TrueNTH Shared Services Documentation

Release 18.11.14.dev43+g80e99d0

CIRG, University of Washington

Dec 18, 2018

Contents

CHAPTER 1

API Documentation

Note: Please see Public API documentation for all public and OAuth protected endpoints.

CHAPTER 2

Contents:

2.1 README

2.1.1 true_nth_usa_portal

Movember TrueNTH USA Shared Services

INSTALLATION

Pick any path for installation

```
$ export PROJECT_HOME=~/truenth_ss
```

Prerequisites (done one time)

Install required packages

```
$ sudo apt-get install postgresql python-virtualenv python-dev
$ sudo apt-get install libffi-dev libpq-dev build-essential redis-server
```

Clone the Project

```
$ git clone https://github.com/uwcirg/true_nth_usa_portal.git $PROJECT_HOME
```

Create a Virtual Environment

This critical step enables isolation of the project from system python, making dependency maintenance easier and more stable. It does require that you `activate` the virtual environment before you interact with python or the installer scripts. The virtual environment can be installed anywhere, using the nested 'env' pattern here.

```
$ virtualenv $PROJECT_HOME/env
```

Activate the Virtual Environment

Required to interact with the python installed in this virtual environment. Forgetting this step will result in obvious warnings about missing dependencies. This needs to be done in every shell session that you work from.

```
$ cd $PROJECT_HOME
$ source env/bin/activate
```

Create the Database

To create the postgresql database that backs your Shared Services issue the following commands:

```
$ sudo -u postgres createuser truenth-dev --pwprompt # enter password at prompt
$ sudo -u postgres createdb truenth-dev --owner truenth-dev
```

Building the schema and populating with basic configured values is done via the *flask sync* command. See details below.

Update pip

The default version of pip provided in the virtual environment is often out of date. Best to update first, for optimal results:

```
$ pip install --upgrade pip setuptools
```

CONFIGURE

Create the configuration file

Create a configuration file if one does not already exist

```
$ cp $PROJECT_HOME/instance/application.cfg{.default,}
```

Add Support For 3rd Party Logins

See *OAuth Config*

Install the Latest Package and Dependencies

Instruct `pip` to install the correct version of all dependencies into the virtual environment. This idempotent step can be run anytime to confirm the correct libraries are installed:

```
pip install --requirement requirements.txt
```

COMMAND LINE INTERFACE

A number of built in and custom extensions for command line interaction are available via the click command line interface, several of which are documented below.

To use or view the usage of the available commands:

1. *Activate the Virtual Environment*
2. Set **FLASK_APP** environment variable to point at **manage.py**

```
export FLASK_APP=manage.py
```

3. Issue the `flask --help` or `flask <cmd> --help` commands for more details

```
flask sync --help
```

Note: All `flask` commands mentioned within this document require the first two steps listed above.

Sync Database and Config Files

The idempotent `sync` function takes necessary steps to build tables, upgrade the database schema and run `seed` to populate with static data. Safe to run on existing or brand new databases.

```
flask sync
```

Add User

Especially useful in bootstrapping a new install, a user may be added and blessed with the admin role from the command line. Be sure to use a secure password.

```
flask add-user --email user@server.com --password reDacted! --role admin
```

Password Reset

Users who forget their passwords should be encouraged to use the **forgot password** link from the login page. In rare instances when direct password reset is necessary, an admin may perform the following:

```
flask password-reset --email forgotten_user@server.com --password newPassword --actor
↪<admin's email>
```

Install the Latest Package, Dependencies and Synchronize DB (via script)

To update your Shared Services installation run the `deploy.sh` script (this process wraps together pulling the latest from the repository, the *pip* and *flask sync* commands listed above).

This script will:

- Update the project with the latest code
- Install any dependencies, if necessary
- Perform any database migrations, if necessary
- Seed any new data to the database, if necessary

```
$ cd $PROJECT_HOME
$ ./bin/deploy.sh
```

To see all available options run:

```
$ ./bin/deploy.sh -h
```

Run the Shared Services Server

To run the flask development server, run the below command from an activated virtual environment

```
$ flask run
```

By default the flask dev server will run without the debugger and listen on port 5000 of localhost. To override these defaults, call `flask run` as follows

```
$ FLASK_DEBUG=1 flask run --port 5001 --host 0.0.0.0
```

Run the Celery Worker

```
$ celery worker --app portal.celery_worker.celery --loglevel=info
```

Alternatively, install an init script and configure. See Daemonizing Celery

Should the need ever arise to purge the queue of jobs, run the following **destructive** command. All tasks should be idempotent by design, so doing this is suggested, especially if the server is struggling.

```
$ celery purge --force --app portal.celery_worker.celery
```

Without running `purge`, celery will resume any unfinished tasks when it restarts

DATABASE

The value of `SQLALCHEMY_DATABASE_URI` defines which database engine and database to use. Alternatively, the following environment variables may be used (and if defined, will be preferred):

1. `PGDATABASE`
2. `PGUSER`
3. `PGPASSWORD`

4. `PGHOST`

At this time, only PostgreSQL is supported.

Migrations

Thanks to Alembic and Flask-Migrate, database migrations are easily managed and run.

Note: Alembic tracks the current version of the database to determine which migration scripts to apply. After the initial install, stamp the current version for subsequent upgrades to succeed:

```
flask db stamp head
```

Note: The *flask sync* command covers this step automatically.

Upgrade

Anytime a database (might) need an upgrade, run the manage script with the `db upgrade` arguments (or run the *deployment script*)

This is idempotent process, meaning it's safe to run again on a database that already received the upgrade.

```
flask db upgrade
```

Note: The *flask sync* command covers this step automatically.

Schema Changes

Update the python source files containing table definitions (typically classes derived from db.Model) and run the manage script to sniff out the code changes and generate the necessary migration steps:

```
flask db migrate
```

Then execute the upgrade as previously mentioned:

```
flask db upgrade
```

Testing

To run the tests, repeat the `postgres createuser && postgres createdb` commands as above with the values for the {user, password, database} as defined in the `TestConfig` class within `portal\config\config.py`

All test modules under the `tests` directory can be executed via `py.test` (again from project root with the virtual environment activated)

```
$ py.test
```

Alternatively, run a single modules worth of tests, telling py.test to not suppress standard out (vital for debugging) and to stop on first error:

```
$ py.test tests/test_intervention.py
```

Tox

The test runner Tox is configured to run the portal test suite and test other parts of the build process, each configured as a separate Tox "environment". To run all available environments, execute the following command:

```
$ tox
```

To run a specific tox environment, "docs" or the docgen environment in this case, invoke tox with the `-e` option eg:

```
$ tox -e docs
```

Tox will also run the environment specified by the `TOXENV` environment variable, as configured in the TravisCI integration.

Tox will pass any options after – to the test runner, py.test. To run tests only from a certain module (analogous the above py.test invocation):

```
$ tox -- tests/test_intervention.py
```

Continuous Integration

This project includes integration with the TravisCI continuous integration platform. The full test suite (every Tox virtual environment) is automatically run for the last commit pushed to any branch, and for all pull requests. Results are reported as passing with a and failing with a .

UI/Integration (Selenium) Testing

UI integration/acceptance testing is performed by Selenium and is included in the test suite and continuous integration setup. Specifically, Sauce Labs integration with TravisCI allows Selenium tests to be run with any number of browser/OS combinations and captures video from running tests.

UI tests can also be run locally (after installing `xvfb` and geckodriver) by passing Tox the virtual environment that corresponds to the UI tests (`ui`).

Setup

- `sudo apt-get install xvfb`
- Install geckodriver from https://github.com/mozilla/geckodriver/releases. For example

```
$ wget https://github.com/mozilla/geckodriver/releases/download/v0.21.0/geckodriver-
↪v0.21.0-linux64.tar.gz
$ tar -xvzf geckodriver-v0.21.0-linux64.tar.gz
$ rm geckodriver-v0.21.0-linux64.tar.gz
```

(continues on next page)

(continued from previous page)

```
$ chmod +x geckdriver
$ sudo mv geckodriver /usr/local/bin/
```

Run Tests

```
$ tox -e ui
```

Dependency Management

Project dependencies are hard-coded to specific versions (see `requirements.txt`) known to be compatible with Shared Services to prevent dependency updates from breaking existing code.

If pyup.io integration is enabled the service will create pull requests when individual dependencies are updated, allowing the project to track the latest dependencies. These pull requests should be merged without need for review, assuming they pass continuous integration.

Documentation

Docs are built separately via sphinx. Change to the docs directory and use the contained Makefile to build - then view in browser starting with the `docs/build/html/index.html` file

```
$ cd docs
$ make html
```

POSTGRESQL WINDOWS INSTALLATION GUIDE

Download

Download PostgreSQL via: https://www.postgresql.org/download/windows/

Creating the Database and User

To create the postgresql database, in pgAdmin click "databases" and "create" and enter the desired characteristics of the database, including the owner. To create the user, similarly in pgAdmin, click "login roles" and "create" and enter the desired characteristics of the user. Ensure that it has permission to login.

Configuration

Installing requirements

Ensure that C++ is installed – if not, download from: https://www.microsoft.com/en-us/download/details.aspx?id=44266

Ensure that `setuptools` is up-to-date by running:

```
$ python -m pip install --upgrade pip setuptools
```

Ensure that `ez_setup` is installed by running:

```
$ pip install ez_setup
```

Install requirements by running:

```
$ pip install --requirement requirements.txt
```

Configuration files

In `$PATH\\data\pg_hba.conf` , change the bottom few lines to read:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# IPv4 local connections:

host    all             all             127.0.0.1/32            trust

# IPv6 local connections:

host    all             all             ::1/128                 trust
```

Copy the default configuration file to the named configuration file

```
$ copy $PROJECT_HOME/instance/application.cfg.default $PROJECT_HOME/instance/
↪application.cfg
```

In `application.cfg`, (below), fill in the values for `SQLALCHEMY_DATABASE_URI` for user, password, localhost, portnum, and dbname.

user, password, and dbname were setup earlier in pgAdmin.

portnum can also be found in pgAdmin.

localhost should be 127.0.0.1

```
SQLALCHEMY_DATABASE_URI = 'postgresql://user:password@localhost:portnum/
dbname'
```

Testing

To test that the database is set up correctly, from a virtual environment run:

```
$ python ./bin/testconnection.py
```

2.2 Configuration

TruenNTH Shared Services can be configured in a number of fashions, to support a variety of use cases.

Three primary mechanisms are in place to setup the system as desired:

- *Flask Configuration Files*
- *Site Persistence*
- *AppText*

2.2.1 Flask Configuration Files

Flask configuration files (`.cfg`) are simple python files used to set Flask configuration parameters.

application.cfg

This primary configuration file lives in the *instance* source directory. See README for initial setup of `application.cfg`.

Only values unique to a particular install belong in *application.cfg* including:

1. passwords
2. keys / secrets
3. filesystem paths or local connection details

All others should likely be handled by *Site Persistence*.

Values with defaults are typically defined in the `portal.config.BaseConfig` class. Most are self explanatory or include inline comments for clarification.

Of special note, the one used to control which set of values are pulled in by *Site Persistence*.

PERSISTENCE_DIR:

```
See also Site Persistence, this controls which persistence directory the
`FLASK_APP=manage.py flask sync` command uses to load persistence data
and build the `site.cfg` file.  The value is relative to the
`portal/config` directory.

For TrueNTH:

    PERSISTENCE_DIR='gil'

For ePROMs:

    PERSISTENCE_DIR='eproms'
```

site.cfg

This configuration file also lives in the *instance* source directory, but unlike *application.cfg*, it is managed by *Site Persistence*. It houses the configuration variables used to define the look of the site, such as those use to differentiate *ePROMs* from *TrueNTH*.

A few worthy of special mention for the task of customizing Shared Services.

REQUIRED_CORE_DATA:

```
Set to control what portions of data are considered *required* prior
to allowing the user to transition beyond initial_queries.  Expects
a list, with the following options:

REQUIRED_CORE_DATA = ['name', 'dob', 'role', 'org', 'clinical', 'tou']
```

PORTAL_STYLESHEET:

```
Define which stylesheet to include.  Defaults to 'css/portal.css'

For ePROMs:

PORTAL_STYLESHEET = 'css/eproms.css'
```

To update the `site.cfg` file contents, edit the `site_persistence_file.json` file or use the `FLASK_APP=manage.py flask export-site` command and commit the changed `site_persistence_file.json` to the appropriate repository.

base.cfg

An optional configuration file loaded before *application.cfg*, useful for setting infrastructure-specific defaults.

2.2.2 Site Persistence

In order to handle the migration of **site specific** data, one can generate or import a persistence file, housing details such as:

- business rules defining when interventions should be presented to users
- customization of intervention text
- organizations and clinics on the site

The `portal.SitePersistence` class manages the import and export of the `site.cfg` configuration file as well as a number of database tables holding significant data required for a rich experience. This should never include any patient or personal data, but will include codified business rules and required data to support them.

Database tables included:

- AccessStrategies
- AppText
- CommunicationRequest
- Interventions
- Organizations
- Questionnaires
- QuestionnaireBanks
- ScheduledJobs

Both importing and exporting use the value of `PERSISTENCE_DIR`. Its value is initially looked for as an environment variable, and if not found, the configuration value of 'GIL' is used. (With 'GIL' set, the *gil* configuration directory is used, otherwise, *eproms*).

Export

Site persistence files can be generated in the `PERSISTENCE_DIR`. See above for correct setting. To generate persistence files from current database values, execute:

```
```FLASK_APP=manage.py flask export-site```
```
```

Import

As a final step in the `seed` process, site persistence brings the respective database tables in sync, and generates the *site.cfg* config file:

`` `FLASK_APP=manage.py flask seed` ``

Detailed logging will inform the user of changes made.

Note: It may be wise to back up the existing database prior to running `python manage.py seed` in the unlikely event of unwanted overwrites or deletes.

2.2.3 AppText

To avoid near duplication of templates needing only a few minor string changes, the `portal.models.AppText` class (and its surrogate `apptext` database table), provide a mechanism for customizing individual strings.

In a template, in place of a static string, insert a jinja2 variable string calling the *app_text* function, including the unique name of the string to be customized. For example, in the *portal.templates.layout.html* file, the value of the title string is imported via:

```
<title>{{ app_text('layout title') }}</title>
```

The value for such an AppText can be manually inserted in the database, or added to the site persistence file. Such an entry looks like:

```
{
  "custom_text": "Movember ePROMs",
  "name": "layout title",
  "resourceType": "AppText"
},
```

AppText can also handle positional arguments as well as references to configuration values to fill in dynamic values within a string. The positional arguments are zero indexed, and must be defined when the template is rendered (i.e. JavaScript variables will not be properly defined until the script is evaluated within the browser, and will therefore not work).

For example, given the application has the configuration value `USER_APP_NAME` set to `TrueNTH` and the following:

```
AppText(name='ex', custom_text='Welcome to {config[USER_APP_NAME]}, {0}. {1} {0}')
```

A template including:

```
<p>{{ app_text('ex', 'Bob', 'Goodbye') }}</p>
```

Would render:

```
<p>Welcome to TrueNTH, Bob. Goodbye Bob</p>
```

2.3 Interventions

2.3.1 Roles

Any client can assume the role of an intervention. By doing so, the client becomes the **official** implementation for said role.

2.3.2 Access

Controlling access to interventions deserves special mention. On the `/client/<client_id>` page, the application developer may view and alter the value of **public_accessible**.

Note: With **public_accessible** set, the intervention will always be displayed.

When **public_accessible** is *not* set, two additional options exist for enabling said intervention.

1. To control per user, the service account associated with the intervention should make use of the `/api/intervention/<intervention_name>/` endpoint.

2. Alternatively, any number of *strategy* functions can be added to an intervention, to give access to any subgroup of users as defined by the strategy itself. The available strategies are defined in the `portal.models.intervention_strategies` module, such as the allow_if_not_in_intervention strategy. Use the `/api/intervention/<intervention_name>/access_rule` endpoint to view or modify.

Note: All of the checks above function as a short-circuited **or**. That is, the first check that evaluates as True grants the user access to the intervention. See *combine_strategies* for a workaround.

Note: An optional **rank** setting (unique integer value sorted in ascending order) may be included to control the order of evaluation when multiple strategies are in use. Strategies with a rank value will be evaluated before those without a set rank.

For example, to add a rule that enables the *care_plan* intervention for users registered with the *UCSF* clinic:

```
$ cat data
{"name": "UCSF Patients",
 "function_details": {
   "function": "limit_by_clinic_list",
   "kwargs': [{"name": "org_list",
              "value": ["UCSF",]},]
  }
}

$ curl -H 'Authorization: Bearer <valid-token>' \
  -H 'Content-Type: application/json' -X POST -d @data \
  https://stg.us.truenth.org/api/intervention/care_plan/access_rule
```

Sometimes it is necessary to combine multiple strategies into a logical **AND** operation. To do so, use the `combine_strategies` function, passing the respective set of *strategy_n* and *strategy_n_kwargs* as so:

```
{
  "name": "not in sr AND in clinc uw",
  "function_details": {
    "function": "combine_strategies",
    "name": "not in sr AND in clinc uw",
    "kwargs": [{
      "name": "strategy_1",
      "value": "allow_if_not_in_intervention"
    }, {
      "name": "strategy_1_kwargs",
      "value": [{
        "name": "intervention_name",
        "value": "sexual_recovery"
      }]
    }, {
      "name": "strategy_2",
      "value": "limit_by_clinic_list"
    }, {
      "name": "strategy_2_kwargs",
      "value": [{
        "name": "org_list",
        "value": ["UW Medicine (University of Washington)",]
      }]
    }]
  }
}
```

The full list of strategies used for DECISION_SUPPORT_P3P:

```
{
  "name": "P3P Access Conditions",
  "description": "[strategy_1: (user NOT IN sexual_recovery)] AND [strategy_2 <a␣
↪nested combined strategy>: ((user NOT IN list of clinics (including UCSF)) OR (user␣
↪IN list of clinics including UCSF and UW))] AND [strategy_3: (user has NOT started␣
↪TX)] AND [strategy_4: (user does NOT have PCaMETASTASIZE)]",
  "function_details": {
    "function": "combine_strategies",
    "kwargs": [
      {
        "name": "strategy_1",
        "value": "allow_if_not_in_intervention"
      },
      {
        "name": "strategy_1_kwargs",
        "value": [
          {
            "name": "intervention_name",
            "value": "sexual_recovery"
          }
        ]
      },
      {
        "name": "strategy_2",
        "value": "combine_strategies"
      },
```

(continues on next page)

(continued from previous page)


```
    {
      "name": "strategy_2_kwargs",
      "value": [
        {
          "name": "combinator",
          "value": "any"
        },
        {
          "name": "strategy_1",
          "value": "not_in_clinic_list"
        },
        {
          "name": "strategy_1_kwargs",
          "value": [
            {
              "name": "org_list",
              "value": [
                "UCSF Medical Center"
              ]
            }
          ]
        },
        {
          "name": "strategy_2",
          "value": "limit_by_clinic_list"
        },
        {
          "name": "strategy_2_kwargs",
          "value": [
            {
              "name": "org_list",
              "value": [
                "UW Medicine (University of Washington)",
                "UCSF Medical Center"
              ]
            }
          ]
        }
      ]
    },
    {
      "name": "strategy_3",
      "value": "observation_check"
    },
    {
      "name": "strategy_3_kwargs",
      "value": [
        {
          "name": "display",
          "value": "treatment begun"
        },
        {
          "name": "boolean_value",
          "value": "false"
        }
      ]
    },
```


(continues on next page)

(continued from previous page)

```
      {
        "name": "strategy_4",
        "value": "observation_check"
      },
      {
        "name": "strategy_4_kwargs",
        "value": [
          {
            "name": "display",
            "value": "PCa localized diagnosis"
          },
          {
            "name": "boolean_value",
            "value": "true"
          }
        ]
      }
    ]
  }
```

2.3.3 Communication

Communicate from an intervention to any group of TrueNTH users via the `/api/intervention/<intervention_name>/communicate` endpoint.

The groups API is used to view existing and create new groups. Add existing users via the `/api/user/<user_id>/groups` endpoint.

2.4 Organizations

Organizations are used to name clinics and parent organizations. Use the */api/organization* endpoint to view the list of organizations in the system.

Add new organizations via **POST** to */api/organization* with a JSON document defining the organization compliant with the FHIR Organization resource.

> **Warning:** The parent organization must exist in the system before a child can name it in the **partOf** reference.

To enable use of the */go/<shortcut_alias>* endpoint, to pre-select clinics for new users, an **identifier** must be included in the FHIR resource.

For example, after looking up the correct ID, a PUT of the following document adds a shortcut alias to the *UCSF Urologic Surgical Oncology* organization.

Note: For the shortcut alias to function, the added identifier must have a **system** value of *http://us.truenth.org/identity-codes/shortcut-alias*

PUT to /api/organization/6

```
$ cat data
{
  "resourceType": "Organization",
  "identifier": [
      { "system": "http://us.truenth.org/identity-codes/shortcut-alias",
        "value": "ucsfurology"
      }
  ]
}

$ curl -H 'Authorization: Bearer <valid-token>' \
  -H 'Content-Type: application/json' -X PUT -d @data \
  https://stg.us.truenth.org/api/organization/6
```

Note that organizations now contain a set of 'options' fields, as follows:

- `use_specific_codings` : toggles whether or not the organization should use the subsequent custom options
- `race_codings` : toggles whether or not the organization should capture race information for its users
- `ethnicity_codings` : as above, but for ethnicity information
- `indigenous_codings` : as above, but for indigenous information

For each organization:

- If an org has a True value for `use_specific_codings`, then the r/e/i properties will use the r/e/i options from that org
- If an org has False value for `use_specific_codings`, and it has a parent, then the r/e/i properties will use the r/e/i options from the parent org. Note that this continues recursively, until either it hits either (a) an org with specific codings turned on, or (b) an org with no parent
- If an org has a False value for `use_specific_codings`, and it has NO parent, then it will return true for all r/e/i properties.

These settings are accessible/set-able through the API (via any endpoint that uses the `as_fhir` or `update_from_fhir` methods)

For each user:

- There's a new property on the User model, `org_coding_display_options`. If the user has any orgs, then this property will iterate through all the user's org. For each of the r/e/i options, if any of the orgs' r/e/i properties return true (using the logic presented above), then that user's r/e/i display setting will be set to true (otherwise, it's false).
- If the user has no orgs, these display settings default to true.

When displaying the user profile, each r/e/i section will check the relevant r/e/i display settings for the profile user, and use that to decide whether or not to display the relevant section.

2.5 Timeouts

Session timeouts are handled slightly differently in the browser and on the server hosting Shared Services.

2.5.1 Backend

After authenticating with Shared Services, a cookie is set with an expiration time corresponding to the value of *PERMANENT_SESSION_LIFETIME*, in seconds. If no requests are made in that duration, the cookie and corresponding redis-backed session automatically expire (via TTL). Subsequent requests will be effectively be unauthenticated and force redirection to the login page.

The backend session (the cookie and corresponding redis entry) can be refreshed from the front-end by sending a POST request to `/api/ping` that will modify the current backend session, refreshing the timeout duration (to the value specified by *PERMANENT_SESSION_LIFETIME*).

2.5.2 Frontend

The browser is made aware of the session duration specified by *PERMANENT_SESSION_LIFETIME* and will prompt a user to refresh their session one minute before it expires, but cannot reliably determine the remaining time in the backend session because it may have been refreshed in another tab or browser window.

2.5.3 Intervention

After authenticating with Shared Services, interventions are granted access through a bearer token that expires after a duration set by *OAUTH2_PROVIDER_TOKEN_EXPIRES_IN* (defaults to 4 hours).

Subsequent requests with the same bearer token refresh its expiration each time.

PERMANENT_SESSION_LIFETIME The lifetime of a permanent session, defaults to one hour. Configures session cookie and corresponding redis-backed session. Configuration value provided by Flask.

OAUTH2_PROVIDER_TOKEN_EXPIRES_IN Bearer token expires time, defaults to four hours. Configuration value provided by Flask-OAuthlib.

2.6 Provider Authentication

2.6.1 OAuth Workflow

In order for a user to access authenticated portal pages they first need to login. When logging in through a 3rd party, such as Facebook or Google, the OAuth workflow is used. In this workflow, after the user clicks on the 3rd party's login button they're taken to the 3rd party's login page where they enter their credentials. Upon successful login the 3rd party passes the portal an access token that allows us to fetch information from the third party on the user's behalf which we use to update the user's account and log them in to our system.

Underneath the hood we use Flask-Dance. At a high level, Flask-Dance uses blueprints to authenticate with providers and returns control to our APIs when auth succeeds or fails. The blueprints and APIs are defined in `portal/views/auth.py`. Upon successful authentication the `login()` API is called with the user's access/bearer token which we use to get info about the user. To get this info we create an instance of `FacebookFlaskDanceProvider` or `GoogleFlaskDanceProvider`, which both inherit from `FlaskDanceProvider`, and call `get_user_info`. This function uses the user's access token to send an authenticated request to the provider. When the request returns with the user's information we use create the user an account if they've never logged in before or update an existing account if they've logged in using a different provider, and finally log them in to the current session. All of this logic takes place in `login_user_with_provider()`.

2.6.2 Configuration

In order to authenticate users the portal must know the public and private keys to each 3rd party application. If you haven't already, you'll need to create a third party app and copy its configuration values to `instance/application.cfg` by following the steps below:

Facebook

To enable Facebook OAuth, create a new app on Facebook's App page and copy the `consumer_key` and `consumer_secret` to `application.cfg`:

```
# application.cfg
[...]
FACEBOOK_OAUTH_CLIENT_ID = '<App ID From FB>'
FACEBOOK_OAUTH_CLIENT_SECRET = '<App Secret From FB>'
```

- Set the `Authorized redirect URIs` to exactly match the location of `<scheme>://<hostname>/login/facebook/`
- Set the deauthorize callback. Go to your app, then choose **Products**, then **Facebook Login**, and finally **Settings**. A text field is provided for the Deauthorize Callback URL. Enter `<scheme>://<hostname>/deauthorized`

Google

To enable Google OAuth, create a new app on Google's API page and copy the `consumer_key` and `consumer_secret` to `application.cfg`:

```
# application.cfg
[...]
GOOGLE_OAUTH_CLIENT_ID = '<App ID From Google>'
GOOGLE_OAUTH_CLIENT_SECRET = '<App Secret From Google>'
```

- Under APIs Credentials, select `OAuth 2.0 client ID`
- Set the `Authorized redirect URIs` to exactly match the location of `<scheme>://<hostname>/login/google/`
- Enable the `Google+ API`

activate

In a **non-production** environment add the following to the bottom of `env/bin/activate`:

```
export OAUTHLIB_RELAX_TOKEN_SCOPE=1
export OAUTHLIB_INSECURE_TRANSPORT=1
```

In a **production** environment you should **only** add the following to the bottom of `env/bin/activate`:

```
export OAUTHLIB_RELAX_TOKEN_SCOPE=1
```

Explination

2.6.3 Adding a new provider

To add a new provider you'll need to

1. Create a new blueprint in `portal/views/auth.py` (see the `google_blueprint` and `facebook_blueprint` as examples and use Flask-Dance Documentation as a reference)
2. Update the existing callback API functions `login()` and `provider_oauth_error` to use your new blueprint (see examples from Google and Facebook blueprints in `portal/views/auth.py`)
3. Create a new class in `portal/models/flaskdanceprovider.py` that inherits from FlaskDanceProvider and overrides send_get_user_json_request(to get user info from the provider (see `FacebookFlaskDanceProvider` and `GoogleFlaskDanceProvider` for examples)
4. Import the class created in #3 into `portal/views/auth.py` and create a new instance of it when `login()` is called by the new provider (see how `FacebookFlaskDanceProvider` and `GoogleFlaskDanceProvider` are used in `login()` for reference)

2.7 Sessions

User session data is stored on the server via Flask-Session, specifically in the same redis server used to house the celery tasks.

2.7.1 redis-cli

To view sessions (or other key/values) stored in redis, fire up the command line interface (CLI) and execute simple queries:

```
$ redis-cli
127.0.0.1:6379> keys session*
1) "session:0e17f42c-72d9-49c1-8066-195a1e770ad2"
2) "session:42c94702-f1cb-447d-a998-409dbd5a99b6"
3) "session:e116b0f1-2271-4473-97d0-6d910a4ff582"
4) "session:2483797a-4261-4c6e-a3d0-1d19d6db6446"
5) "session:3ae29547-943c-48e9-bc7e-b44b78c99551"
6) "session:2264efc6-eb5a-46c0-98c6-fb458b435256"
7) "session:1ac11b4b-bafc-41b5-9d93-b6fb90608054"
[...]
127.0.0.1:6379> ttl session:1ac11b4b-bafc-41b5-9d93-b6fb90608054
(integer) 2677441
127.0.0.1:6379> dump session:3e3ff4ed-2848-41e5-b78f-3ea909219d52
"\x00\xc3@\xd4@\xdf\x16(dp1\nS'_fresh'\np2\nI01\ns \x11\x01id \x0e\x003␣
↪\x1b\x1f892f7fec2c15835660cba1324da22125\x17e167e65bbe5de394d486a744␣
↪0\x1007be014719895f627 E\x1f58b1ab0de00d8e2b5bc9bb4e29a7e3c7\x108329d9d2051ec0e84␣
↪\x86\x004@\x91\x03user\x80\x95\x035\nV2@\x11`\xa2\x006\xa0\x0c\t_permanent␣
↪0\x007`\xc6\x01s.\x06\x00\xb2\xbd\xb0W\xf3d\x18\x0c"
```

(continues on next page)

(continued from previous page)

ttl: Time To Live. Once expired, redis will delete the respective session.

dump: The session data is a pickled python dictionary.

2.8 Development

2.8.1 Context

This documentation is oriented towards supporting CHCR implementation of non-authenticated designs and content: mostly front-end. Note that one complexity is that this code base is used for two different systems/configuations (and more will be added): TrueNTH USA, and ePROMs.

2.8.2 System-specific text (app_text)

app_text

2.8.3 System-specific pages

For example, adding a link from the landing page to a "prostate cancer 101" page, but only for TrueNTH (not ePROMs). Guidance: use `SHOW_*` configurations. See this example

2.8.4 Mapping URL's to views

Eg in `views/patients.py`:

```
@patients.route('/patient_profile/<int:patient_id>')
```

2.8.5 Retrieving content from Liferay

Note that one of the systems used for this is AppText Information on managing content in Liferay is here

2.8.6 Use of front-end libs

LESS, jquery, bootstrap, and other

CSS file - for truenth:

- `css/portal.css`
- `less/portal.less`

Note: CSS files are compiled from LESS, and that both the CSS and LESS files are managed in git. Locally, do `less portal/static/less/portal.less portal/static/css/portal.css` Compilation likely to be moved to deploy.sh, at which point we won't need to manage css files in git.

2.9 Internationalization

2.9.1 Indicating Translatable Strings

We use gettext for this within python files; we also use Liferay to manage content in different languages.

Surround all strings with _( ) and it will automatically attempt to find a translation, like:

_('CELLPHONE')

This should automatically be available in any template file.

Note: we are moving to a model where en_US is used as the key here, with no need to use an english .po file.*

For adding new translations, you need to add the blank translation to the .pot file:

<optional comment pointing to where in the code the translation is used>
msgid "Cellphone"
msgstr ""

2.9.2 Updating Translation Files

GNU Gettext translation files consist of a single Portable Object Template file (POT file) and Portable Object (PO file) for each localization (language).

Updating POT files

To update the .pot file with all source strings from the apptext/interventions tables run the following command:

```
$ FLASK_APP=manage.py flask translations
```

Updating PO files

To update the PO files with the latest translations from Smartling, run the following command:

```
$ FLASK_APP=manage.py flask translation-download
```

2.9.3 Initializing Translation Files

You can create a new .pot file with all extracted translations from the code by running the following pybabel command:

```
$ pybabel extract -F instance/babel.cfg -o portal/translations/messages.pot portal/
```

2.9.4 External Documentation

jinja i18n-extension

gettext

2.10 Code Documentation

All the project files contain some level of inline documentation. Organized below by module.

Note: This does not include API endpoints documented via swagger, as the swagger syntax is incompatable with restructuredText

2.10.1 Portal

Portal module

portal.factories.app.**configure_app**(*app*, *config*)
Load successive configs - overriding defaults

portal.factories.app.**configure_blueprints**(*app*, *blueprints*)
Register blueprints with application

portal.factories.app.**configure_cache**(*app*)
Configure requests-cache

portal.factories.app.**configure_csrf**(*app*)
Initialize CSRF protection

See *csrf.csrf_protect()* for implementation. Not using default as OAuth API use needs exclusion.

portal.factories.app.**configure_dogpile**(*app*)
Initialize dogpile cache with config values

portal.factories.app.**configure_extensions**(*app*)
Bind extensions to application

portal.factories.app.**configure_healthcheck**(*app*)
Configure the API used to check the health of our dependencies

portal.factories.app.**configure_logging**(*app*)
Configure logging.

portal.factories.app.**configure_metadata**(*app*)
Add distribution metadata for display in templates

portal.factories.app.**create_app**(*config=None*, *app_name=None*, *blueprints=None*)
Returns the configured flask app

AUDIT module

Maintain a log exclusively used for recording auditable events.

Any action deemed an auditable event should make a call to auditable_event()

Audit data is also persisted in the database *audit* table.

portal.audit.**auditable_event**(*message*, *user_id*, *subject_id*, *context=u'other'*)
Record auditable event

message: The message to record, i.e. "log in via facebook" user_id: The authenticated user id performing the action subject_id: The user id upon which the action was performed

portal.audit.**configure_audit_log**(*app*)
Configure audit logging.

The audit log is only active when running as a service (not during database updates, etc.) It should only received auditable events and never be rotated out.

Extensions used at application level

Generally the objects instantiated here are needed for imports throughout the system, but require factory pattern initialization once the flask *app* comes to life.

Defined here to break the circular dependencies. See *app.py* for additional configuration of most objects defined herein.

class portal.extensions.**OAuthOrAlternateAuth**(*app=None*)
Specialize OAuth2Provider with alternate authorization

require_oauth(**scopes*)
Specialze the superclass decorator with alternates

This method is intended to be in lock step with the super class, with the following two exceptions:

1. if actively "TESTING", skip oauth and return the function, effectively undecorated.
2. if the user appears to be locally logged in (i.e. browser session cookie with a valid user.id), return the effecively undecorated function.

Namespace module to house system URIs for use in FHIR

2.10.2 Portal.Config

Configuration

class portal.config.config.**BaseConfig**
Base configuration - override in subclasses

class portal.config.config.**DefaultConfig**
Default configuration

class portal.config.config.**TestConfig**
Testing configuration - used by unit tests

portal.config.config.**best_sql_url**()
Return compliant sql url from available environment variables

portal.config.config.**testing_sql_url**()
Return compliant sql url from available environment variables

If tests are being run with pytest-xdist workers, a pre-existing database will be required for each worker, suffixed with the worker index.

SitePersistence Module

class portal.config.site_persistence.**ModelDetails**(*cls*, *sequence_name*, *lookup_field*)

cls
Alias for field number 0

lookup_field
Alias for field number 2

sequence_name
Alias for field number 1

class portal.config.site_persistence.**SitePersistence**(*target_dir*)
Manage import and export of dynamic site data

export(*staging_exclusion=False*)
Generate JSON files defining dynamic site objects

Parameters **staging_exclusion** – set only if persisting exclusions to retain on staging when pulling over production data

Export dynamic data, such as Organizations and Access Strategies for import into other sites. This does NOT export values expected to live in the site config file or the static set generated by the seed managment command.

To import the data, use the seed command as defined in manage.py

import_(*keep_unmentioned*, *staging_exclusion=False*)
If persistence file is found, import the data

Parameters

- **keep_unmentioned** – if True, unmentioned data, such as an organization or intervention in the current database but not in the persistence file, will be left in place. if False, any unmentioned data will be purged as part of the import process.
- **staging_exclusion** – set only if persisting exclusions to retain on staging when pulling over production data

2.10.3 Portal.Models

Address module

Address data lives in the 'addresses' table. Several entities link to address via foreign keys.

class portal.models.address.**Address**(***kwargs*)
SQLAlchemy class for *addresses* table

as_fhir()

city

country

district

classmethod from_fhir(*data*)

id

line1

line2

line3

lines

postalCode

state

type

use

Audit Module

class portal.models.audit.**Audit**(***kwargs*)
ORM class for audit data

Holds meta info about changes in other tables, such as when and by whom the data was added. Several other tables maintain foreign keys to audit rows, such as *Observation* and *Procedure*.

as_fhir()
Typically included as *meta* data in containing FHIR resource

comment

context

classmethod from_logentry(*entry*)
Parse and create an Audit instance from audit log entry

Prior to version v16.5.12, audit entries only landed in log. This may be used to convert old entries, but newer ones should already be there.

id

subject_id

timestamp

user_id

version

class portal.models.audit.**Context**

account = 5

assessment = 2

authentication = 3

consent = 6

group = 10

intervention = 4

login = 1

observation = 8

organization = 9

other = 0

procedure = 11

relationship = 12

role = 13

tou = 14

user = 7

portal.models.audit.**lookup_version**()

Auth related model classes

class portal.models.auth.**AuthProvider**(***kwargs*)

as_fhir()

created_at

id

provider

provider_id

token

user

user_id

class portal.models.auth.**AuthProviderPersistable**(***kwargs*)

For persistence to function, need instance serialization

The base class for AuthProvider implements a non persistence-compliant version of `as_fhir()` as needed to show FHIR compliant identifiers in demographics.

This subclass (adapter) exists solely to provide serialization methods that work with persistence.

as_fhir()

serialize the AuthProvider

created_at

classmethod from_fhir(*data*)

id

provider

provider_id

token

update_from_fhir(*data*)

user

user_id

class portal.models.auth.**Grant**(***kwargs*)

client

client_id

code

delete()

expires

id

redirect_uri

scopes

user

user_id

validate_redirect_uri(*redirect_uri*)
Validate the redirect_uri from the OAuth Grant request

The RFC requires exact match on the redirect_uri. In practice this is too great of a burden for the interventions. Make sure it's from the same scheme::///host:port the client registered with

http://tools.ietf.org/html/rfc6749#section-4.1.3

class portal.models.auth.**Mock**

class portal.models.auth.**Token**(***kwargs*)

access_token

as_json()
serialize the token - used to preserve service tokens

client

client_id

expires

classmethod from_json(*data*)

id

refresh_token

scopes

token_type

update_from_json(*data*)

user

user_id

portal.models.auth.**create_service_token**(*client*, *user*)
Generate and return a bearer token for service calls

Partners need a mechanism for automated, authorized API access. This function returns a bearer token for subsequent authorized calls.

NB - as this opens a back door, it's only offered to users with the single role 'service'.

portal.models.auth.**load_grant**(*client_id*, *code*)

portal.models.auth.**load_token**(*access_token=None*, *refresh_token=None*)

portal.models.auth.**save_grant**(*client_id*, *code*, *request*, **args*, ***kwargs*)

portal.models.auth.**save_token**(*token*, *request*, **args*, ***kwargs*)

portal.models.auth.**token_janitor**()
Called by scheduled job to clean up and send alerts

No value in keeping around stale tokens, so we delete any that have expired.

For service tokens, trigger an email alert if they will be expiring soon.

Returns list of unreachable email addresses

Model classes for retaining FHIR data

class portal.models.fhir.**BundleType**

portal.models.fhir.**bundle_results**(*elements*, *bundle_type=<BundleType.searchset: 8>*, *links=None*)
Generate FHIR Bundle from element lists

Parameters

- **elements** – iterable of FHIR Resources to bundle
- **bundle_type** – limited by FHIR to be of the BundleType enum.
- **links** – links related to this bundle, such as API used to generate

Returns a FHIR compliant bundle

portal.models.fhir.**v_or_first**(*value*, *field_name*)
Return desired from list or scalar value

Parameters

- **value** – the raw data, may be a single value (directly returned) or a list from which the first element will be returned
- **field_name** – used in error text when multiple values are found for a constrained item.

Some fields, such as *name* were assumed to always be a single dictionary containing single values, whereas the FHIR spec defines them to support 0..* meaning we must handle a list.

NB - as the datamodel still only expects one, a 400 will be raised if given multiple values, using the *field_name* in the text.

portal.models.fhir.**v_or_n**(*value*)
Return None unless the value contains data

class portal.models.flaskdanceprovider.**FacebookFlaskDanceProvider**(*blueprint*, *token*)
fetches user info from Facebook after successfull auth

After the user successfully authenticates with Facebook this class fetches the user's info from Facebook

send_get_user_json_request()
sends a GET request to Facebook for user data

This function is used to get user information from Facebook that is encoded in json.

:return Response

class portal.models.flaskdanceprovider.**FlaskDanceProvider**(*blueprint*, *token*, *standard_key_to_provider_key_map*)
base class for flask dance providers

When a new provider is added to the protal's consumer oauth flow a descendent of this class needs to be created to get the user's information from the provider after a successful auth

get_user_info()
gets user info from the provider

This function parses json returned from the provider and returns an instance of FlaskProviderUserInfo that is filled with the user's information

:return FlaskProviderUserInfo with the user's info

parse_json(*user_json*)
parses the user's json and returns it in a standard format

Providers encode user information in json. This function parses the json and stores values in an instance of FlaskProviderUserInfo

Parameters **user_json** – info about the user encoded in json

:return instance of FlaskProviderUserInfo with the user's info

send_get_user_json_request()
sends a request to the provider to get user json

This function must be overriden in descendant classes to return a response with the user's json

class portal.models.flaskdanceprovider.**FlaskProviderUserInfo**
a common format for user info fetched from providers

Each provider packages user info a litle differently. Google, for example, uses "given_name" and the key for the user's first name, and Facebook uses "first_name". To make it easier for our code to parse responses in a common function this class provides a common format to store the results from each provider.

class portal.models.flaskdanceprovider.**GoogleFlaskDanceProvider**(*blueprint*, *token*)
fetches user info from Google after successfull auth

After the user successfully authenticates with Google this class fetches the user's info from Google

send_get_user_json_request()
sends a GET request to Google for user data

This function is used to get user information from Google that is encoded in json.

:return Response

class portal.models.flaskdanceprovider.**MockFlaskDanceProvider**(*provider_name*, *token*, *user_json*, *fail_to_get_user_json*)
creates user info from test data to validate auth logic

This class should only be used during testing. It simply mocks user json that is normally retrieved from a provider which allows us to granularly test auth logic

send_get_user_json_request()
return a mock request based on test data passed into the constructor

Normally a request is sent to a provider and user json is returned. This function mocks out that request by returning a response with the user json passed through the test backdoor

class portal.models.flaskdanceprovider.**MockJsonResponse**(*ok*, *user_json*)
mocks a GET json response

During auth we send a request to providers that returns user json. During tests we need to mock out providers so we can test our auth logic. This class is used to mock out requests that are normally sent to providers.

json()
returns mock json

Identifier Model Module

class portal.models.identifier.**Identifier**(***kwargs*)
Identifier ORM, for FHIR Identifier resources

add_if_not_found(*commit_immediately=False*)
Add self to database, or return existing

Queries for similar, matching on **system** and **value** alone. Note the database unique constraint to match.

@return: the new or matched Identifier

class portal.models.identifier.**UserIdentifier**(***kwargs*)
ORM class for user_identifiers data

Holds links to any additional identifiers a user may have, such as study participation.

Intervention Module

class portal.models.intervention.**DisplayDetails**(*access*, *intervention*, *user_intervention*)
Simple abstraction to communicate display details to front end

To provide a custom experience, intevention access can be set at several levels. For a user, access is either available or not, and when available, the link controls may be intentionally disabled for a reason the intervention should note in the status_text field.

Attributes:: access: {True, False} card_html: Text to display on the card link_label: Text used to label the button or hyperlink link_url: URL for the button or link - link to be disabled when null status_text: Text to inform user of status, or why it's disabled

class portal.models.intervention.**Intervention**(***kwargs*)

as_json()
Returns the 'safe to export' portions of an intervention

The client_id and link_url are non-portable between systems. The id is also independent - return the rest of the not null fields as a simple json dict.

NB for staging exclusions to function, link_url and client_id are now included. Take care to remove it from persistence files where it is NOT portable, for example, when generating persistence files programmatically.

display_for_user(*user*)
Return the intervention display details for the given user

Somewhat complicated method, depending on intervention configuration. The following ordered steps are used to determine if a user should have access to an intervention. The first 'true' found provides access, otherwise the intervention will not be displayed.

1. call each strategy_function in intervention.access_strategies. Note, on rare occasions, a strategy may alter the UserIntervention attributes given the circumstances.
2. check for a UserIntervention row defining access for the given user on this intervention.
3. check if the intervention has *public_access* set

@return DisplayDetails object defining 'access' and other details for how to render the intervention.

fetch_strategies()
Generator to return each registered strategy

Strategies need to be brought to life from their persisted state. This generator does so, and returns them in a call ready fashion, ordered by the strategy's rank.

quick_access_check(*user*)
Return boolean representing given user's access to intervention

Somewhat complicated method, depending on intervention configuration. The following ordered steps are used to determine if a user should have access to an intervention. The first 'true' found is returned (as to make the check as quick as possible).

1. check if the intervention has *public_access* set
2. check for a UserIntervention row defining access for the given user on this intervention.
3. call each strategy_function in intervention.access_strategies.

@return boolean representing 'access'.

class portal.models.intervention.**UserIntervention**(***kwargs*)

classmethod user_access_granted(*intervention_id*, *user_id*)
Shortcut to query for specific (intervention, user) access

portal.models.intervention.**add_static_interventions**()
Seed database with default static interventions

Idempotent - run anytime to push any new interventions into existing dbs

Module for intervention access strategy functions

Determining whether or not to provide access to a given intervention for a user is occasionally tricky business. By way of the access_strategies property on all interventions, one can add additional criteria by defining a function here (or elsewhere) and adding it to the desired intervention.

function signature: takes named parameters (intervention, user) and returns a boolean - True grants access (and short circuits further access tests), False does not.

NB - several functions are closures returning access_strategy functions with the parameters given to the closures.

class portal.models.intervention_strategies.**AccessStrategy**(***kwargs*)
ORM to persist access strategies on an intervention

The function_details field contains JSON defining which strategy to use and how it should be instantiated by one of the closures implementing the access_strategy interface. Said closures must be defined in this module (a security measure to keep unsanitized code out).

as_json()
Return self in JSON friendly dictionary

instantiate()
Bring the serialized access strategy function to life

Using the JSON in self.function_details, instantiate the function and return it ready to use.

portal.models.intervention_strategies.**allow_if_not_in_intervention**(*intervention_name*)
Strategy API checks user does not belong to named intervention

portal.models.intervention_strategies.**combine_strategies**(***kwargs*)
Make multiple strategies into a single statement

The nature of the access lookup returns True for the first success in the list of strategies for an intervention. Use this method to chain multiple strategies together into a logical **and** fashion rather than the built in locical **or**.

NB - kwargs must have keys such as 'strategy_n', 'strategy_n_kwargs' for every 'n' strategies being combined, starting at 1. Set arbitrary limit of 6 strategies for time being.

Nested strategies may actually want a logical 'OR'. Optional kwarg *combinator* takes values {'any', 'all'} - default 'all' means all strategies must evaluate true. 'any' means just one must eval true for a positive result.

portal.models.intervention_strategies.**in_role_list**(*role_list*)
Requires user is associated with any role in the list

portal.models.intervention_strategies.**limit_by_clinic_w_id**(*identifier_value*, *identifier_system='http://us.truenth.org/identit* *codes/decision-support-group'*, *combinator=u'any'*, *include_children=True*)
Requires user is associated with {any,all} clinics with identifier

Parameters

- **identifier_value** – value string for identifer associated with org(s)
- **identifier_system** – system string for identifier, defaults to DECISION_SUPPORT_GROUP
- **combinator** – determines if the user must be in 'any' (default) or 'all' of the clinics in the given list. NB combining 'all' with include_children=True would mean all orgs in the list AND all chidren of all orgs in list must be associated with the user for a true result.
- **include_children** – include children in the organization tree if set (default), otherwise, only include the organizations in the list

portal.models.intervention_strategies.**not_in_clinic_w_id**(*identifier_value*, *identifier_system='http://us.truenth.org/identity-codes/decision-support-group'*, *include_children=True*)
Requires user isn't associated with any clinic in the list

Parameters

- **identifier_value** – value string for identifer associated with org(s)
- **identifier_system** – system string for identifier, defaults to DECISION_SUPPORT_GROUP
- **include_children** – include children in the organization tree if set (default), otherwise, only include the organizations directly associated with the identifier

portal.models.intervention_strategies.**not_in_role_list**(*role_list*)
Requires user isn't associated with any role in the list

portal.models.intervention_strategies.**observation_check**(*display*, *boolean_value*, *invert_logic=False*)
Returns strategy function for a particular observation and logic value

Parameters

- **display** – observation coding.display from TRUENTH_CLINICAL_CODE_SYSTEM
- **boolean_value** – ValueQuantity boolean true or false expected
- **invert_logic** – Effective binary `not` to apply to test. If set, will return True only if given observation with boolean_value is NOT defined for user

NB a history of observations is maintained, with the most recent taking precedence.

portal.models.intervention_strategies.**tx_begun**(*boolean_value*)
Returns strategy function testing if user is known to have started Tx

Parameters **boolean_value** – true for known treatment started (i.e. procedure indicating tx has begun), false to confirm a user doesn't have a procedure indicating tx has begun

portal.models.intervention_strategies.**update_card_html_on_completion**()
Update description and card_html depending on state

portal.models.lazy.**lazyprop**(*fn*)
Property decorator for lazy intialization (load on first request)

Useful on any expensive to load attribute on any class. Simply decorate the 'getter' with @lazyprop, where the function definition loads the object to be assigned to the given attribute.

As the SQLAlchemy session is NOT thread safe and this tends to be the primary use of the lazyprop decorator, we include the thread identifier in the key

portal.models.lazy.**query_by_name**(*cls*, *name*)
returns a lazy load function capable of caching object

Use this alternative for classes with dynamic attributes (names not hardcoded in class definition), as property decorators (i.e. @lazyprop) don't function properly.

As the SQLAlchemy session is NOT thread safe, we include the thread identifier in the key

NB - attribute instances must be unique over (cls.__name__, name) within the containing class to avoid collisions.

@param cls: ORM class to query @param name: name field in ORM class to uniquely define object

Model classes for message data

class portal.models.message.**EmailMessage**(***kwargs*)

as_json()

body

id

recipients

send_message(*cc_address=None*)
Send the message

Parameters **cc_address** – include valid email address to send a carbon copy

NB the cc isn't persisted with the rest of the record.

sender

sent_at

static style_message(*body*)
Implicitly called on send, to wrap body with style tags

subject

user_id

portal.models.message.**log_message**(*message*, *app*)
Configured to handle signals on email_dispatched - log the event

Model classes for organizations and related entities.

Designed around FHIR guidelines for representation of organizations, locations and healthcare services which are used to describe hospitals and clinics.

class portal.models.organization.**LocaleExtension**(*organization*, *extension*)

children

extension_url = 'http://hl7.org/fhir/valueset/languages'

class portal.models.organization.**OrgNode**(*id*, *parent=None*, *children=None*)
Node in tree of organizations - used by org tree

Simple tree implementation to house organizations in a hierarchical structure. One root - any number of nodes at each tier. The organization identifiers (integers referring to the database primary key) are used as reference keys.

insert(*id*, *partOf_id=None*)
Insert new nodes into the org tree

Designed for this special organizaion purpose, we expect the tree is built from the top (root) down, so no rebalancing is necessary.

Parameters

- **id** – of organizaiton to insert
- **partOf_id** – if organization has a parent - its identifier

Returns the newly inserted node

top_level()
Lookup top_level organization id from the given node

Use OrgTree.find() to locate starter node, if necessary

class portal.models.organization.**OrgTree**
In-memory organizations tree for hierarchy and structure

Organizations may define a 'partOf' in the database records to describe where the organization fits in a hierarchy. As there may be any number of organization tiers, and the need exists to lookup where an organiztion fits in this hiearchy. For example, needing to lookup the top level organization for any node, or all the organizations at or below a level for permission issues. etc.

This singleton class will build up the tree when it's first needed (i.e. lazy load).

Note, the root of the tree is a dummy object, so the first tier can be multiple *top-level* organizations.

static all_ids_with_rp(*research_protocol*)
Returns set of org IDs that are associated with Research Protocol

As child orgs are considered to be associated if the parent org is, this will return the full list for optimized comparisons.

all_leaf_ids()

all_leaves_below_id(*organization_id*)
Given org at arbitrary level, return list of leaf nodes below it

all_top_level_ids()
Return list of all top level organization identifiers

at_and_above_ids(*organization_id*)
Returns list of ids from any point in tree and up the parent stack

Parameters **organization_id** – node in tree, will be included in return list

Returns list of organization ids from the one given on up including every parent found in chain

at_or_below_ids(*organization_id*, *other_organizations*)
Check if the other_organizations are at or below given organization

Parameters

- **organization_id** – effective parent to check against
- **other_organizations** – iterable of organization_ids as potential children.

Returns True if any org in other_organizations is equal to the given organization_id, or a child of it.

find(*organization_id*)
Locates and returns node in OrgTree for given organization_id

Parameters **organization_id** – primary key of organization to locate

Returns OrgNode from OrgTree

Raises ValueError if not found - unexpected

find_top_level_orgs(*organizations*, *first=False*)
Returns top level organization(s) from those provided

Parameters

- **organizations** – organizations against which top level organization(s) will be queried
- **first** – if set, return the first org in the result list rather than a set of orgs.

Returns set of top level organization(s), or a single org if `first` is set.

here_and_below_id(*organization_id*)
Given org at arbitrary level, return list at and below

classmethod invalidate_cache()
Invalidate cache on org changes

lookup_table = None

populate_tree()
Recursively build tree from top down

root = None

top_level_names()
Fetch org names for *all_top_level_ids*

Returns list of top level org names

visible_patients(*staff_user*)
Returns patient IDs for whom the current staff_user can view

Staff users can view all patients at or below their own org level.

NB - no patients should ever have a consent on file with the special organization 'none of the above' - said organization is ignored in the search.

class portal.models.organization.**Organization**(***kwargs*)
Model representing a FHIR organization

Organizations represent a collection of people that have come together to achieve an objective. As an example, all the healthcare services provided by the same university hospital will belong to the organization representing said university hospital.

Organizations can reference other organizations via the 'partOf_id', where children name their parent organization id.

addresses

as_fhir(*include_empties=True*)
Return JSON representation of organization

Parameters **include_empties** – if True, returns entire object definition; if False, empty elements are removed from the result

Returns JSON representation of a FHIR Organization resource

coding_options

static consent_agreements(*locale_code*)
Return consent agreements for all top level organizations

Parameters **locale_code** – preferred locale, typically user's.

Returns dictionary keyed by top level organization id containing a VersionedResource for each organization IFF the organization has a custom consent agreement on file. The *organization_name* is also added to the versioned resource to simplify UI code.

default_locale

default_locale_id

email

ethnicity_codings

classmethod from_fhir(*data*)

classmethod generate_bundle(*limit_to_ids=None*, *include_empties=True*)
Generate a FHIR bundle of existing orgs ordered by ID

Parameters

- **limit_to_ids** – if defined, only return the matching set, otherwise all organizations found
- **include_empties** – set to include empty attributes

Returns

id

identifiers

indigenous_codings

locales

name

organization_research_protocols

partOf_id

phone

phone_id

race_codings

research_protocol(*as_of_date*)
Lookup research protocol for this org valid at as_of_date

Complicated scenario as it may only be defined on the parent or further up the tree. Secondly, we keep history of research protocols in case backdated entry is necessary.

Returns research protocol for org (or parent org) valid as_of_date

research_protocols
A descriptor that presents a read/write view of an object attribute.

rps_w_retired(*consider_parents=False*)
accessor to collate research protocols and retired_as_of values

The SQLAlchemy association proxy doesn't provide easy access to *intermediary* table data - i.e. columns in the link table between a many:many association. This accessor collates the value stored in the intermediary table, *retired_as_of* with the research protocols for this organization.

Parameters **consider_parents** – if set and the org doesn't have an associated RP, continue up the org hiearchy till one is found.

Returns ready query for use in iteration or count or other methods. Query will produce a list of tuples (ResearchProtocol, retired_as_of) associated with the organization, ordered by *retired_as_of* dates with nulls last.

shortname
Return shortname identifier if found, else the org name

timezone

type

type_id

update_from_fhir(*data*)

use_specific_codings

users

class portal.models.organization.**OrganizationAddress**(***kwargs*)
link table for organization : n addresses

address_id

id

organization_id

class portal.models.organization.**OrganizationIdentifier**(***kwargs*)
link table for organization : n identifiers

id

identifier_id

organization_id

class portal.models.organization.**OrganizationLocale**(***kwargs*)

coding_id

id

organization_id

class portal.models.organization.**OrganizationResearchProtocol**(*research_protocol=None*, *organization=None*, *retired_as_of=None*)

id

organization

organization_id

research_protocol

research_protocol_id

retired_as_of

class portal.models.organization.**ResearchProtocolExtension**(*organization*, *extension*)

apply_fhir()

as_fhir(*include_empties=True*)

children

extension_url = 'http://us.truenth.org/identity-codes/research-protocol'

class portal.models.organization.**UserOrganization**(***kwargs*)
link table for users (n) : organizations (n)

id

organization

organization_id

user_id

portal.models.organization.**add_static_organization**()
Insert special *none of the above* org at index 0

portal.models.organization.**org_extension_map**(*organization*, *extension*)
Map the given extension to the Organization

FHIR uses extensions for elements beyond base set defined. Lookup an adapter to handle the given extension for the organization.

Parameters

- **organization** – the org to apply to or read the extension from
- **extension** – a dictionary with at least a 'url' key defining the extension.

Returns adapter implementing apply_fhir and as_fhir methods

:raises exceptions.ValueError: if the extension isn't recognized

Performer module - encapsulate the FHIR Performer resource

class portal.models.performer.**ObservationPerformer**(***kwargs*)
Link table for observation to list of performers

id

observation_id

performer_id

class portal.models.performer.**Performer**(***kwargs*)
ORM for FHIR Performer - performers table

add_if_not_found(*commit_immediately=False*)
Add self to database, or return existing

Queries for matching, existing Performer. Populates self.id if found, adds to database first if not.

as_fhir()
Return self in JSON FHIR formatted string

FHIR is not currently consistant in performer inclusion. For example, Observation.performer is simply a list of Reference resources, whereas Procedure.performer is a list including the resource labeled as an *actor* and a codable concept labeled as the *role* defining the actor's role.

Returns the best JSON FHIR formatted string for the instance

codeable_concept

codeable_concept_id
The codeable concept for performers including a role

classmethod from_fhir(*fhir*)
Return performer instance from JSON FHIR formatted string

See note in *as_fhir*, the format of a performer depends on context. Populate *self.codeable_concept* only if it's included as a *role*.

Returns new performer instance from values in given *fhir*

id

observations

reference_txt
Text for performer (aka *actor*), i.e. {"reference": "patient/12"}

Procedure Model

class portal.models.procedure.**Procedure**(***kwargs*)
ORM class for procedures

Similar to the profiles published by SMART

Each Procedure must have*Procedure* must have

1 patient in Procedure.subject (aka Procedure.user)

1 code in Procedure.code (pointing to a CodeableConcept) with *system* of http://snomed.info/sct

1 performed datetime in Procedure.performedDateTime

as_fhir()
produces FHIR representation of procedure in JSON format

audit
tracks when and by whom the *procedure* was retained, included as *meta* data in the FHIR output

code
procedure.code (a *CodeableConcept*) defines the procedure. coding.system is required to be *http://snomed.info/sct*

end_time
when defined, produces a performedPeriod, otherwise *start_time* is used alone as performedDateTime

classmethod from_fhir(*data*, *audit*)
Parses FHIR data to produce a new procedure instance

start_time
required whereas end_time is optional

Reference module - encapsulate FHIR Reference type

exception portal.models.reference.**MissingReference**
Raised when FHIR references cannot be found

exception portal.models.reference.**MultipleReference**
Raised when FHIR references retrieve multiple results

class portal.models.reference.**Reference**

as_fhir()
Return FHIR compliant reference string

FHIR uses the Reference Resource within a number of other resources to define things like who performed an observation or what organization another is a partOf.

Returns the appropriate JSON formatted reference string.

classmethod intervention(*intervention_id*)
Create a reference object from given intervention

Intervention references maintained by name - lookup from given id.

classmethod organization(*organization_id*)
Create a reference object from a known organization id

classmethod parse(*reference_dict*)
Parse an organization from a FHIR Reference resource

Typical format: "{'Reference': 'Organization/12'}" or "{'reference': 'api/patient/6'}"

FHIR is a little sloppy on upper/lower case, so this parser is also flexible.

Returns the referenced object - instantiated from the db

:raises ***portal.models.reference.MissingReference*****: if** the referenced object can not be found

:raises ***portal.models.reference.MultipleReference*****: if** the referenced object retrieves multiple results

:raises **exceptions.ValueError****: if the text format** can't be parsed

classmethod patient(*patient_id*)
Create a reference object from a known patient id

classmethod practitioner(*practitioner_id*)
Create a reference object from a known patient id

classmethod questionnaire(*questionnaire_name*)
Create a reference object from a known questionnaire name

classmethod questionnaire_bank(*questionnaire_bank_name*)
Create a reference object from a known questionnaire bank

classmethod research_protocol(*research_protocol_name*)
Create a reference object from a known research protocol

Relationship module

Relationship data lives in the *relationships* table, populated via: *FLASK_APP=manage.py flask seed*

To extend the list of roles, add name: description pairs to the STATIC_RELATIONSHIPS dict within, and rerun the seed command above.

class portal.models.relationship.**Relationship**(***kwargs*)
SQLAlchemy class for *relationships* table

description

id

name

portal.models.relationship.**add_static_relationships**()
Seed database with default static relationships

Idempotent - run anytime to pick up any new relationships in existing dbs

Role module

Role data lives in the *roles* table, populated via: *flask seed*

To restrict access to a given role, use the ROLE object: @roles_required(ROLE.ADMIN.value)

To extend the list of roles, add name: description pairs to the STATIC_ROLES dict within.

class portal.models.role.**Role**(***kwargs*)
SQLAlchemy class for *roles* table

as_json()

description

display_name
Generate and return 'Title Case' version of name 'title_case'

id

name

users

portal.models.role.**add_static_roles**()
Seed database with default static roles

Idempotent - run anytime to pick up any new roles in existing dbs

Telecom Module

FHIR uses a telecom structure for email, fax, phone, etc.

class portal.models.telecom.**ContactPoint**(***kwargs*)
ContactPoint model for storing FHIR telecom entries

as_fhir()

classmethod from_fhir(*data*)

id

rank

system

update_from_fhir(*data*)

use

value

class portal.models.telecom.**Telecom**(*email=None*, *contact_points=None*)
Telecom model - not a formal db front at this time

Several FHIR resources include telecom entries. This helper class wraps common functions.

as_fhir()

cp_dict()

classmethod from_fhir(*data*)

User model

exception portal.models.user.**RoleError**

class portal.models.user.**User**(***kwargs*)

active

add_observation(*fhir*, *audit*)

add_organization(*organization_name*)
Shortcut to add a clinic/organization by name

add_password_verification_failure()
remembers when a user fails password verification

Each time a user fails password verification this function is called. Use user.is_locked_out to tell whether this has been called enough times to lock the user out of the system

Returns total failures since last reset

add_relationship(*other_user*, *relationship_name*)

add_roles(*role_list*, *acting_user*)
Add one or more roles to user's existing roles

Parameters

- **role_list** – list of role objects defining what roles to add
- **acting_user** – user performing action, for permissions, etc.

Raises 409 if any named roles are already assigned to the user

add_service_account()
Service account generation.

For automated, authenticated access to protected API endpoints, a service user can be created and used to generate a long-life bearer token. The account is a user with the service role, attached to a sposor account - the (self) individual creating it.

Only a single service account is allowed per user. If one is found to exist for this user, simply return it.

all_consents
Access to all consents including deleted and expired

alt_phone

alt_phone_id

as_fhir(*include_empties=True*)
Return JSON representation of user

Parameters **include_empties** – if True, returns entire object definition; if False, empty elements are removed from the result

Returns JSON representation of a FHIR Patient resource

auth_providers

birthdate

check_role(*permission*, *other_id*)
check user for adequate role

if user is an admin or a service account, grant carte blanche otherwise, must be self or have a relationship granting permission to "verb" the other user.

returns true if permission should be granted, raises 404 if the other_id can't be found, otherwise raise a 401

clinical_history(*requestURL=None*, *patch_dstu2=False*)

classmethod column_names()

concept_value(*codeable_concept*)
Look up logical value for given concept

Returns the most current setting for a given concept, by interpreting the results of a matching `fetch_value_status_for_concept()` call.

NB - as there are states beyond true/false, such as "unknown" for a given concept, this does NOT return a boolean but a string.

Returns a string, typically "true", "false" or "unknown"

confirmed_at

current_encounter
Shortcut to current encounter, if present

An encounter is typically bound to the logged in user, not the subject, if a different user is performing the action.

deactivate_tous(*acting_user*, *types=None*)
Mark user's current active ToU agreements as inactive

Marks the user's current active ToU agreements as inactive. User must agree to ToUs again upon next login (per CoreData logic). If types provided, only deactivates agreements of that ToU type. Called when the ToU agreement language is updated.

Parameters

- **acting_user** – user behind the request for permission checks
- **types** – ToU types for which to invalide agreements (optional)

deceased

deceased_id

delete_roles(*role_list*, *acting_user*)
Delete one or more roles from user's existing roles

Parameters

- **role_list** – list of role objects defining what roles to remove
- **acting_user** – user performing action, for permissions, etc.

Raises 409 if any named roles are not currently assigned to the user

delete_user(*acting_user*)
Mark user deleted from the system

Due to audit constraints, we do NOT actually delete the user, but mark the user as deleted. See *permanently_delete_user* for more serious alternative.

Parameters

- **self** – user to mark deleted
- **acting_user** – individual executing the command, for audit trail

deleted

deleted_id

display_name

documents

email

email_ready()
Returns (True, None) IFF user has valid email & necessary criteria

As user's frequently forget their passwords or start in a state without a valid email address, the system should NOT email invites or reminders unless adequate data is on file for the user to perform a reset password loop.

NB exceptions exist for systems with the NO_CHALLENGE_WO_DATA configuration set, as those systems allow for change of password without the verification step, if the user doesn't have a required field set.

Returns (Success, Failure message), such as (True, None) if the user account is "email_ready" or (False, _"invalid email") if the reason for failure is a lack of valid email address.

encounters

ethnicities

external_study_id
Return the value of the user's external study identifier(s)

If more than one external study identifiers are found for the user, values will be joined by ', '

failed_login_attempts_before_lockout
Number of failed login attempts before lockout

fetch_datetime_for_concept(*codeable_concept*)
Return newest issued timestamp from matching observation

fetch_value_status_for_concept(*codeable_concept*)
Return matching ValueQuantity & status for this user

Given the possibility of multiple matching observations, returns the most current info available.

See also `concept_value()`

Returns (value_quantity, status) tuple for the observation if found on the user, else (None, None)

first_name

first_top_organization()
Return first top level organization for user

NB, none of the above doesn't count and will not be retuned.

A user may have any number of organizations, but most business decisions, assume there is only one. Arbitrarily returning the first from the matching query in case of multiple.

Returns a single top level organization, or None

classmethod from_fhir(*data*)

fuzzy_match(*first_name*, *last_name*, *birthdate*)
Returns probability score [0-100] of it being the same user

gender

groups

has_relationship(*relationship_name*, *other_user*)

has_role(*role_name*)
Return True if the user has one of the specified roles. Return False otherwise.

has_roles() accepts a 1 or more role name parameters has_role(role_name1, role_name2, role_name3).

For example: has_roles('a', 'b')

Translates to: User has role 'a' OR role 'b'

id

identifiers
Return list of identifiers

Several identifiers are "implicit", such as the primary key from the user table, and any auth_providers associated with this user. These will be prepended to the existing identifiers but should never be stored, as they're generated from other fields.

Returns list of implicit and existing identifiers

image_url

implicit_identifiers()
Generate and return the implicit identifiers

The primary key, email and auth providers are all visible in formats such as demographics, but should never be stored as user_identifiers, less problems of duplicate, out of sync data arise.

This method generates those on the fly for display purposes.

Returns list of implicit identifiers

indigenous

interventions

is_locked_out
tells if user is temporarily locked out

To slow down brute force password attacks we temporarily lock users out of the system for a short period of time. This property tells whether or not the user is locked out.

is_registered()
Returns True if user has completed registration

Not to be confused with the `registered` column (which captures the moment when the account was created), `is_registered` returns true once the user has blessed their account with login credentials, such as a password or auth_provider access.

Roles are considered in this check - special roles such as `access_on_verify` and `write_only` should never exist on registered users, and therefore this method will return False for any users with these roles.

last_name

last_password_verification_failure

leaf_organizations()
Return list of 'leaf' organization ids for user's orgs

Users, especially staff, have arbitrary number of organization associations, at any level of the organization hierarchy. This method looks up all child leaf nodes from the users existing orgs.

locale

locale_code

locale_display_options
Collates all the locale options from the user's orgs to establish which should be visible to the user

locale_id

locale_name

lockout_period_minutes
The lockout period in minutes

lockout_period_timedelta
The lockout period as a timedelta

mask_email(*prefix=u'__invite__'*)
Mask temporary account email to avoid collision with registered

Temporary user accounts created for the purpose of invites get in the way of the user creating a registered account. Add a hidden prefix to the email address in the temporary account to avoid collision.

merge_with(*other_id*)
merge details from other user into self

Primary usage stems from different account registration flows. For example, users are created when invited by staff to participate, and when the same user later opts to register, a second account is generated during the registration process (either by flask-user or other mechanisms like add_user).

NB - caller MUST manage email due to unique constraints

notifications

observations

org_coding_display_options
Collates all race/ethnicity/indigenous display options from the user's orgs to establish which options to display

organizations

password

password_verification_failures

phone

phone_id

practitioner_id

procedure_history(*requestURL=None*)

procedures

promote_to_registered(*registered_user*)
Promote a weakly authenticated account to a registered one

questionnaire_responses

races

reactivate_user(*acting_user*)
Reactivate a previously deleted user

This method clears the deleted status - by removing the link from the user to the audit recording the delete. Audit itself is retained for tracking purposes, and a new one will be created for posterity

Parameters

- **self** – user to reactivate
- **acting_user** – individual executing the command, for audit trail

registered

relationships

reset_lockout()
resets variables that track lockout

We track when the user fails password verification to lockout users when they fail too many times. This function resets those variables

reset_password_token

rolelist
Generate UI friendly string of user's roles by name

roles

save_observation(*codeable_concept*, *value_quantity*, *audit*, *status*, *issued*)
Helper method for creating new observations

staff_html()
Helper used from templates to display any custom staff/provider text

Interventions can add personalized HTML for care staff to consume on the /patients list. Look up any values for this user on all interventions.

subject_audits

timezone

update_birthdate(*fhir*)

update_consents(*consent_list*, *acting_user*)
Update user's consents

Adds the provided list of consent agreements to the user. If the user had pre-existing consent agreements between the same organization_id, the new will replace the old

NB this will only modify/update consents between the user and the organizations named in the given consent_list.

update_deceased(*fhir*)

update_from_fhir(*fhir*, *acting_user=None*)
Update the user's demographics from the given FHIR

If a field is defined, it is the final definition for the respective field, resulting in a deletion of existing values in said field that are not included.

Parameters

- **fhir** – JSON defining portions of the user demographics to change
- **acting_user** – user requesting the change, used in audit logs

update_orgs(*org_list*, *acting_user*, *excuse_top_check=False*)
Update user's organizations

Uses given list of organizations as the definitive list for the user - meaning any current affiliations not mentioned will be deleted.

Parameters

- **org_list** – list of organization objects for user's orgs
- **acting_user** – user behind the request for permission checks
- **excuse_top_check** – Set True to excuse check for changes to top level orgs, say during initial account creation

update_roles(*role_list*, *acting_user*)
Update user's roles

Parameters

- **role_list** – list of role objects defining exactly what roles the user should have. Any existing roles not mentioned will be deleted from user's list
- **acting_user** – user performing action, for permissions, etc.

user_audits

username

valid_consents
Access to consents that have neither been deleted or expired

class portal.models.user.**UserEthnicity**(***kwargs*)

coding_id

id

user_id

class portal.models.user.**UserEthnicityExtension**(*user*, *extension*)

children

extension_url = u'http://hl7.org/fhir/StructureDefinition/us-core-ethnicity'

class portal.models.user.**UserIndigenous**(***kwargs*)

coding_id

id

user_id

class portal.models.user.**UserIndigenousStatusExtension**(*user*, *extension*)

children

extension_url = 'http://us.truenth.org/fhir/StructureDefinition/AU-NHHD-METeOR-id-2910

class portal.models.user.**UserRace**(***kwargs*)

coding_id

id

user_id

class portal.models.user.**UserRaceExtension**(*user*, *extension*)

children

extension_url = u'http://hl7.org/fhir/StructureDefinition/us-core-race'

class portal.models.user.**UserRelationship**(***kwargs*)

SQLAlchemy class for *user_relationships* table

Relationship is assumed to be ordered such that: <user_id> has a <relationship.name> with <other_user_id>

as_json()

serialize the relationship - used to preserve service users

classmethod from_json(*data*)

id

other_user

other_user_id

relationship

relationship_id

update_from_json(*data*)

user

user_id

class portal.models.user.**UserRoles**(***kwargs*)

id

role_id

user_id

portal.models.user.**add_role**(*user*, *role_name*)

portal.models.user.**add_user**(*user_info*)
Given the result from an external IdP, create a new user

portal.models.user.**current_user**()
Obtain the "current" user object

Works for both remote oauth sessions and locally logged in sessions.

returns current user object, or None if not logged in (local or remote)

portal.models.user.**default_email**(*context=None*)
Function to provide a unique, default email if none is provided

Parameters **context** – is populated by SQLAlchemy - see Context-Sensitive default functions in http://docs.sqlalchemy.org/en/latest/core/defaults.html

Returns a unique email string to avoid unique constraints, if an email isn't provided in the context

portal.models.user.**flag_test**()
Find all non-service users and flag as test

portal.models.user.**get_user**(*uid*)

portal.models.user.**get_user_or_abort**(*uid*, *allow_deleted=False*)
Wraps *get_user* and raises error if not found

Safe to call with path or parameter info. Confirms integer value before attempting lookup.

Parameters

- **uid** – integer value for user id to look up
- **allow_deleted** – set true to allow access to deleted users

:raises werkzeug.exceptions.BadRequest: w/o a uid

:raises werkzeug.exceptions.NotFound: if the given uid isn't an integer, or if no matching user

:raises werkzeug.exceptions.Forbidden: if the named user has been deleted, unless *allow_deleted* is set

Returns user if valid and found

portal.models.user.**permanently_delete_user**(*username*, *user_id=None*, *acting_user=None*, *actor=None*)
Given a username (email), purge the user from the system

Includes wiping out audit rows, observations, etc. May pass either username or user_id. Will prompt for acting_user if not provided.

Parameters

- **username** – username (email) for user to purge
- **user_id** – id of user in liew of username
- **acting_user** – user taking the action, for record keeping

portal.models.user.**user_extension_map**(*user*, *extension*)
Map the given extension to the User

FHIR uses extensions for elements beyond base set defined. Lookup an adapter to handle the given extension for the user.

Parameters

- **user** – the user to apply to or read the extension from
- **extension** – a dictionary with at least a 'url' key defining the extension. Should include a 'valueCodeableConcept' structure when being used in an apply context (i.e. direct FHIR data)

Returns adapter implementing apply_fhir and as_fhir methods

:raises exceptions.ValueError: if the extension isn't recognized

portal.models.user.**validate_email**(*email*)
Not done at model level, as there are exceptions

We allow for placeholders and masks on email, so not all emails are valid. This validation function is generally only used when an end user changing an address or another use requires validation.

Furthermore, due to the complexity of valid email addresses, just look for some obvious signs - such as the '@' symbol and at least 6 chars.

:raises werkzeug.exceptions.BadRequest: if obviously invalid

2.10.4 Portal.Views

Note: This does not include API endpoints documented via swagger, as the swagger syntax is incompatable with restructuredText

Auth related view functions

portal.views.auth.**deauthorized**()
Callback URL configured on facebook when user deauthorizes

We receive POST data when a user deauthorizes the session between TrueNTH and Facebook. The POST includes a signed_request, decoded as seen below.

Configuration set on Facebook Developer pages: app->settings->advanced->Deauthorize Callback URL

portal.views.auth.**next_after_login**()
Redirection to appropriate target depending on data and auth status

Multiple authorization paths in, some needing up front information before returning, this attempts to handle such state decisions. In other words, this function represents the state machine to control initial flow.

When client applications (interventions) request OAuth tokens, we sometimes need to postpone the action of authorizing the client while the user logs in to TrueNTH.

After completing authentication with TrueNTH, additional data may need to be obtained, such as a TOU agreement. In such a case, the user will be directed to initial_queries, then back here for redirection to the appropriate 'next'.

Implemented as a view method for integration with flask-user config.

portal.views.auth.**login**(*blueprint*, *token*)
successful provider login callback

After successful authorization at the provider, control returns here. The blueprint and the oauth bearer token are used to log the user into the portal

:return returns False to disable saving oauth token

portal.views.auth.**logout**(*prevent_redirect=False*, *reason=None*)
logout view function

Logs user out by requesting the previously granted permission to use authenticated resources be deleted from the OAuth server, and clearing the browser session.

Parameters

- **prevent_redirect** – set only if calling this function during another process where redirection after logout is not desired
- **reason** – set only if calling from another process where a driving reason should be noted in the audit

Optional query string parameter timed_out should be set to clarify the logout request is the result of a stale session

Cross Domain Decorators

portal.views.crossdomain.**crossdomain**(*origin=None*, *methods=None*, *headers=('Authorization', 'X-Requested-With', 'X-CSRFToken', 'Content-Type')*, *max_age=21600*, *automatic_options=True*)
Decorator to add specified crossdomain headers to response

Parameters

- **origin** – '*' to allow all origins, otherwise a string with a single origin or a list of origins that might access the resource. If no origin is provided, use request.headers['Origin'], but ONLY if it validates. If no origin is provided and the request doesn't include an **Origin** header, no CORS headers will be added.
- **methods** – Optionally a list of methods that are allowed for this view. If not provided it will allow all methods that are implemented.
- **headers** – Optionally a list of headers that are allowed for this request.
- **max_age** – The number of seconds as integer or timedelta object for which the preflighted request is valid.
- **automatic_options** – If enabled the decorator will use the default Flask OPTIONS response and attach the headers there, otherwise the view function will be called to generate an appropriate response.

:raises **werkzeug.exceptions.Unauthorized:** if no origin is provided and the one in request.headers['Origin'] doesn't validate as one we know.

Intervention API view functions

portal.views.intervention.**intervention_rule_list**(**args*, ***kwargs*)
Return the list of intervention rules for named intervention

NB - not documenting in swagger at this time, intended for internal use only. See http://truenth-shared-services.readthedocs.io/en/latest/interventions.html#access

portal.views.intervention.**intervention_rule_set**(**args*, ***kwargs*)
POST an access rule to the named intervention

Submit a JSON doc with the access strategy details to include for the named intervention.

Only available as a service account API - the named intervention must be associated with the service account sponsor.

NB - interventions have a global 'public_access' setting. Only when unset are access rules consulted.

NB - not documenting in swagger at this time, intended for internal use only. See http://truenth-shared-services.readthedocs.io/en/latest/interventions.html#access

Patient view functions (i.e. not part of the API or auth)

portal.views.patients.**patient_profile**(**args*, ***kwargs*)
individual patient view function, intended for staff

portal.views.patients.**patients_root**(**args*, ***kwargs*)
patients view function, intended for staff

Present the logged in staff the list of patients matching the staff's organizations (and any descendant organizations)

Portal view functions (i.e. not part of the API or auth)

class portal.views.portal.**ChallengeIdForm**(*formdata=<object object>*, ***kwargs*)

class portal.views.portal.**SettingsForm**(*formdata=<object object>*, ***kwargs*)

class portal.views.portal.**ShortcutAliasForm**(*formdata=<object object>*, ***kwargs*)

static validate_shortcut_alias(*field*)
Custom validation to confirm an alias match

portal.views.portal.**access_via_token**(*token*, *next_step=None*)
Limited access users enter here with special token as auth

Tokens contain encrypted data including the user_id and timestamp from when it was generated.

If the token is found to be valid, and the user_id isn't associated with a *privilidged* account, the behavior depends on the roles assigned to the token's user_id: * WRITE_ONLY users will be directly logged into the weak auth account * others will be given a chance to prove their identity

Parameters **next_step** – if the user is to be redirected following validation and intial queries, include a value. These come from a controlled vocabulary - see *NextStep*

portal.views.portal.**admin**(**args*, ***kwargs*)
user admin view function

portal.views.portal.**celery_test**(*x=16*, *y=16*)
Simple view to test asynchronous tasks via celery

portal.views.portal.**challenge_identity**(*user_id=None*, *next_url=None*, *merging_accounts=False*, *access_on_verify=False*, *request_path=None*)
Challenge the user to verify themselves

Can't expose the parameters for security reasons - use the session, namespace each variable i.e. session['challenge.user_id'] unless calling as a function.

Parameters

- **user_id** – the user_id to verify - invited user or the like

- **next_url** – destination url on successful challenge completion
- **merging_accounts** – boolean value, set true IFF on success, the user account will be merged into a new account, say from a weak authenicated WRITE_ONLY invite account
- **access_on_verify** – boolean value, set true IFF on success, the user should be logged in once validated, i.e. w/o a password
- **request_path** – the requested url prior to redirection to here necessary in no cookie situations, to redirect user back

portal.views.portal.**communicate**(**args*, ***kwargs*)
Direct call to trigger communications to given user.

Typically handled by scheduled jobs, this API enables testing of communications without the wait.

Include a *force=True* query string parameter to first invalidate the cache and look for fresh messages before triggering the send.

Include a *purge=True* query string parameter to throw out existing communications for the user first, thus forcing a resend (implies a force)

Include a *trace=True* query string parameter to get details found during processing - like a debug trace.

portal.views.portal.**communications_dashboard**(**args*, ***kwargs*)
Communications Dashboard

Displays a list of communication requests from the system; includes a preview mode for specific requests.

portal.views.portal.**contact_sent**(*message_id*)
show invite sent

portal.views.portal.**get_all_tag_data**(**allTags*)
query LR based on all required tags

this is an AND condition; all required tags must be present

Parameters **allTags** – variable number of tags to be queried, e.g., 'tag1', 'tag2'

portal.views.portal.**get_any_tag_data**(**anyTags*)
query LR based on any tags

this is an OR condition; will match any tag specified

Parameters **anyTag** – a variable number of tags to be queried, e.g., 'tag1', 'tag2'

portal.views.portal.**initial_queries**()
Initial consent terms, initial queries view function

portal.views.portal.**invite**(**args*, ***kwargs*)
invite other users via form data

see also /api/user/{user_id}/invite

portal.views.portal.**invite_sent**(**args*, ***kwargs*)
show invite sent

portal.views.portal.**patient_invite_email**(**args*, ***kwargs*)
Patient Invite Email Content

portal.views.portal.**patient_reminder_email**(**args*, ***kwargs*)
Patient Reminder Email Content

portal.views.portal.**preview_communication**(**args*, ***kwargs*)
Communication message preview

portal.views.portal.**profile**(**args*, ***kwargs*)
profile view function

portal.views.portal.**report_error**(**args*, ***kwargs*)
Useful from front end, client-side to raise attention to problems

On occasion, an exception will be generated in the front end code worthy of gaining attention on the server side. By making a GET request here, a server side error will be generated (encouraging the system to handle it as configured, such as by producing error email).

OAuth protected to prevent abuse.

Any of the following query string arguments (and their values) will be included in the exception text, to better capture the context. None are required.

Subject_id User on which action is being attempted

Message Details of the error event

Page_url The page requested resulting in the error

actor_id need not be sent, and will always be included - the OAuth protection guarentees and defines a valid current user.

portal.views.portal.**report_slow_queries**(*response*)
Log slow database queries

This will only function if BOTH values are set in the config: DATABASE_QUERY_TIMEOUT = 0.5 # threshold in seconds SQLALCHEMY_RECORD_QUERIES = True

portal.views.portal.**reporting_dashboard**(**args*, ***kwargs*)
Executive Reporting Dashboard

Only accessible to Admins, or those with the Analyst role (no PHI access).

Usage: graphs showing user registrations and logins per day; filterable by date and/or by intervention

User Stats: counts of users by role, intervention, etc.

Institution Stats: counts of users per org

Analytics: Usage stats from piwik (time on site, geographic usage, referral sources for new visitors, etc)

portal.views.portal.**require_cookies**()
give front end opportunity to verify cookies

Renders HTML including cookie check, then redirects back to *target* NB - query string 'cookies_tested=True' added to target for client to confirm this process happened.

portal.views.portal.**research_dashboard**(**args*, ***kwargs*)
Research Dashboard

Only accessible to those with the Researcher role.

portal.views.portal.**settings**(**args*, ***kwargs*)
settings panel for admins

portal.views.portal.**spec**(**args*, ***kwargs*)
generate swagger friendly docs from code and comments

View function to generate swagger formatted JSON for API documentation. Pulls in a few high level values from the package data (see setup.py) and via flask-swagger, makes use of any yaml comment syntax found in application docstrings.

Point Swagger-UI to this view for rendering

portal.views.portal.**specific_clinic_entry**()
Entry point with form to insert a coded clinic shortcut

Invited users may start here to obtain a specific clinic assignment, by entering the code or shortcut alias they were given.

Store the clinic in the session for association with the user once registered and redirect to the standard landing page.

NB if already logged in - this will bounce user to home

portal.views.portal.**specific_clinic_landing**(*clinic_alias*)
Invited users start here to obtain a specific clinic assignment

Store the clinic in the session for association with the user once registered and redirect to the standard landing page.

portal.views.portal.**stock_consent**(*org_name*)
Simple view to render default consent with named organization

We generally store the unique URL pointing to the content of the agreement to which the user consents. Special case for organizations without a custom consent agreement on file.

Parameters **org_name** – the org_name to include in the agreement text

2.10.5 Open API/Swagger

API endpoints are documented inline, in the function docstring following the Open API (formerly Swagger) specification.

Examples

Schema Reuse

Open API schemas can be defined once and referenced by any other document. For example, the `FHIRPatient schema` defined in the body of one request . . . :

```
operationId: setPatientDemographics
tags:
  - Demographics
produces:
  - application/json
parameters:
  - name: patient_id
    in: path
    description: TrueNTH patient ID
    required: true
    type: integer
    format: int64
  - in: body
    name: body
    schema:
      id: FHIRPatient
      required:
        - resourceType
      properties:
        resourceType:
```

(continues on next page)

(continued from previous page)

```
          type: string
          description: defines FHIR resource type, must be Patient
```

. . . can be referenced in the body of the response:

```
operationId: getPatientDemographics
produces:
  - application/json
parameters:
  - name: patient_id
    in: path
    description:
      Optional TrueNTH patient ID, defaults to the authenticated user.
    required: true
    type: integer
    format: int64
responses:
  200:
    description:
      Returns demographics for requested portal user id as a FHIR
      patient resource (http://www.hl7.org/fhir/patient.html) in JSON.
      Defaults to logged-in user if `patient_id` is not provided.
    schema:
      $ref: "#/definitions/FHIRPatient"
```

2.11 Docker

2.11.1 Background

Docker is an open-source project that can be used to automate the deployment of applications inside software containers. Docker defines specifications and provides tools that can be used to automate building and deploying software containers.

Dockerfiles declaratively define how to build a Docker *image* that is subsequently run as a *container*, any number of times. Configuration in Dockerfiles is primarily driven by image build-time arguments (`ARG`) and environment variables (`ENV`) that may be overridden.

Docker-compose (through `docker-compose.yaml`) defines the relationship (exposed ports, volume mappings) between the Shared Services web container and the other services it depends on (redis, postgresql).

2.11.2 Getting Started

Install *docker-compose* as per environment. For example, from a debian system:

```
# add user to docker group
sudo usermod -aG docker $USER
sudo pip install docker_compose
```

Note: A clean environment and fresh git checkout are recommended, but not required

Copy and edit the default environment file (from the project root):

```
cp docker/portal.env.default docker/portal.env
# update SERVER_NAME to include port if not binding with 80/443
# SERVER_NAME=localhost:8080
```

Note: All docker-compose commands are run from the `docker/` directory

Download and run the `latest` images:

```
docker-compose pull web
docker-compose up web
```

By default, the `truenth_portal` image with the `latest` tag is downloaded and used. To use an image with another tag, set the `DOCKER_IMAGE_TAG` environment variable:

```
export DOCKER_IMAGE_TAG='stable'
docker-compose pull web
docker-compose up web
```

2.11.3 Docker Images

Two Dockerfiles (`Dockerfile.build` and `Dockerfile`) define how to build a docker image capable of creating a Debian package from the portal codebase, and how to install and configure the package into a working Shared Services instance.

Building a Debian Package

To build a Debian package from the current branch of your local repo:

```
# Build debian package from current local branch
docker-compose -f docker-compose.build.yaml run builder
```

If you would like to create a package from a remote repository you can override the local repo as follows below:

```
# Override default with environment variable
export GIT_REPO='https://github.com/USERNAME/true_nth_usa_portal'

# Build the package from the above repo
docker-compose -f docker-compose.build.yaml run builder
```

Building a Shared Services Docker Image

If you would like to build a Shared Services image, follow the instructions in *Building a Debian Package*, and run the following docker-compose commands:

```
# Override default (Artifactory) docker repo to differentiate locally-built images
export DOCKER_REPOSITORY=''

# Build the "web" image locally
docker-compose build web

docker-compose up web
```

2.11.4 Advanced Usage

Running in Background

Docker-compose services can be run in the background by adding the `--detach` option. Services started in detached mode will run until stopped or killed.:

```
# Start the "web" service (and dependencies) in background
docker-compose up --detach web
```

Viewing Logs

Docker-compose will only show logs of the requested services (usually `web`), when not run in the background. To view the logs of all running services:

```
# Tail and follow logs of all services
docker-compose logs --follow

# Tail and follow logs of a specific service
docker-compose logs --follow celerybeat
```

PostgreSQL Access

To interact with the running database container, started via the `docker-compose` instructions above, use `docker exec` as follows below:

```
docker-compose exec db psql --username postgres --dbname portaldb
```

Account Bootstrapping

To bootstrap an admin account after a fresh install, run the below `flask` CLI command:

```
docker-compose exec web \
    flask add-user \
        --email 'admin_email@example.com' \
        --password 'exampleP@$$W0RD' \
        --role admin
```

2.11.5 Advanced Configuration

Environment variables defined in the `portal.env` environment file are only passed to the underlying containers. However, some environment variables are used for configuration specific to docker-compose.

An additional environment file, specifically named `.env`, in the current working directory can define environment variables available through the entire docker-compose file (including containers). These docker-compose-level environment variables can also be set in the shell invoking docker-compose.

One use for environmental variables defined in the `.env` file is overriding the default `COMPOSE_PROJECT_NAME` which can be used to namespace multiple deployments running on the same host. In production deployments `COMPOSE_PROJECT_NAME` is set to correspond to the domain being served.

2.11.6 Continuous Delivery

Our continuous integration setup leverages TravisCI's docker support and deployment integration to create and deploy Debian packages and Docker images for every commit.

Packages and images are built in a separate *job* (named `build-artifacts`) that corresponds with a tox environment that does nothing and that's allowed to fail without delaying the build or affecting its status.

If credentials are configured, packages and images will be uploaded to their corresponding repository after the build process. Otherwise, artifacts will only be built, but not uploaded or deployed.

Currently, our TravisCI setup uses packages locally-built on TravisCI instead of pushing, then pulling from our Debian repository. This may lead to non-deterministic builds and should probably be reconciled at some point, ideally using TravisCI build stages.

Configuration

Most if not all values needed to build and deploy Shared Services are available as environment variables with sane, CIRG-specific defaults. Please see the global section of .travis.yml.

image Docker images are the basis of containers. An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes.

container A container is a runtime instance of a docker image. A Docker container consists of: * A Docker image * Execution environment * A standard set of instructions

environment file A file for defining environment variables. One per line, no shell syntax (export etc).

build A group of TravisCI jobs tied to a single commit; initiated by a pull request or push

job A discrete unit of work that is part of a build. All jobs part of a build must pass for the build to pass (unless a job is set as an allowed failure).

2.12 Contributing

2.12.1 Git Flow Workflow

TrueNTH Shared Services attempts to conform to the guidelines established by the git-flow branching model.

For an introduction, see the excellent git-flow-cheatsheet.

To initialize on a debian system, install the git-flow package:

```
sudo apt-get install git-flow
```

Return to the root of your TrueNTH Shared Services checkout and initialize:

```
cd ~/true_nth_usa_portal
git-flow init
```

You should be able to accept all the defaults (caveat: in some cases "Branch name for production releases: []" won't have a default; in that case, use "master"). The results are written to the nested *.git/config* file, such as:

```
[gitflow "branch"]
        master = master
        develop = develop
[gitflow "prefix"]
        feature = feature/
        release = release/
        hotfix = hotfix/
        support = support/
        versiontag =
```

2.12.2 Work on New Feature

Work on new feature takes place in a fresh branch off of develop. **git-flow** makes this easy:

```
git flow feature start my-feature-name
```

2.12.3 Publish Feature

Once the feature is ready to share, and all changes have been committed locally, push the feature branch to github:

```
git flow feature publish
```

2.12.4 Pull Request

To bring the feature into the main develop branch, head over to github and trigger a **pull request**.

2.12.5 Rebase

Occasionally, it's desirable or even necessary to bring commits on another branch into your feature branch prior to publication.

For example, to bring changes into your branch that have been pushed to *develop* since your feature branch was cut:

```
git checkout develop
git pull
git checkout feature/<my-feature-name>
git flow feature rebase
```

2.13 Testing

2.13.1 Running Unit Tests

See Testing from the README

2.13.2 Debugging Views

A number of endpoints can be used to view details of a patient, or manually trigger an instant reminder, to simplify testing and debugging.

All of these endpoints are restricted by the same rules as any API, namely the authenticated user must have appropriate permissions to make the request, typically governed by user ROLE and shared organizations between the patient and the current user. A user can also view their own data in most cases.

For all of the following, replace the variable name within the angle brackets with the appropriate value.

Communicate

Trigger an immediate lookup and transmission of any assessment reminder emails for a user, rather than wait for the next scheduled job to handle.

Request `/communicate/<patient_id>`

Additional query string parameters supported:

```
trace=True
  Shows details of the lookup process
purge=True
  invalidates the assessment_cache for the patient
  prior to executing the lookup
```

Assessment Status

Request `/api/patient/<patient_id>/assessment-status` to view current assessment status details:

```
assessment_status
  The *overall* status for the patient's assessments.

completed_ids
  A list of the named assessments for the current questionnaire bank which
  the patient has already submitted.

outstanding_indefinite_work
  The ``irondemog`` or ``irondemog3`` assessment is special, belonging to
  the indefinite camp.  If the user is eligible and still needs to complete
  this assessment, this variable will be set to ``1``.

qb_name
  The current Questionnaire Bank for the patient.

questionnaires_ids
  The list of questionnaires the user needs to complete for the current
  Questionnaire Bank (specifically those which haven't been previously
  started and suspended).

resume_ids
  The list of questionnaires the user has begun but not yet completed
  for the current Questionnaire Bank.
```

Additional query string parameters supported:

```
trace=True
  Shows details of the lookup process
```

Invalidate Assessment Cache

Although many URLs listed in this document also support the `purge=True` parameter, it's also possible to invalidate the cached assessment status of any given patient, which will then force a fresh lookup the next time it is needed.

Request `/api/invalidate/<patient_id>` invalidates given user's cache, and returns the patient data in FHIR format.

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V