Unified Cognitive Assessment Research Platform Developer Manual

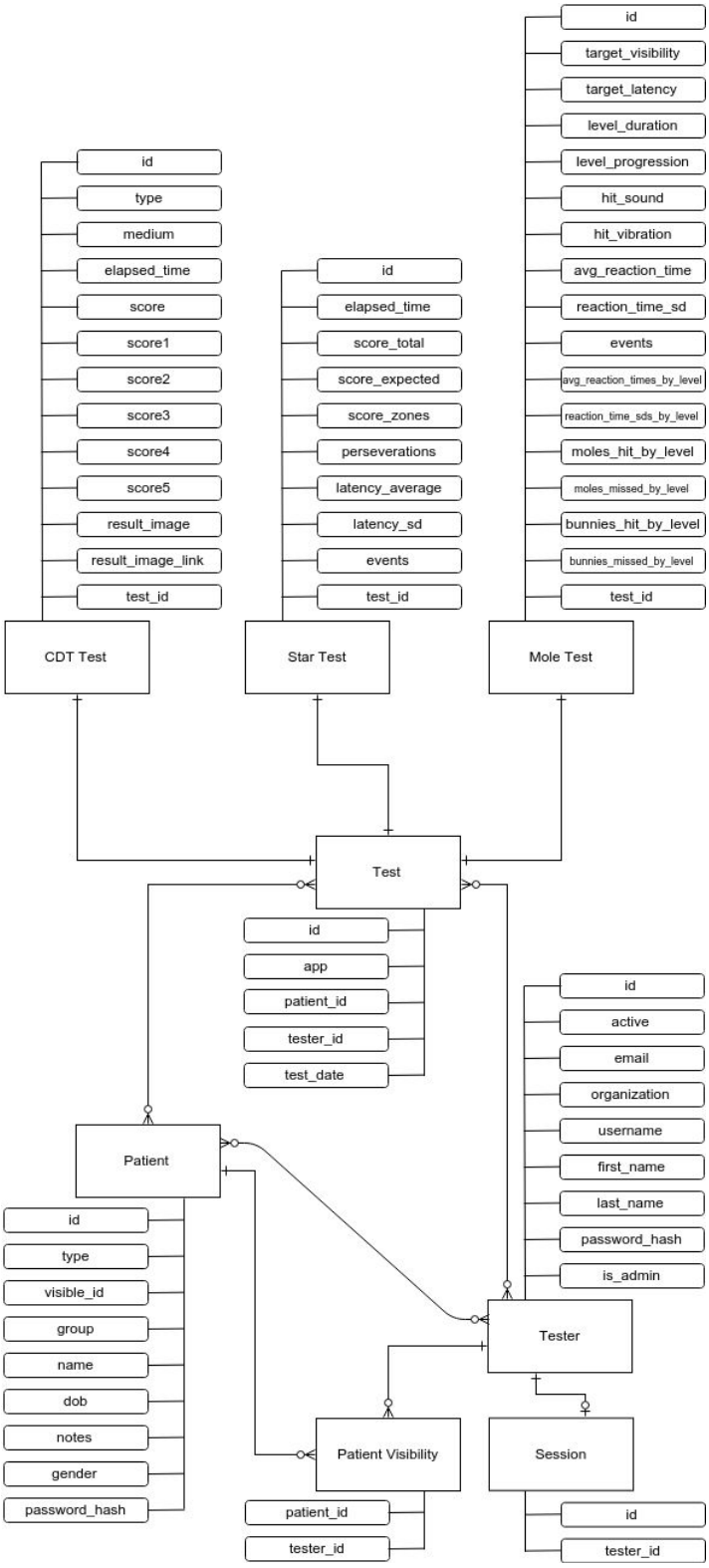
Repository URL: https://bitbucket.org/guana/phydsl-games

Architecture

Overall

Upward arrows indicate data being sent from the user to the system; downward arrows indicate data being retrieved from the system by the user.

Entity Relationships



Back-end Deployment

In this section, the UCAP back-end refers to the API, business logic, database, and all the files needed to set up and maintain the server. Although it's not strictly necessary, it's convenient if the UCAP website lives in the same place as the back-end; for this reason, the UCAP back-end is extended to include all the website files (API-calling controllers, templates, scripts, etc.). In the UCAP repository, the back-end lives in the top-level directories `ucap-backend/` and `server-files/`.

Throughout this section, we'll execute some commands on the back-end server and others on a local machine that contains the UCAP repository. Commands on the server are prefaced with `ubuntu@remote`, and local commands with `me@local`. Unless otherwise specified, execute `ubuntu@remote` commands from the home directory and `me@local` commands from the directory containing the UCAP repository.

Initial deployment

This subsection assumes you have a machine with a fresh Ubuntu 12.04+ installation with username `ubuntu` and an open port 80 to host the back-end. If not, see these instructions for setting up an Ubuntu VM on the Cybera Rapid Access Cloud.

To start, we install some dependencies:

```
ubuntu@remote $ sudo apt-get install apache2 libapache2-mod-wsgi python-dev
python-pip sqlite3
```

`apache2` is the web server, `libapache2-mod-wsgi` provides an interface between the Apache server and Python web applications, `python-pip` is an easy way to install Python packages.

We enable mod-wsgi as follows:

```
ubuntu@remote $ sudo a2enmod wsgi
```

Both the API and the website back-end (that is, the controllers that call the API) are implemented as Flask apps. Before installing Flask, we'll set up a Python virtual environment to keep it isolated from the rest of the system.

First we install virtualenv:

```
ubuntu@remote $ sudo pip install virtualenv
```

Next, we create a directory called `ucap/` in the home directory (with open permissions, since the SQLite database will eventually go here). In `ucap/` we'll make our virtualenv, which must be named `ucap_venv`:

```
ubuntu@remote $ mkdir ucap
ubuntu@remote $ chmod 0777 ucap
ubuntu@remote $ virtualenv ucap/ucap_venv
```

Now we activate the virtual environment, install Flask, and leave the environment:

```
ubuntu@remote $ source ucap/ucap_venv/bin/activate
(ucap_venv) ubuntu@remote $ pip install flask
(ucap_venv) ubuntu@remote $ deactivate
```

Before installing the back-end proper (i.e. `ucap-backend/`), we'll set up an Apache virtual host for it. The file's already been made; we just need to send it to the server and move it to the right directory (we do this in two steps because of permissions issues). We'll also enable the UCAP website and disable the default website.

```
me@local $ scp -i [path to key file] phydsl-games/server_files/ucap.conf
ubuntu@[remote address]:/home/ubuntu/ucap.conf
ubuntu@remote $ sudo mv ucap.conf /etc/apache2/sites-available/ucap.conf
ubuntu@remote $ sudo a2ensite ucap.conf
ubuntu@remote $ sudo a2dissite default
```

Now we can install the back-end. As with `ucap.conf`, it's a two-step process: first we send the `ucap-backend/` directory from our local machine to the server, and then we move it to the right place (`/var/www/`). Each step is encapsulated in a shell script (the second of which belongs on the server). Note that the first time running the third command (`sh update_server.sh`) will throw two "No such file or directory" errors; to fix this, we'll have to create the directory it's complaining about (but only this time).

```
me@local $ sh ucap-backend_to_server.sh
me@local $ scp -i [path to key file] update_server.sh ubuntu@[remote
address]:/home/ubuntu/update_server.sh
ubuntu@remote $ sh update_server.sh
ubuntu@remote $ sudo mkdir /var/www/ucap-backend/ucap/static/test_images
ubuntu@remote $ sudo chmod 0777 /var/www/ucap-backend/ucap/static/test_images
```

`ucap-backend/` includes the required .wsgi file and works right after executing `update_server.sh`. We can now access the website from the browser.

The last step is installing the database. Again, this is a two-step process. On our local machine, we execute a shell script that builds the SQLite database from the schema in `server_files/` and sends it to the right place on the server; on the server, we change the database's permissions.

```
me@local $ sh db_to_server.sh
ubuntu@remote $ chmod 0777 ucap/ucap_database.db
```

That's it! For updating the back-end during development, see "Day-to-day deployment".

Day-to-day deployment

Updates to the back-end fall into two categories: 1) updates to the database schema, and 2) updates to everything else (i.e. to anything in `ucap-backend/`).

After updating the schema, a single command pushes these changes to the database on the server (erasing all data in the database in the process):

```
me@local $ sh db_to_server.sh
```

After updating anything in `ucap-backend/`, deploying is a two-step process (we sent `update_server.sh` to the server in "Initial deployment"):

```
me@local $ sh ucap-backend_to_server.sh
ubuntu@remote $ sh update_server.sh
```

Current Deployment and Tester Activation

The UCAP back-end is currently deployed at 162.246.156.143. When deploying on a different server, be sure to replace all occurrences of 162.246.156.143 in the repository and in `db_to_server.sh` and `ucap-backend_to-server.sh` with the new address.

In the current deployment, new testers are automatically activated on insertion into the database. This happens in the `create_tester` function in `DBManager.py`, which inserts a value of 1 for the `active` field of the `testers` table. On a production server, 0 should be inserted instead of 1. (Email activation hasn't yet been set up, so for a

tester to be activated on a production server, an admin would have to manually change the tester's `active` field from 0 to 1.)

New Game Integration

Back-end

Setting up the back-end infrastructure for a new game is straightforward. First, choose a unique identifier for the game (e.g. “star” for Star Cancellation or “mole” for Whack-a-Mole), henceforth referred to as the “app code”, and create a schema table in `server_files/ucap_schema.sql` called `[app code]_tests`. This table contains fields for whatever should be saved as part of the test results. These fields must be named identically to the keys in the key-value pairs that the games send to the server when saving test results (see “Front-end”). There’s no need to include fields for the app code, patient ID, tester ID, or test date, as these are already included in the generic `tests` table, from which the `[app code]_tests` tables can be thought of as inheriting. The only required field in `[app_code]_tests` is `test_id`, which should have a foreign key referencing the `id` field of `tests`. A primary `id` field is also recommended, though not necessary.

Next, modify the `get_tests` function in DBManager.py to include a case for the new game (beginning with `elif app_code == '[app_code]'`). Simply copy the template provided by any of the existing games in this function and change the fields as necessary.

As a final, optional, step, you can define a post-processing function in `api.py`. When a test result is saved to the UCAP server, the post-processing function manipulates the data as desired before it gets inserted into the database. As the sole argument, pass `test`, which is a dictionary containing all the values that were sent from the game app. The keys are the same as the corresponding fields in the `[app_code]_tests` table. Game-specific fields (that is, all fields other than `app`, `test_date`, `patient_id`, and `tester_id`) are contained in the `test['details']` sub-dictionary. The post-processing function should not return anything. See `cdt_post_process` for an example.

Front-end

The repository contains two Android libraries that can be included in a game to integrate it into UCAP (integrated games will henceforth be referred to as “UCAP games”): ucap-test-setup and simple-async-http.

The most important components of ucap-test-setup are:

- **A title screen (LoginActivity.java).** Here, a tester can log in to a UCAP game with the same credentials used to access the website. There's also an option for a "quick test", which allows a tester to administer the test without logging in. This is the launcher activity for all UCAP games.
- **A patient selection screen (ExistingPatientsActivity.java).** After logging in, the tester is directed here and must choose a patient.
- **A base activity (BaseActivity.java).** When the tester is logged in, any activity that extends this activity shows a menu with a log out option.

In simple-async-http, only two methods will ever usually need to be executed: `HTTPPostAsyncTask` and `HTTPGetAsyncTask`. Call these as follows:

```
httpPostAsyncTask = new HTTPPostAsyncTask(
        activity,
        WebConfig.getURL()+"/api/[method]",
        is_authentication_required,
        jsonParams,
        doneResponse);
httpPostAsyncTask.execute();
```

`activity` is the activity context; the second argument is the url of the API method being called; `is_authentication_required` is a boolean that, if `true`, results in the tester's session ID being passed to the API method; `jsonParams` is a `JSONObject` containing the POST or GET parameters; and `doneResponse` is a class containing the method that gets executed when the server returns a response. `doneResponse` is defined as follows (it can have any name, but `processFinish` must have that name):

```
private AsyncTaskResponse doneResponse = new AsyncTaskResponse(){

        public void processFinish(JSONObject jsonResponse){

                // Do stuff with jsonResponse

        }

 };
```

`jsonResponse` contains the data the API method returned. Note that with simple-async-http's current design, status codes are not accessible.

`HTTPGetAsyncTask` is called in exactly the same way as `HTTPPostAsyncTask`, and takes the same arguments.

To integrate a game into UCAP, follow these steps (when specific files are mentioned, such as the Android Manifest, assume these files belong to the package of the game to be integrated unless otherwise stated):

1. Include the ucap-test-setup and simple-async-http libraries in the game.
2. In the Android Manifest, add the following activities*. Note that LoginActivity must be made the launcher activity.

```
<activity
        android:name="com.example.ucaptestsetup.LoginActivity"
        android:label="@string/app_name"
        android:screenOrientation="portrait" >
        <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
</activity>
<activity
        android:name="com.example.ucaptestsetup.ExistingPatientsActivity"
        android:label="@string/title_activity_existing_patients"
        android:screenOrientation="sensorPortrait" >
</activity>
```

* The `com.example.ucaptestsetup` package name should be changed to something like `edu.ualberta.ssrg.mda.ucaptestsetup` but trying this caused some major merge conflicts that I didn't have time to deal with.

3. In `strings.xml`, define a string called `app_name` with the title of the game. This title will appear in the title screen.

4. In `colors.xml`, define the following colors, which make up the game's color scheme:

```
<color name="ucap_base">#[hex value]</color>
<color name="ucap_base_light">#[hex value]</color>
<color name="ucap_base_dark">#[hex value]</color>
```

5. In the main package (that is, the package given in the `package` attribute in the Android Manifest), create the following two classes:
    a. **`WebConfig.java`.** Use the following template, and set the `url` attribute to the url of the UCAP server.

```
public class WebConfig {

    private static String url = "[url]";

    public static String getURL(){
         return url;
    }

}
```

b. **`TestSetupExitActivity.java` (with the associated xml file).** After either choosing a patient in the patient selection screen or choosing a quick test in the title screen, the tester will land on this activity. For example, in Clock Drawing Test, TestSetupExitActivity contains a prompt to choose between a tablet test and a paper test, whereas in Star Cancellation, TestSetupExitActivity is a loading screen that displays before the game starts. The activity can be anything, as long as it's called TestSetupExitActivity.

ucap-test-setup doesn't yet include a final test result screen, as this screen's layout can vary widely depending on the game. However, the workflow for saving a test result, as well as much of the display logic, are independent of the layout, and so it would be a good idea to implement a Java test result activity without the corresponding xml file in ucap-test-setup. Until then, `TestResultActivity.java` in Clock Drawing Test can serve as a template for test result screens in future games.

After a tester has administered a test, the results must be saved to the UCAP server. In the three existing games, this takes place in the test result activity (`TestResultActivity.java` for Clock Drawing Test and `EndGameActivity.java` in Star Cancellation and Whack-a-Mole). This is where simple-async-http comes in. Simply execute a POST request to /api/create_completed_test as follows (don't forget to define saveTestResponse, where you could, for example, display the test results, so that the tester sees the results only if they were successfully saved remotely):

```
jsonParams = new JSONObject();
try {
       jsonParams.accumulate("app", appCode);
       jsonParams.accumulate("test_date", testRequest.getDate());
       jsonParams.accumulate("patient_id", testRequest.getPatientId());
       // Add test-specific fields here
} catch (JSONException e) {
       Log.d("InputStream", e.getLocalizedMessage());
}
```

```
                httpPostAsyncTask = new HTTPPostAsyncTask(activity,
WebConfig.getURL()+"/api/create_completed_test", true, jsonParams, saveTestResponse);
                httpPostAsyncTask.execute();
```

`testRequest` is an instance of `TestRequest`, which is created in ucap-test-setup. While `patient_id` is set automatically in ucap-test-setup, you must set `app` and `test_date` somewhere in the game (in fact, it might be a good idea to modify ucap-test-setup to set `test_date` instead). `app` is the unique identifier for the game in question (see "Back-end"). Besides the generic fields of `app`, `test_date`, and `patient_id`,* you can add other, game-specific fields; the only requirement is that they have identically-named corresponding fields in the game's test results table (i.e. `[app_code]_tests`; see "Back-end").

*In "Back-end", another generic test result field, `tester_id`, is mentioned; this field is set automatically in the back-end.

Website

For each game, the website displays a table of test results (`tests.html`). Next to each table is a link that exports the table as a CSV file.

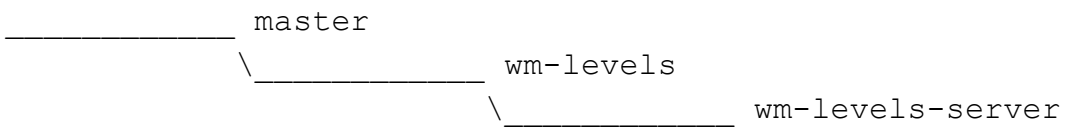
To enable CSV-exporting for a new game, first add two constants at the top of `web_routes.py`: `[app_code]_TEST_KEYS` and `[app_code]_TEST_HEADERS` (these names aren't required, but it keeps with the pattern the other games follow). `[app_code]_TEST_KEYS` is a list of the names of the fields from the `tests` and `[app_code]_tests` tables that should be included in the CSV, and `[app_code]_TEST_HEADERS` is a parallel list indicating the name that should be displayed for each field (see the lists for the existing games as examples). Next, in the CSV-generating part of the `web_get_tests` function (also in `web_routes.py`), add a case for the new game (beginning with `elif request.args.get('app_code') == '[app_code]'`); simply follow the example of Clock Drawing Test and Star Cancellation (Whack-a-Mole has some additional custom processing that can be safely ignored).

Next, the test results need to be displayed in a table on the website itself (`tests.html`). In the `table` HTML element with ID `list`, add a case for the new game (beginning with the Jinja2 line `{% if request.args.get('app_code') == '[app_code]' %}`), in which the table headers and rows are defined. Again, simply follow the example of the other games.

Finally, in `choose_test.html`, add a link to the test results table for the new game, again following the example of the other games.

A Note on the Repository Branches

The repository consists of three branches: master, wm-levels, and wm-levels-server. Each branch begins where the previous one ends:



wm-levels is the same as master, but with a different version of Whack-a-Mole. The main differences are that the wm-levels version introduces levels and configurable game parameters, and also severs its connection with UCAP. wm-levels-server integrates the wm-levels Whack-a-Mole with UCAP.