returned:
  atus is NULL and current uun

matched the “uun” column; “invites” where status is NULL and current_uun matched the “friend_uun” column; “associated” where status is TRUE and current uun matches either column. Data included in the object will be the names corresponding to the UUN and a type flag. This API call is made when the “friends” App window is opened, every minute while it is open and immediately after the “enable/invite/approve/decline” requests above.

Note that rows in the “friends” table are deleted automatically by purge function from where status is NULL (invites have not been accepted) after a period of time, e.g. 7 days.

To alleviate tracking concerns for “friends” data we could consider (as in the demonstration application) hashing all UUN values (with a random salt value held only on the server) within PostgreSQL so that casual internal support access (and accidental end user remote access by API bug or deliberate hack) reveals no useful information.

All the API functionality is implemented directly in PostgreSQL using
views and appropriate update rules. The REST/API will connect locally as a specific set user (no connections from outside the server for that user). It would be easily implementable with Python/FlaskRestful for example, although the authenticated side might need more work.

The App simply makes the necessary REST/API URI
:sults, it needs little other logic.
e via Curl calls. It holds no state

other than its connection token.

Implementation Sequence

Below will probably be done in two discreet stages, each running through some or all of the steps below. First would be to implement the core support for finding free machines (all steps). Second would be to implement support for the “friends” feature (steps 8 through 12).

1. Create back end PostgreSQL service.
2. Add LCFG spanning map and pgluser configuration to map hostclient principals to db users.
3. Create REST/API for client “status” table update.
4. Create machine client.
5. Test then deploy onto all student lab machines.
6. Monitor content on PostgreSQL to confirm functionality.
7. Implement and test feed for “active_user” table update.
8. Implement PostgreSQL functionality for REST/API calls including permissions. Test directly in PostgreSQL.
9. Implement a Python/Flask REST/API interface.
10. Test REST/API functionality with Curl.
11. Implement a full test suite using Curl and a simulated data set of some kind.
12. Write the Android App.

17TH JANUARY 2018 LEAVE A COMMENT