CommCareHQ Documentation

Release 1.0

Dimagi

May 25, 2017

Contents

Contents:

CHAPTER 1

Reporting

A report is a logical grouping of indicators with common config options (filters etc)

The way reports are produced in CommCare is still evolving so there are a number of different frameworks and methods for generating reports. Some of these are *legacy* frameworks and should not be used for any future reports.

Recommended approaches for building reports

TODO: SQL reports, Elastic reports, Custom case lists / details,

Things to keep in mind:

- report API
- *Fluff*
- sqlagg
- couchdbkit-aggregate (legacy)

Example Custom Report Scaffolding

```
class MyBasicReport(GenericTabularReport, CustomProjectReport):
    name = "My Basic Report"
    slug = "my_basic_report"
    fields = ('corehq.apps.reports.filters.dates.DatespanFilter',)

    @property
    def headers(self):
        return DataTablesHeader(DataTablesColumn("Col A"),
                                DataTablesColumnGroup(
                                    "Group 1",
                                    DataTablesColumn("Col B"),
                                    DataTablesColumn("Col C")),
```

```
                                DataTablesColumn("Col D"))

    @property
    def rows(self):
        return [
            ['Row 1', 2, 3, 4],
            ['Row 2', 3, 2, 1]
        ]
```

Hooking up reports to CommCare HQ

Custom reports can be configured in code or in the database. To configure custom reports in code follow the following instructions.

First, you must add the app to *HQ_APPS* in *settings.py*. It must have an *__init__.py* and a *models.py* for django to recognize it as an app.

Next, add a mapping for your domain(s) to the custom reports module root to the *DOMAIN_MODULE_MAP* variable in *settings.py*.

Finally, add a mapping to your custom reports to *__init__.py* in your custom reports submodule:

```
from myproject import reports

CUSTOM_REPORTS = (
    ('Custom Reports', (
        reports.MyCustomReport,
        reports.AnotherCustomReport,
    )),
)
```

Reporting on data stored in SQL

As described above there are various ways of getting reporting data into and SQL database. From there we can query the data in a number of ways.

Extending the `SqlData` class

The `SqlData` class allows you to define how to query the data in a declarative manner by breaking down a query into a number of components.

This approach means you don't write any raw SQL. It also allows you to easily include or exclude columns, format column values and combine values from different query columns into a single report column (e.g. calculate percentages).

In cases where some columns may have different filter values e.g. males vs females, **sqlagg** will handle executing the different queries and combining the results.

This class also implements the `corehq.apps.reports.api.ReportDataSource`.

See *Report API* and sqlagg for more info.

e.g.

```
class DemoReport(SqlTabularReport, CustomProjectReport):
    name = "SQL Demo"
    slug = "sql_demo"
    fields = ('corehq.apps.reports.filters.dates.DatespanFilter',)

    # The columns to include the the 'group by' clause
    group_by = ["user"]

    # The table to run the query against
    table_name = "user_report_data"

    @property
    def filters(self):
        return [
            BETWEEN('date', 'startdate', 'enddate'),
        ]

    @property
    def filter_values(self):
        return {
            "startdate": self.datespan.startdate_param_utc,
            "enddate": self.datespan.enddate_param_utc,
            "male": 'M',
            "female": 'F',
        }

    @property
    def keys(self):
        # would normally be loaded from couch
        return [["user1"], ["user2"], ['user3']]

    @property
    def columns(self):
        return [
            DatabaseColumn("Location", SimpleColumn("user_id"), format_fn=self.
↪username),
            DatabaseColumn("Males", CountColumn("gender"), filters=self.filters+[EQ(
↪'gender', 'male')]),
            DatabaseColumn("Females", CountColumn("gender"), filters=self.filters+[EQ(
↪'gender', 'female')]),
            AggregateColumn(
                "C as percent of D",
                self.calc_percentage,
                [SumColumn("indicator_c"), SumColumn("indicator_d")],
                format_fn=self.format_percent)
        ]

    _usernames = {"user1": "Location1", "user2": "Location2", 'user3': "Location3"}
↪# normally loaded from couch
    def username(self, key):
        return self._usernames[key]

    def calc_percentage(num, denom):
        if isinstance(num, Number) and isinstance(denom, Number):
            if denom != 0:
                return num * 100 / denom
            else:
                return 0
```

```
        else:
            return None

    def format_percent(self, value):
        return format_datatables_data("%d%%" % value, value)
```

Using the sqlalchemy API directly

TODO

Report API

Part of the evolution of the reporting frameworks has been the development of a *report api*. This is essentially just a change in the architecture of reports to separate the data from the display. The data can be produced in various formats but the most common is an list of dicts.

e.g.

```
data = [
  {
    'slug1': 'abc',
    'slug2': 2
  },
  {
    'slug1': 'def',
    'slug2': 1
  }
  ...
]
```

This is implemented by creating a report data source class that extends `corehq.apps.reports.api.ReportDataSource` and overriding the `get_data()` function.

These data sources can then be used independently or the CommCare reporting user interface and can also be reused for multiple use cases such as displaying the data in the CommCare UI as a table, displaying it in a map, making it available via HTTP etc.

An extension of this base data source class is the `corehq.apps.reports.sqlreport.SqlData` class which simplifies creating data sources that get data by running an SQL query. See section on *SQL reporting* for more info.

e.g.

```
class CustomReportDataSource(ReportDataSource):
    def get_data(self):
        startdate = self.config['start']
        enddate = self.config['end']

        ...

        return data

config = {'start': date(2013, 1, 1), 'end': date(2013, 5, 1)}
ds = CustomReportDataSource(config)
data = ds.get_data()
```

Adding dynamic reports

Domains support dynamic reports. Currently the only verison of these are maps reports. There is currently no documentation for how to use maps reports. However you can look at the *drew* or *aaharsneha* domains on prod for examples.

How pillow/fluff work

GitHub

Note: This should be rewritten, I wrote it when I was first trying to understand how fluff works.

A Pillow provides the ability to listen to a database, and on changes, the class *BasicPillow* calls change_transform and passes it the changed doc dict. This method can process the dict and transform it, or not. The result is then passed to the method `change_transport`, which must be implemented in any subclass of `BasicPillow`. This method is responsible for acting upon the changes.

In fluff's case, it stores an indicator document with some data calculated from a particular type of doc. When a relevant doc is updated, the calculations are performed. The diff between the old and new indicator docs is calculated, and sent to the db to update the indicator doc.

fluff's *Calculator* object auto-detects all methods that are decorated by subclasses of *base_emitter* and stores them in a *_fluff_emitters* array. This is used by the *calculate* method to return a dict of emitter slugs mapped to the result of the emitter function (called with the newly updated doc) coerced to a list.

to rephrase: fluff emitters accept a doc and return a generator where each element corresponds to a contribution to the indicator

CHAPTER 2

Change Feeds

The following describes our approach to change feeds on HQ. For related content see Cory's brown bag on the topic

What they are

A change feed is modeled after the CouchDB `_changes` feed. It can be thought of as a real-time log of "changes" to our database. Anything that creates such a log is called a "(change) publisher".

Other processes can listen to a change feed and then do something with the results. Processes that listen to changes are called "subscribers". In the HQ codebase "subscribers" are referred to as "pillows" and most of the change feed functionality is provided via the pillowtop module. This document refers to pillows and subscribers interchangeably.

Common use cases for change subscribers:

- **ETL (our main use case)**
  - Saving docs to ElasticSearch
  - Custom report tables
  - UCR data sources
- Cache invalidation
- Real-time visualizations (e.g. dimagisphere)

Architecture

We use kafka as our primary back-end to facilitate change feeds. This allows us to decouple our subscribers from the underlying source of changes so that they can be database-agnostic. For legacy reasons there are still change feeds that run off of CouchDB's `_changes` feed however these are in the process of being phased out.

Topics

Topics are a kafka concept that are used to create logical groups (or "topics") of data. In the HQ codebase we use topics primarily as a 1:N mapping to HQ document classes (or ‘‘doc_type‘‘s). Forms and cases currently have their own topics, while everything else is lumped in to a "meta" topic. This allows certain pillows to subscribe to the exact category of change/data they are interested in (e.g. a pillow that sends cases to elasticsearch would only subscribe to the "cases" topic).

Document Stores

Published changes are just "stubs" but do not contain the full data that was affected. Each change should be associated with a "document store" which is an abstraction that represents a way to retrieve the document from its original database. This allows the subscribers to retrieve the full document while not needing to have the underlying source hard-coded (so that it can be changed). To add a new document store, you can use one of the existing subclasses of `DocumentStore` or roll your own.

Publishing changes

Publishing changes is the act of putting them into kafka from somewhere else.

From Couch

Publishing changes from couch is easy since couch already has a great change feed implementation with the `_changes` API. For any database that you want to publish changes from the steps are very simple. Just create a `ConstructedPillow` with a `CouchChangeFeed` feed pointed at the database you wish to publish from and a `KafkaProcessor` to publish the changes. There is a utility function (`get_change_feed_pillow_for_db`) which creates this pillow object for you.

From SQL

Currently SQL-based change feeds are published from the app layer. Basically, you can just call a function that publishes the change in a `.save()` function (or a `post_save` signal). See the functions in form_processors.change_publishers and their usages for an example of how that's done.

It is planned (though unclear on what timeline) to find an option to publish changes directly from SQL to kafka to avoid race conditions and other issues with doing it at the app layer. However, this change can be rolled out independently at any time in the future with (hopefully) zero impact to change subscribers.

From anywhere else

There is not yet a need/precedent for publishing changes from anywhere else, but it can always be done at the app layer.

Subscribing to changes

It is recommended that all new change subscribers be instances (or subclasses) of `ConstructedPillow`. You can use the `KafkaChangeFeed` object as the change provider for that pillow, and configure it to subscribe to one or more topics. Look at usages of the `ConstructedPillow` class for examples on how this is done.

Porting a new pillow

Porting a new pillow to kafka will typically involve the following steps. Depending on the data being published, some of these may be able to be skipped (e.g. if there is already a publisher for the source data, then that can be skipped).

1. Setup a publisher, following the instructions above.
2. Setup a subscriber, following the instructions above.
3. For non-couch-based data sources, you must setup a `DocumentStore` class for the pillow, and include it in the published feed.
4. For any pillows that require additional bootstrap logic (e.g. setting up UCR data tables or bootstrapping elasticsearch indexes) this must be hooked up manually.

CHAPTER 3

Pillows

What they are

A pillow is a subscriber to a change feed. When a change is published the pillow receives the document, performs some calculation or transform, and publishes it to another database.

Creating a pillow

All pillows inherit from *ConstructedPillow* class. A pillow consists of a few parts:

1. Change Feed
2. Checkpoint
3. Processor(s)
4. Change Event Handler

Change Feed

Change feeds are documented here.

The 10,000 foot view is a change feed publishes changes which you can subscribe to.

Checkpoint

The checkpoint is a json field that tells processor where to start the change feed.

Processor(s)

A processor is what handles the transformation or calculation and publishes it to a database. Most pillows only have one processor, but sometimes it will make sense to combine processors into one pillow when you are only iterating over a small number of documents (such as custom reports).

When creating a processor you should be aware of how much time it will take to process the record. A useful baseline is:

86400 seconds per day / # of expected changes per day = how long your processor should take

Note that it should be faster than this as most changes will come in at once instead of evenly distributed throughout the day.

Change Event Handler

This fires after each change has been processed. The main use case is to save the checkpoint to the database.

Error Handling

Pillow errors are handled by saving to model *PillowError*. A celery queue reads from this model and retries any errors on the pillow.

Monitoring

There are several datadog metrics with the prefix *commcare.change_feed* that can be helpful for monitoring pillows.

For UCR pillows the pillow log will contain any data sources and docs that have exceeded a threshold and can be used to find expensive data sources.

Troubleshooting

A pillow is falling behind

A pillow can fall behind for two reasons:

1. The processor is too slow for the number of changes that are coming in.
2. There has been an issue with the change feed that has caused the checkpoint to be "rewound"

Optimizing a processor

To solve #1 you should use any monitors that have been set up to attempt to pinpoint the issue.

If this is a UCR pillow use the *profile_data_source* management command to profile the expensive data sources.

Parallel Processors

The UCR pillows currently have options to split the pillow into multiple. They include *ucr_divsion*, *include_ucrs* and *exclude_ucrs*. Look to the pillow code for more information on these.

Soon it will be possible to add partitions to kafka topics, which will be the preferred way to horizontally scale pillows.

Rewound Checkpoint

Occasionally checkpoints will be "rewound" to a previous state causing pillows to process changes that have already been processed. This usually happens when a couch node fails over to another. If this occurs, stop the pillow, wait for confirmation that the couch nodes are up, and fix the checkpoint using: *./manage.py fix_checkpoint_after_rewind <pillow_name>*

Problem with checkpoint for pillow name: First available topic offset for topic is num1 but needed num2

This happens when the earliest checkpoint that kafka knows about for a topic is after the checkpoint the pillow wants to start at. This often happens if a pillow has been stopped for a month and has not been removed from the settings.

To fix this you should verify that the pillow is no longer needed in the environment. If it isn't, you can delete the checkpoint and re-deploy. This should eventually be followed up by removing the pillow from the settings.

If the pillow is needed and should be running you're in a bit of a pickle. This means that the pillow is not able to get the required document ids from kafka. It also won't be clear what documents the pillows has and has not processed. To fix this the safest thing will be to force the pillow to go through all relevant docs. Once this process is started you can move the checkpoint for that pillow to the most recent offset for its topic.

CHAPTER 4

API

Bulk User Resource

Resource name: `bulk_user`
First version available: `v0.5`

This resource is used to get basic user data in bulk, fast. This is especially useful if you need to get, say, the name and phone number of every user in your domain for a widget.

Currently the default fields returned are:

```
id
email
username
first_name
last_name
phone_numbers
```

Supported Parameters:

- `q` - query string
- `limit` - maximum number of results returned
- `offset` - Use with `limit` to paginate results
- `fields` - restrict the fields returned to a specified set

Example query string:

```
?q=foo&fields=username&fields=first_name&fields=last_name&limit=100&offset=200
```

This will return the first and last names and usernames for users matching the query "foo". This request is for the third page of results (200-300)

Additional notes:

It is simple to add more fields if there arises a significant use case.

Potential future plans: Support filtering in addition to querying. Support different types of querying. Add an order_by option

CHAPTER 5

Reporting: Maps in HQ

What is the "Maps Report"?

We now have map-based reports in HQ. The "maps report" is not really a report, in the sense that it does not query or calculate any data on its own. Rather, it's a generic front-end visualization tool that consumes data from some other place... other places such as another (tabular) report, or case/form data (work in progress).

To create a map-based report, you must configure the map report template with specific parameters. These are:

- `data_source` – the backend data source which will power the report (required)
- `display_config` – customizations to the display/behavior of the map itself (optional, but suggested for anything other than quick prototyping)

There are two options for how this configuration actually takes place:

- via a domain's "dynamic reports" (see *Adding dynamic reports*), where you can create specific configurations of a generic report for a domain
- subclass the map report to provide/generate the config parameters. You should **not** need to subclass any code functionality. This is useful for making a more permanent map configuration, and when the configuration needs to be dynamically generated based on other data or domain config (e.g., for CommTrack)

Orientation

Abstractly, the map report consumes a table of data from some source. Each row of the table is a geographical feature (point or region). One column is identified as containing the geographical data for the feature. All other columns are arbitrary attributes of that feature that can be visualized on the map. Another column may indicate the name of the feature.

The map report contains, obviously, a map. Features are displayed on the map, and may be styled in a number of ways based on feature attributes. The map also contains a legend generated for the current styling. Below the map is a table showing the raw data. Clicking on a feature or its corresponding row in the table will open a detail popup. The columns shown in the table and the detail popup can be customized.

Attribute data is generally treated as either being numeric data or enumerated data (i.e., belonging to a number of discrete categories). Strings are inherently treated as enum data. Numeric data can be treated as enum data be specifying thresholds: numbers will be mapped to enum 'buckets' between consecutive thresholds (e.g, thresholds of `10`, `20` will create enum categories: `< 10`, `10-20`, `> 20`).

Styling

Different aspects of a feature's marker on the map can be styled based on its attributes. Currently supported visualizations (you may see these referred to in the code as "display axes" or "display dimensions") are:

- varying the size (numeric data only)
- varying the color/intensity (numeric data (color scale) or enum data (fixed color palette))
- selecting an icon (enum data only)

Size and color may be used concurrently, so one attribute could vary size while another varies the color... this is useful when the size represents an absolute magnitude (e.g., # of pregnancies) while the color represents a ratio (% with complications). Region features (as opposed to point features) only support varying color.

A particular configuration of visualizations (which attributes are mapped to which display axes, and associated styling like scaling, colors, icons, thresholds, etc.) is called a *metric*. A map report can be configured with many different metrics. The user selects one metric at a time for viewing. *Metrics may not correspond to table columns one-to-one*, as a single column may be visualized multiple ways, or in combination with other columns, or not at all (shown in detail popup only). If no metrics are specified, they will be auto-generated from best guesses based on the available columns and data feeding the report.

There are several sample reports that comprehensively demo the potential styling options:

- Demo 1
- Demo 2

See *Display Configuration*

Data Sources

Set this config on the `data_source` property. It should be a `dict` with the following properties:

- `geo_column` – the column in the returned data that contains the geo point (default: `"geo"`)
- `adapter` – which data adapter to use (one of the choices below)
- extra arguments specific to each data adapter

Note that any report filters in the map report are passed on verbatim to the backing data source.

One column of the data returned by the data source must be the geodata (in `geo_column`). For point features, this can be in the format of a geopoint xform question (e.g, `42.366 -71.104`). The geodata format for region features is outside the scope of the document.

`report`

Retrieve data from a `ReportDataSource` (the abstract data provider of Simon's new reporting framework – see *Report API*)

Parameters:

- `report` – fully qualified name of `ReportDataSource` class
- `report_params` – `dict` of static config parameters for the `ReportDataSource` (optional)

`legacyreport`

Retrieve data from a `GenericTabularReport` which has not yet been refactored to use Simon's new framework. *Not ideal* and should only be used for backwards compatibility. Tabular reports tend to return pre-formatted data, while the maps report works best with raw data (for example, it won't know `4%` or `30 mg` are numeric data, and will instead treat them as text enum values). *Read more*.

Parameters:

- `report` – fully qualified name of tabular report view class (descends from `GenericTabularReport`)
- `report_params` – `dict` of static config parameters for the `ReportDataSource` (optional)

`case`

Pull case data similar to the Case List.

(In the current implementation, you must use the same report filters as on the regular Case List report)

Parameters:

- `geo_fetch` – a mapping of case types to directives of how to pull geo data for a case of that type. Supported directives:
  - name of case property containing the `geopoint` data
  - `"link:xxx"` where `xxx` is the case type of a linked case; the adapter will then serach that linked case for geo-data based on the directive of the linked case type *(not supported yet)*

  In the absence of any directive, the adapter will first search any linked `Location` record *(not supported yet)*, then try the `gps` case property.

`csv` and `geojson`

Retrieve static data from a csv or geojson file on the server (only useful for testing/demo– this powers the demo reports, for example).

Display Configuration

Set this config on the `display_config` property. It should be a `dict` with the following properties:

(Whenever 'column' is mentioned, it refers to a column slug as returned by the data adapter)

All properties are optional. The map will attempt sensible defaults.

- `name_column` – column containing the name of the row; used as the header of the detail popup
- `column_titles` – a mapping of columns to display titles for each column
- `detail_columns` – a list of columns to display in the detail popup
- `table_columns` – a list of columns to display in the data table below the map

- `enum_captions` – display captions for enumerated values. A `dict` where each key is a column and each value is another `dict` mapping enum values to display captions. These enum values reflect the results of any transformations from `metrics` (including `_other`, `_null`, and `-`).
- `numeric_format` – a mapping of columns to functions that apply the appropriate numerical formatting for that column. Expressed as the body of a function that returns the formatted value (`return` statement required!). The unformatted value is passed to the function as the variable `x`.
- `detail_template` – an underscore.js template to format the content of the detail popup
- `metrics` – define visualization metrics (see *Styling*). An array of metrics, where each metric is a `dict` like so:
  - `auto` – column. Auto-generate a metric for this column with no additional manual input. Uses heuristics to determine best presentation format.

  OR

  - `title` – metric title in sidebar (optional)

  AND one of the following for each visualization property you want to control

  - `size` (static) – set the size of the marker (radius in pixels)
  - `size` (dynamic) – vary the size of the marker dynamically. A dict in the format:
    - `column` – column whose data to vary by
    - `baseline` – value that should correspond to a marker radius of 10px
    - `min` – min marker radius (optional)
    - `max` – max marker radius (optional)
  - `color` (static) – set the marker color (css color value)
  - `color` (dynamic) – vary the color of the marker dynamically. A dict in the format:
    - `column` – column whose data to vary by
    - `categories` – for enumerated data; a mapping of enum values to css color values. Mapping key may also be one of these magic values:
      - `_other`: a catch-all for any value not specified
      - `_null`: matches rows whose value is blank; if absent, such rows will be hidden
    - `colorstops` – for numeric data. Creates a sliding color scale. An array of colorstops, each of the format `[<value>, <css color>]`.
    - `thresholds` – (optional) a helper to convert numerical data into enum data via "buckets". Specify a list of thresholds. Each bucket comprises a range from one threshold up to but not including the next threshold. Values are mapped to the bucket whose range they lie in. The "name" (i.e., enum value) of a bucket is its lower threshold. Values below the lowest threshold are mapped to a special bucket called `"-"`.
  - `icon` (static) – set the marker icon (image url)
  - `icon` (dynamic) – vary the icon of the marker dynamically. A dict in the format:
    - `column` – column whose data to vary by
    - `categories` – as in `color`, a mapping of enum values to icon urls
    - `thresholds` – as in `color`

`size` and `color` may be combined (such as one column controlling size while another controls the color). `icon` must be used on its own.

For date columns, any relevant number in the above config (`thresholds`, `colorstops`, etc.) may be replaced with a date (in ISO format).

Raw vs. Formatted Data

Consider the difference between raw and formatted data. Numbers may be formatted for readability (`12,345,678`, `62.5%`, `27 units`); enums may be converted to human-friendly captions; null values may be represented as `--` or `n/a`. The maps report works best when it has the raw data and can perform these conversions itself. The main reason is so that it may generate useful legends, which requires the ability to appropriately format values that may never appear in the report data itself.

There are three scenarios of how a data source may provide data:

- *(worst)* only provide formatted data

  Maps report cannot distinguish numbers from strings from nulls. Data visualizations will not be useful.

- *(sub-optimal)* provide both raw and formatted data (most likely via the `legacyreport` adapter)

  Formatted data will be shown to the user, but maps report will not know how to format data for display in legends, nor will it know all possible values for an enum field – only those that appear in the data.

- *(best)* provide raw data, and explicitly define enum lists and formatting functions in the report config

CHAPTER 6

UI Helpers

There are a few useful UI helpers in our codebase which you should be aware of. Save time and create consistency.

Paginated CRUD View

Use *corehq.apps.hqwebapp.views.CRUDPaginatedViewMixin* the with a *TemplateView* subclass (ideally one that also subclasses *corehq.apps.hqwebapp.views.BasePageView* or *BaseSectionPageView*) to have a paginated list of objects which you can create, update, or delete.

The Basic Paginated View

In its very basic form (a simple paginated view) it should look like:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    # your template should extend style/base_paginated_crud.html
    template_name = 'puppyapp/paginated_puppies.html

    # all the user-visible text
    limit_text = "puppies per page"
    empty_notification = "you have no puppies"
    loading_messagge = "loading_puppies"

    # required properties you must implement:

    @property
    def parameters(self):
        """
        Specify a GET or POST from an HttpRequest object.
        """
        # Usually, something like:
        return self.request.POST if self.request.method == 'post' else self.request.
↪GET
```

```
    @property
    def total(self):
        # How many documents are you paginating through?
        return Puppy.get_total()

    @property
    def column_names(self):
        # What will your row be displaying?
        return [
            "Name",
            "Breed",
            "Age",
        ]

    @property
    def paginated_list(self):
        """
        This should return a list (or generator object) of data formatted as follows:
        [
            {
                'itemData': {
                    'id': <id of item>,
                    <json dict of item data for the knockout model to use>
                },
                'template': <knockout template id>
            }
        ]
        """
        for puppy in Puppy.get_all():
            yield {
                'itemData': {
                    'id': puppy._id,
                    'name': puppy.name,
                    'breed': puppy.breed,
                    'age': puppy.age,
                },
                'template': 'base-puppy-template',
            }

    def post(self, *args, **kwargs):
        return self.paginated_crud_response
```

The template should use knockout templates to render the data you pass back to the view. Each template will have access to everything inside of *itemData*. Here's an example:

```
{% extends 'style/base_paginated_crud.html' %}

{% block pagination_templates %}
<script type="text/html" id="base-puppy-template">
    <td data-bind="text: name"></td>
    <td data-bind="text: breed"></td>
    <td data-bind="text: age"></td>
</script>
{% endblock %}
```

Allowing Creation in your Paginated View

If you want to create data with your paginated view, you must implement the following:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...
    def get_create_form(self, is_blank=False):
        if self.request.method == 'POST' and not is_blank:
            return CreatePuppyForm(self.request.POST)
        return CreatePuppyForm()

    def get_create_item_data(self, create_form):
        new_puppy = create_form.get_new_puppy()
        return {
            'newItem': {
                'id': new_puppy._id,
                'name': new_puppy.name,
                'breed': new_puppy.breed,
                'age': new_puppy.age,
            },
            # you could use base-puppy-template here, but you might want to add an␣
↪update button to the
            # base template.
            'template': 'new-puppy-template',
        }
```

The form returned in *get_create_form()* should make use of crispy forms.

```
from django import forms
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Layout
from crispy_forms.bootstrap import StrictButton, InlineField

class CreatePuppyForm(forms.Form):
    name = forms.CharField()
    breed = forms.CharField()
    dob = forms.DateField()

    def __init__(self, *args, **kwargs):
        super(CreatePuppyForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
        self.helper.form_style = 'inline'
        self.helper.form_show_labels = False
        self.helper.layout = Layout(
            InlineField('name'),
            InlineField('breed'),
            InlineField('dob'),
            StrictButton(
                mark_safe('<i class="icon-plus"></i> %s' % "Create Puppy"),
                css_class='btn-success',
                type='submit'
            )
        )

    def get_new_puppy(self):
        # return new Puppy
        return Puppy.create(self.cleaned_data)
```

Allowing Updating in your Paginated View

If you want to update data with your paginated view, you must implement the following:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...
    def get_update_form(self, initial_data=None):
        if self.request.method == 'POST' and self.action == 'update':
            return UpdatePuppyForm(self.request.POST)
        return UpdatePuppyForm(initial=initial_data)

    @property
    def paginated_list(self):
        for puppy in Puppy.get_all():
            yield {
                'itemData': {
                    'id': puppy._id,
                    ...
                    # make sure you add in this line, so you can use the form in your␣
↪template:
                    'updateForm': self.get_update_form_response(
                        self.get_update_form(puppy.inital_form_data)
                    ),
                },
                'template': 'base-puppy-template',
            }

    @property
    def column_names(self):
        return [
            ...
            # if you're adding another column to your template, be sure to give it a␣
↪name here...
            _('Action'),
        ]

    def get_updated_item_data(self, update_form):
        updated_puppy = update_form.update_puppy()
        return {
            'itemData': {
                'id': updated_puppy._id,
                'name': updated_puppy.name,
                'breed': updated_puppy.breed,
                'age': updated_puppy.age,
            },
            'template': 'base-puppy-template',
        }
```

The *UpdatePuppyForm* should look something like:

```
class UpdatePuppyForm(CreatePuppyForm):
    item_id = forms.CharField(widget=forms.HiddenInput())

    def __init__(self, *args, **kwargs):
        super(UpdatePuppyForm, self).__init__(*args, **kwargs)
        self.helper.form_style = 'default'
        self.helper.form_show_labels = True
        self.helper.layout = Layout(
```

```
            Div(
                Field('item_id'),
                Field('name'),
                Field('breed'),
                Field('dob'),
                css_class='modal-body'
            ),
            FormActions(
                StrictButton(
                    "Update Puppy",
                    css_class='btn-primary',
                    type='submit',
                ),
                HTML('<button type="button" class="btn" data-dismiss="modal">Cancel</
↪button>'),
                css_class="modal-footer"
            )
        )

    def update_puppy(self):
        return Puppy.update_puppy(self.cleaned_data)
```

You should add the following to your *base-puppy-template* knockout template:

```
<script type="text/html" id="base-puppy-template">
    ...
    <td> <!-- actions -->
        <button type="button"
                data-toggle="modal"
                data-bind="
                    attr: {
                        'data-target': '#update-puppy-' + id
                    }
                "
                class="btn btn-primary">
            Update Puppy
        </button>

        <div class="modal hide fade"
             data-bind="
                attr: {
                    id: 'update-puppy-' + id
                }
             ">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal" aria-hidden=
↪"true">&times;</button>
                <h3>
                    Update puppy <strong data-bind="text: name"></strong>:
                </h3>
            </div>
            <div data-bind="html: updateForm"></div>
        </div>
    </td>
</script>
```

Allowing Deleting in your Paginated View

If you want to delete data with your paginated view, you should implement something like the following:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...

    def get_deleted_item_data(self, item_id):
        deleted_puppy = Puppy.get(item_id)
        deleted_puppy.delete()
        return {
            'itemData': {
                'id': deleted_puppy._id,
                ...
            },
            'template': 'deleted-puppy-template',  # don't forget to implement this!
        }
```

You should add the following to your *base-puppy-template* knockout template:

```
<script type="text/html" id="base-puppy-template">
    ...
    <td> <!-- actions -->
        ...
        <button type="button"
                data-toggle="modal"
                data-bind="
                    attr: {
                        'data-target': '#delete-puppy-' + id
                    }
                "
                class="btn btn-danger">
            <i class="icon-remove"></i> Delete Puppy
        </button>

        <div class="modal hide fade"
             data-bind="
                attr: {
                    id: 'delete-puppy-' + id
                }
             ">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal" aria-hidden=
↪"true">&times;</button>
                <h3>
                   Delete puppy <strong data-bind="text: name"></strong>?
                </h3>
            </div>
            <div class="modal-body">
                <p>
                    Yes, delete the puppy named <strong data-bind="text: name"></
↪strong>.
                </p>
            </div>
            <div class="modal-footer">
                <button type="button"
                        class="btn"
                        data-dismiss="modal">
```

```
                    Cancel
                </button>
                <button type="button"
                        class="btn btn-danger delete-item-confirm"
                        data-loading-text="Deleting Puppy...">
                    <i class="icon-remove"></i> Delete Puppy
                </button>
            </div>
        </div>
    </td>
</script>
```

Refreshing The Whole List Base on Update

If you want to do something that affects an item's position in the list (generally, moving it to the top), this is the feature you want.

You implement the following method (note that a return is not expected):

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...

    def refresh_item(self, item_id):
        # refresh the item here
        puppy = Puppy.get(item_id)
        puppy.make_default()
        puppy.save()
```

Add a button like this to your template:

```
<button type="button"
        class="btn refresh-list-confirm"
        data-loading-text="Making Default...">
    Make Default Puppy
</button>
```

Now go on and make some CRUD paginated views!

CHAPTER 7

Using Class-Based Views in CommCare HQ

We should move away from function-based views in django and use class-based views instead. The goal of this section is to point out the infrastructure we've already set up to keep the UI standardized.

The Base Classes

There are two styles of pages in CommCare HQ. One page is centered (e.g. registration, org settings or the list of projects). The other is a two column, with the left gray column acting as navigation and the right column displaying the primary content (pages under major sections like reports).

A Basic (Centered) Page

To get started, subclass *BasePageView* in *corehq.apps.hqwebapp.views*. *BasePageView* is a subclass of django's *TemplateView*.

```
class MyCenteredPage(BasePageView):
    urlname = 'my_centered_page'
    page_title = "My Centered Page"
    template_name = 'path/to/template.html'

    @property
    def page_url(self):
        # often this looks like:
        return reverse(self.urlname)

    @property
    def page_context(self):
        # You want to do as little logic here.
        # Better to divvy up logical parts of your view in other instance methods or␣
↪properties
        # to keep things clean.
        # You can also do stuff in the get() and post() methods.
```

```
        return {
            'some_property': self.compute_my_property(),
            'my_form': self.centered_form,
        }
```

urlname This is what django urls uses to identify your page

page_title This text will show up in the <*title*> tag of your template. It will also show up in the primary heading of your template.

If you want to do use a property in that title that would only be available after your page is instantiated, you should override:

```
@property
def page_name(self):
    return mark_safe("This is a page for <strong>%s</strong>" % self.kitten.name)
```

page_name will not show up in the <*title*> tags, as you can include html in this name.

template_name Your template should extend *style/base_page.html*

It might look something like:

```
{% extends 'style/base_page.html' %}

{% block js %}{{ block.super }}
    {# some javascript imports #}
{% endblock %}

{% block js-inline %}{{ block.super }}
    {# some inline javascript #}
{% endblock %}

{% block page_content %}
    My page content! Woo!
{% endblock %}

{% block modals %}{{ block.super }}
    {# a great place to put modals #}
{% endblock %}
```

A Section (Two-Column) Page

To get started, subclass *BaseSectionPageView* in *corehq.apps.hqwebapp.views*. You should implement all the things described in the minimal setup for *A Basic (Centered) Page* in addition to:

```
class MySectionPage(BaseSectionPageView):
    ...  # everything from BasePageView

    section_name = "Data"
    template_name = 'my_app/path/to/template.html'

    @property
    def section_url(self):
        return reverse('my_section_default')
```

Note: Domain Views

If your view uses *domain*, you should subclass *BaseDomainView*. This inserts the domain name as into the *main_context* and adds the *login_and_domain_required* permission. It also implements *page_url* to assume the basic *reverse* for a page in a project: *reverse(self.urlname, args=[self.domain])*

section_name This shows up as the root name on the section breadcrumbs.

template_name Your template should extend *style/base_section.html*

It might look something like:

```
{% extends 'style/base_section.html' %}

{% block js %}{{ block.super }}
    {# some javascript imports #}
{% endblock %}

{% block js-inline %}{{ block.super }}
    {# some inline javascript #}
{% endblock %}

{% block main_column %}
    My page content! Woo!
{% endblock %}

{% block modals %}{{ block.super }}
    {# a great place to put modals #}
{% endblock %}
```

Note: Organizing Section Templates

Currently, the practice is to extend *style/base_section.html* in a base template for your section (e.g. *users/base_template.html*) and your section page will then extend its section's base template.

Adding to Urlpatterns

Your *urlpatterns* should look something like:

```
urlpatterns = patterns(
    'corehq.apps.my_app.views',
    ...,
    url(r'^my/page/path/$', MyCenteredPage.as_view(), name=MyCenteredPage.urlname),
)
```

Hierarchy

If you have a hierarchy of pages, you can implement the following in your class:

```
class MyCenteredPage(BasePageView):
    ...

    @property
    def parent_pages(self):
```

```
        # This will show up in breadcrumbs as MyParentPage > MyNextPage >␣
↪MyCenteredPage
        return [
            {
                'title': MyParentPage.page_title,
                'url': reverse(MyParentPage.urlname),
            },
            {
                'title': MyNextPage.page_title,
                'url': reverse(MyNextPage.urlname),
            },
        ]
```

If you have a hierarchy of pages, it might be wise to implement a *BaseParentPageView* or *Base<InsertSectionName>View* that extends the *main_context* property. That way all of the pages in that section have access to the section's context. All page-specific context should go in *page_context*.

```
class BaseKittenSectionView(BaseSectionPageView):

    @property
    def main_context(self):
        main_context = super(BaseParentView, self).main_context
        main_context.update({
            'kitten': self.kitten,
        })
        return main_context
```

Permissions

To add permissions decorators to a class-based view, you need to decorate the *dispatch* instance method.

```
class MySectionPage(BaseSectionPageView):
    ...

    @method_decorator(can_edit)
    def dispatch(self, request, *args, **kwargs)
        return super(MySectionPage, self).dispatch(request, *args, **kwargs)
```

GETs and POSTs (and other http methods)

Depending on the type of request, you might want to do different things.

```
class MySectionPage(BaseSectionPageView):
    ...

    def get(self, request, *args, **kwargs):
        # do stuff related to GET here...
        return super(MySectionPage, self).get(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        # do stuff related to post here...
        return self.get(request, *args, **kwargs)  # or any other HttpResponse object
```

Limiting HTTP Methods

If you want to limit the HTTP request types to just GET or POST, you just have to override the *http_method_names* class property:

```
class MySectionPage(BaseSectionPageView):
    ...
    http_method_names = ['post']
```

Note: Other Allowed Methods

put, *delete*, *head*, *options*, and *trace* are all allowed methods by default.

CHAPTER 8

Testing best practices

Test set up

Doing a lot of work in the `setUp` call of a test class means that it will be run on every test. This quickly adds a lot of run time to the tests. Some things that can be easily moved to `setUpClass` are domain creation, user creation, or any other static models needed for the test.

Sometimes classes share the same base class and inherit the `setUpClass` function. Below is an example:

```
# BAD EXAMPLE

class MyBaseTestClass(TestCase):

    @classmethod
    def setUpClass(cls):
        ...


class MyTestClass(MyBaseTestClass):

    def test1(self):
        ...

class MyTestClassTwo(MyBaseTestClass):

    def test2(self):
        ...
```

In the above example the `setUpClass` is run twice, once for `MyTestClass` and once for `MyTestClassTwo`. If `setUpClass` has expensive operations, then it's best for all the tests to be combined under one test class.

```
# GOOD EXAMPLE

class MyBigTestClass(TestCase):
```

```
    @classmethod
    def setUpClass(cls):
        ...

    def test1(self):
        ...

    def test2(self):
        ...
```

However this can lead to giant Test classes. If you find that all the tests in a package or module are sharing the same set up, you can write a setup method for the entire package or module. More information on that can be found here.

Test tear down

It is important to ensure that all objects you have created in the test database are deleted when the test class finishes running. This often happens in the `tearDown` method or the `tearDownClass` method. However, unneccessary cleanup "just to be safe" can add a large amount of time onto your tests.

Using SimpleTestCase

The SimpleTestCase runs tests without a database. Many times this can be achieved through the use of the mock library. A good rule of thumb is to have 80% of your tests be unit tests that utilize `SimpleTestCase`, and then 20% of your tests be integration tests that utilize the database and `TestCase`.

CommCareHQ also has some custom in mocking tools.

- Fake Couch - Fake implementation of CouchDBKit api for testing purposes.
- ESQueryFake - For faking ES queries.

Squashing Migrations

There is overhead to running many migrations at once. Django allows you to squash migrations which will help speed up the migrations when running tests.

CHAPTER 9

Forms in HQ

Best practice principles:

- Use as little hardcoded HTML as possible.
- Submit and validate forms asynchronously to your class-based-view's *post* method.
- Protect forms against CSRF
- Be consistent with style across HQ. We are currently using Bootstrap 2.3's horizontal forms across HQ.
- Use *django.forms*.
- Use *crispy forms <http://django-crispy-forms.readthedocs.org/en/latest/>* for field layout.

Making forms CSRF safe

HQ is protected against cross site request forgery attacks i.e. if a *POST/PUT/DELETE* request doesn't pass csrf token to corresponding View, the View will reject those requests with a 403 response. All HTML forms and AJAX calls that make such requests should contain a csrf token to succeed. Making a form or AJAX code pass csrf token is easy and the Django docs give detailed instructions on how to do so. Here we list out examples of HQ code that does that

1. If crispy form is used to render HTML form, csrf token is included automagically
2. For raw HTML form, use *{% csrf_token %}* tag in the form HTML, see tag_csrf_example.
3. If request is made via AJAX, it will be automagically protected by *ajax_csrf_setup.js* (which is included in base bootstrap template) as long as your template is inherited from the base template. (*ajax_csrf_setup.js* overrides *$.ajaxSettings.beforeSend* to accomplish this)
4. If an AJAX call needs to override *beforeSend* itself, then the super *$.ajaxSettings.beforeSend* should be explicitly called to pass csrf token. See ajax_csrf_example
5. If request is made via Angluar JS controller, the angular app needs to be configured to send csrf token. See angular_csrf_example

6. If HTML form is created in Javascript using raw nodes, csrf-token node should be added to that form. See js_csrf_example_1 and js_csrf_example_2
7. If an inline form is generated using outside of *RequestContext* using *render_to_string* or its cousins, use *csrf_inline* custom tag. See inline_csrf_example
8. If a View needs to be exempted from csrf check (for whatever reason, say for API), use *csrf_exampt* decorator to avoid csrf check. See csrf_exempt_example
9. For any other special unusual case refer to Django docs. Essentially, either the HTTP request needs to have a csrf-token or the corresponding View should be exempted from CSRF check.

An Example Complex Asynchronous Form With Partial Fields

We create the following base form, subclassing *django.forms.Form*:

```
from django import forms
from crispy_forms.helper import FormHelper
from crispy_forms import layout as crispy

class PersonForm(forms.Form):
    first_name = forms.CharField()
    last_name = forms.CharField()
    pets = forms.CharField(widget=forms.HiddenInput)

    def __init__(self, *args, **kwargs):
        super(PersonForm, self).__init__(*args, **kwargs)

        self.helper = FormHelper()
        self.helper.layout = crispy.Layout(
            # all kwargs passed to crispy.Field turn into that tag's attributes and␣
↪underscores
            # become hyphens. so data_bind="value: name" gets inserted as data-bind=
↪"value: name"
            crispy.Field('first_name', data_bind="value: first_name"),
            crispy.Field('last_name', data_bind="value: last_name"),
            crispy.Div(
                data_bind="template: {name: 'pet-form-template', foreach: pets}, "
                          "visible: isPetVisible"
            ),
            # form actions creates the gray box around the submit / cancel buttons
            FormActions(
                StrictButton(
                    _("Update Information"),
                    css_class="btn-primary",
                    type="submit",
                ),
                # todo: add a cancel 'button' class!
                crispy.HTML('<a href="%s" class="btn">Cancel</a>' % cancel_url),
                # alternatively, the following works if you capture the name="cancel"
↪'s event in js:
                Button('cancel', 'Cancel'),
            ),
        )

    @property
    def current_values(self):
```

```
        values = dict([(name, self.person_form[name].value()) for name in self.person_
↪form.keys()])
        # here's where you would make sure events outputs the right thing
        # in this case, a list so it gets converted an ObservableArray for the␣
↪knockout model
        return values

    def clean_first_name(self):
        first_name = self.cleaned_data['first_name']
        # validate
        return first_name

    def clean_last_name(self):
        last_name = self.cleaned_data['last_name']
        # validate
        return last_name

    def clean_pets(self):
        # since we could have any number of pets we tell knockout to store it as json␣
↪in a hidden field
        pets = json.loads(self.cleaned_data['pets'])
        # validate pets
        # suggestion:
        errors = []
        for pet in pets:
            pet_form = PetForm(pet)
            pet_form.is_valid()
            errors.append(pet_form.errors)
        # raise errors as necessary
        return pets


class PetForm(forms.Form):
    nickname = CharField()

    def __init__(self, *args, **kwargs):
        super(PetForm, self).__init__(*args, **kwargs)

        self.helper = FormHelper()
        # since we're using this form to 'nest' inside of PersonForm, we want to␣
↪prevent
        # crispy forms from auto-including a form tag:
        self.helper.form_tag = False

        self.helper.layout = crispy.Layout(
            Field('nickname', data_bind="value: nickname"),
        )
```

The view will look something like:

```
class PersonFormView(BaseSectionPageView):
    # see documentation on ClassBasedViews for use of BaseSectionPageView
    template_name = 'people/person_form.html'
    allowed_post_actions = [
        'person_update',
        'select2_field_update',  # an example of another action you might consider
    ]
```

```
    @property
    @memoized
    def person_form(self):
        initial = {}
        if self.request.method == 'POST':
            return PersonForm(self.request.POST, initial={})
        return PersonForm(initial={})

    @property
    def page_context(self):
        return {
            'form': self.person_form,
            'pet_form': PetForm(),
        }

    @property
    def post_action:
        return self.request.POST.get('action')

    def post(self, *args, **kwargs):
        if self.post_action in self.allowed_post_actions:
            return HttpResponse(json.dumps(getattr(self, '%s_response' % self.
↪action)))
        # NOTE: doing the entire form asynchronously means that you have to␣
↪explicitly handle the display of
        # errors for each field. Ideally we should subclass crispy.Field to something␣
↪like KnockoutField
        # where we'd add something in the template for errors.
        raise Http404()

    @property
    def person_update_response(self):
        if self.person_form.is_valid():
            return {
                'data': self.person_form.current_values,
            }
        return {
            'errors': self.person_form.errors.as_json(),
            # note errors looks like:
            # {'field_name': [{'message': "msg", 'code': "invalid"}, {'message': "msg
↪", 'code': "required"}]}
        }
```

The template *people/person_form.html*:

```
{% extends 'people/base_template.html' %}
{% load hq_shared_tags %}
{% load i18n %}
{% load crispy_forms_tags %}

{% block js %}{{ block.super }}
    <script src="{% static 'people/ko/form.person.js' %}"></script>
{% endblock %}

{% block js-inline %}{{ block.super }}
    <script>
        var personFormModel = new PersonFormModel(
            {{ form.current_values|JSON }},
```

```
        );
        $('#person-form').koApplyBindings(personFormModel);
        personFormModel.init();
    </script>
{% endblock %}

{% block main_column %}
<div id="manage-reminders-form">
    <form class="form form-horizontal" method="post">
        {% crispy form %}
    </form>
</div>

<script type="text/html" id="pet-form-template">
    {% crispy pet_form %}
</script>
{% endblock %}
```

Your knockout code in *form.person.js*:

```
var PersonFormModel = function (initial) {
    'use strict';
    var self = this;

    self.first_name = ko.observable(initial.first_name);
    self.last_name = ko.observable(initial.last_name);

    self.petObjects = ko.observableArray();
    self.pets = ko.computed(function () {
        return JSON.stringify(_.map(self.petObjects(), function (pet) {
            return pet.asJSON();
        }));
    });

    self.init = function () {
        var pets = JSON.parse(initial.pets || '[]');
        self.petObjects(_.map(pets, function (initial_data) {
            return new Pet(initial_data);
        }));
    };

};

var Pet = function (initial) {
    'use strict';
    var self = this;

    self.nickname = ko.observable(initial.nickname);

    self.asJSON = ko.computed(function () {
        return {
            nickname: self.nickname()
        }
    });
};
```

That should hopefully get you 90% there. For an example on HQ see *corehq.apps.reminders.views.CreateScheduledReminderView* <*https://github.com/dimagi/commcare-*

hq/blob/master/corehq/apps/reminders/views.py#L486>

CHAPTER 10

HQ Management Commands

This is a list of useful management commands. They can be run using `$ python manage.py <command>` or `$ ./manage.py <command>`. For more information on a specific command, run `$ ./manage.py <command> --help`

bootstrap Bootstrap a domain and user who owns it. Usage:: $ ./manage.py bootstrap [options] <domain> <email> <password>

bootstrap_app Bootstrap an app in an existing domain. Usage:: $ ./manage.py bootstrap_app [options] <domain_name> <app_name>

clean_pyc Removes all python bytecode (.pyc) compiled files from the project.

copy_domain Copies the contents of a domain to another database. Usage:: $ ./manage.py copy_domain [options] <sourcedb> <domain>

ptop_reindexer_fluff Fast reindex of fluff docs. Usage:: $ ./manage.py ptop_reindexer_fluff <pillow_name>

run_ptop Run the pillowtop management command to scan all _changes feeds

runserver

Starts a lightweight web server for development which outputs additional debug information.

`--werkzeug` Tells Django to use the Werkzeug interactive debugger.

syncdb

Create the database tables for all apps in INSTALLED_APPS whose tables haven't already been created, except those which use migrations.

`--migrate` Tells South to also perform migrations after the sync.

test Runs the test suite for the specified applications, or the entire site if no apps are specified. Usage:: $ ./manage.py test [options] [appname ...]

CHAPTER 11

CommTrack

What happens during a CommTrack submission?

This is the life-cycle of an incoming stock report via sms.

1. SMS is received and relevant info therein is parsed out
2. The parsed sms is converted to an HQ-compatible xform submission. This includes:
   - stock/requisition info (i.e., just the data provided in the sms)
   - location to which this message applies (provided in message or associated with sending user)
   - standard HQ submission meta-data (submit time, user, etc.)

   Notably missing: anything that updates cases
3. The submission is *not* submitted yet, but rather processed further on the server. This includes:
   - looking up the product sub-cases that actually store stock/consumption values. (step (2) looked up the location ID; each supply point is a case associated with that location, and actual stock data is stored in a sub-case – one for each product – of the supply point case)
   - applying the stock actions for each product in the correct order (a stock report can include multiple actions; these must be applied in a consistent order or else unpredictable stock levels may result)
   - computing updated stock levels and consumption (using somewhat complex business and reconciliation logic)
   - dumping the result in case blocks (added to the submission) that will update the new values in HQ's database
   - **post-processing also makes some changes elsewhere in the instance, namely:**
     - also added are 'inferred' transactions (if my stock was 20, is now 10, and i had receipts of 15, my inferred consumption was 25). This is needed to compute consumption rate later. Conversely, if a deployment tracks consumption instead of receipts, receipts are inferred this way.

    - transactions are annotated with the order in which they were processed

  Note that normally CommCare generates its own case blocks in the forms it submits.

4. The updated submission is submitted to HQ like a normal form

Submitting a stock report via CommCare

CommTrack-enabled CommCare submits xforms, but those xforms **do not** go through the post-processing step in (3) above. Therefore these forms must generate their own case blocks and mimic the end result that commtrack expects. This is severely lacking as we have not replicated the full logic from the server in these xforms (unsure if that's even possible, nor do we like the prospect of maintaining the same logic in two places), nor can these forms generate the inferred transactions. As such, the capabilities of the mobile app are greatly restricted and cannot support features like computing consumption.

This must be fixed and it's really not worth even discussing much else about using a mobile app until it is.

CHAPTER 12

CloudCare

Overview

The goal of this section is to give an overview of the CloudCare system for developers who are new to CloudCare. It should allow one's first foray into the system to be as painless as possible by giving him or her a high level overview of the system.

Backbone

On the frontend, CloudCare is a single page backbone.js app. The app, module, form, and case selection parts of the interface are rendered by backbone while the representation of the form itself is controlled by touchforms (described below).

When a user navigates CloudCare, the browser is not making full page reload requests to our Django server, instead, javascript is used to modify the contents of the page and change the url in the address bar. Whenever a user directly enters a CloudCare url like `/a/<domain>/cloudcare/apps/<urlPath>` into the browser, the cloudcare_main view is called. This page loads the backbone app and perhaps bootstraps it with the currently selected app and case.

The Backbone Views

The backbone app consists of several `Backbone.Views` subclasses. What follows is a brief description of several of the most important classes used in the CloudCare backbone app.

`cloudCare.AppListView` Renders the list of apps in the current domain on the left hand side of the page.

`cloudCare.ModuleListView` Renders the list of modules in the currently selected app on the left hand side of the page.

`cloudCare.FormListView` Renders the list of forms in the currently selected module on the left hand side of the page.

`cloudCareCases.CaseMainView` Renders the list of cases for the selected form. Note that this list is populated asynchronously.

cloudCareCases.CaseDetailsView Renders the table displaying the currently selected case's properties.

cloudCare.AppView AppView holds the module and form list views. It is also responsible for inserting the form html into the DOM. This html is constructed using JSON returned by the touchforms process and several js libs found in the `/touchforms/formplayer/static/formplayer/script/` directory. This is kicked off by the AppView's `_playForm` method. AppView also inserts `cloudCareCases.CaseMainViews` as necessary.

cloudCare.AppMainView AppMainView (not to be confused with AppView) holds all of the other views and is the entry point for the application. Most of the applications event handling is set up inside AppMainView's `initialize` method. The AppMainView has a router. Event handlers are set on this router to modify the state of the backbone application when the browser's back button is used, or when the user enters a link to a certain part of the app (like a particular form) directly.

Touchforms

The backbone app is not responsible for processing the XFrom. This is done instead by our XForms player, touchforms. Touchforms runs as a separate process on our servers, and sends JSON to the backbone application representing the structure of the XForm. Touchforms is written in jython, and serves as a wrapper around the JavaRosa that powers our mobile applications.

Offline Cloudcare

What is it?

First of all, the "offline" part is a misnomer. This does not let you use CloudCare completely offline. We need a new name.

Normal CloudCare requires a round-trip request to the HQ touchforms daemon every time you answer/change a question in a form. This is how it can handle validation logic and conditional questions with the exact same behavior as on the phone. On high-latency or unreliable internet this is a major drag.

"Offline" CloudCare fixes this by running a local instance of the touchforms daemon. CloudCare (in the browser) communicates with this daemon for all matters of maintaining the xform session state. However, CloudCare still talks directly to HQ for other CloudCare operations, such as initial launch of a form, submitting the completed form, and everything outside a form session (case list/select, etc.). Also, the local daemon itself will call out to HQ as needed by the form, such as querying against the casedb. *So you still need internet!*

How does it work?

The touchforms daemon (i.e., the standard JavaRosa/CommCare core with a Jython wrapper) is packaged up as a standalone jar that can be run from pure Java. This requires bundling the Jython runtime. This jar is then served as a "Java Web Start" (aka JNLP) application (same as how you download and run WebEx).

When CloudCare is in offline mode, it will prompt you to download the app; once you do the app will auto-launch. CloudCare will poll the local port the app should be running on, and once its ready, will then initialize the form session and direct all touchforms queries to the local instance rather than HQ.

The app download should persist in a local application cache, so it will not have to be downloaded each time. The initial download is somewhat beefy (14MB) primarily due to the inclusion of the Jython runtime. It is possible we may be able to trim this down by removing unused stuff. When started, the app will automatically check for updates

(though there may be a delay before the updates take effect). When updating, only the components that changed need to be re-downloaded (so unless we upgrade Jython, the big part of the download is a one-time cost).

When running, the daemon creates an icon in the systray. This is also where you terminate it.

How do I get it?

Offline mode for CloudCare is currently hidden until we better decide how to intergrate it, and give it some minimal testing. To access:

- Go to the main CloudCare page, but don't open any forms
- Open the chrome dev console (`F12` or `ctrl+shift+J`)
- Type `enableOffline()` in the console
- Note the new 'Use Offline CloudCare' checkbox on the left

CHAPTER 13

Internationalization

This page contains the most common techniques needed for managing CommCare HQ localization strings. For more comprehensive information, consult the Django Docs translations page or this helpful blog post.

Tagging strings in views

TL;DR: `ugettext` should be used in code that will be run per-request. `ugettext_lazy` should be used in code that is run at module import.

The management command `makemessages` pulls out strings marked for translation so they can be translated via transifex. All three ugettext functions mark strings for translation. The actual translation is performed separately. This is where the ugettext functions differ.

- `ugettext`: The function immediately returns the translation for the currently selected language.
- `ugettext_lazy`: The function converts the string to a translation "promise" object. This is later coerced to a string when rendering a template or otherwise forcing the promise.
- `ugettext_noop`: This function only marks a string as translation string, it does not have any other effect; that is, it always returns the string itself. This should be considered an advanced tool and generally avoided. It could be useful if you need access to both the translated and untranslated strings.

The most common case is just wrapping text with ugettext.

```
from django.utils.translation import ugettext as _

def my_view(request):
    messages.success(request, _("Welcome!"))
```

Typically when code is run as a result of a module being imported, there is not yet a user whose locale can be used for translations, so it must be delayed. This is where *ugettext_lazy* comes in. It will mark a string for translation, but delay the actual translation as long as possible.

```
class MyAccountSettingsView(BaseMyAccountView):
    urlname = 'my_account_settings'
    page_title = ugettext_lazy("My Information")
    template_name = 'settings/edit_my_account.html'
```

When variables are needed in the middle of translated strings, interpolation can be used as normal. However, named variables should be used to ensure that the translator has enough context.

```
message = _("User '{user}' has successfully been {action}.").format(
    user=user.raw_username,
    action=_("Un-Archived") if user.is_active else _("Archived"),
)
```

This ends up in the translations file as:

```
msgid "User '{user}' has successfully been {action}."
```

Using `ugettext_lazy`

The *ugettext_lazy* method will work in the majority of translation situations. It flags the string for translation but does not translate it until it is rendered for display. If the string needs to be immediately used or manipulated by other methods, this might not work.

When using the value immediately, there is no reason to do lazy translation.

```
return HttpResponse(ugettext("An error was encountered."))
```

It is easy to forget to translate form field names, as Django normally builds nice looking text for you. When writing forms, make sure to specify labels with a translation flagged value. These will need to be done with *ugettext_lazy*.

```
class BaseUserInfoForm(forms.Form):
    first_name = forms.CharField(label=ugettext_lazy('First Name'), max_length=50,␣
↪required=False)
    last_name = forms.CharField(label=ugettext_lazy('Last Name'), max_length=50,␣
↪required=False)
```

`ugettext_lazy`, a cautionary tale

`ugettext_lazy` does not return a string. This can cause complications.

When using methods to manipulate a string, lazy translated strings will not work properly.

```
group_name = ugettext("mobile workers")
return group_name.upper()
```

Converting `ugettext_lazy` objects to json will crash. You should use `dimagi.utils.web.json_handler` to properly coerce it to a string.

```
>>> import json
>>> from django.utils.translation import ugettext_lazy
>>> json.dumps({"message": ugettext_lazy("Hello!")})
TypeError: <django.utils.functional.__proxy__ object at 0x7fb50766f3d0> is not JSON␣
↪serializable
>>> from dimagi.utils.web import json_handler
```

```
>>> json.dumps({"message": ugettext_lazy("Hello!")}, default=json_handler)
'{"message": "Hello!"}'
```

Tagging strings in template files

There are two ways translations get tagged in templates.

For simple and short plain text strings, use the *trans* template tag.

```
{% trans "Welcome to CommCare HQ" %}
```

More complex strings (requiring interpolation, variable usage or those that span multiple lines) can make use of the *blocktrans* tag.

If you need to access a variable from the page context:

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

If you need to make use of an expression in the translation:

```
{% blocktrans with amount=article.price %}
    That will cost $ {{ amount }}.
{% endblocktrans %}
```

This same syntax can also be used with template filters:

```
{% blocktrans with myvar=value|filter %}
    This will have {{ myvar }} inside.
{% endblocktrans %}
```

In general, you want to avoid including HTML in translations. This will make it easier for the translator to understand and manipulate the text. However, you can't always break up the string in a way that gives the translator enough context to accurately do the translation. In that case, HTML inside the translation tags will still be accepted.

```
{% blocktrans %}
    Manage Mobile Workers <small>for CommCare Mobile and
    CommCare HQ Reports</small>
{% endblocktrans %}
```

Text passed as constant strings to template block tag also needs to be translated. This is most often the case in CommCare with forms.

```
{% crispy form _("Specify New Password") %}
```

Keeping translations up to date

Once a string has been added to the code, we can update the .po file by running *makemessages*.

To do this for all langauges:

```
$ django-admin.py makemessages --all
```

It will be quicker for testing during development to only build one language:

```
$ django-admin.py makemessages -l fra
```

After this command has run, your .po files will be up to date. To have content in this file show up on the website you still need to compile the strings.

```
$ django-admin.py compilemessages
```

You may notice at this point that not all tagged strings with an associated translation in the .po shows up translated. That could be because Django made a guess on the translated value and marked the string as fuzzy. Any string marked fuzzy will not be displayed and is an indication to the translator to double check this.

Example:

```
#: corehq/__init__.py:103
#, fuzzy
msgid "Export Data"
msgstr "Exporter des cas"
```

CHAPTER 14

Profiling

Practical guide to profiling a slow view or function

This will walkthrough one way to profile slow code using the @profile decorator.

At a high level this is the process:

1. Find the function that is slow
2. Add a decorator to save a raw profile file that will collect information about function calls and timing
3. Use libraries to analyze the raw profile file and spit out more useful information
4. Inspect the output of that information and look for anomalies
5. Make a change, observe the updated load times and repeat the process as necessary

Finding the slow function

This is usually pretty straightforward. The easiest thing to do is typically use the top-level entry point for a view call. In this example we are investigating the performance of commtrack location download, so the relevant function would be commtrack.views.location_export.:

```
@login_and_domain_required
def location_export(request, domain):
    response = HttpResponse(mimetype=Format.from_format('xlsx').mimetype)
    response['Content-Disposition'] = 'attachment; filename="locations.xlsx"'
    dump_locations(response, domain)
    return response
```

Getting a profile dump

To get a profile dump, simply add the following decoration to the function.:

```
from dimagi.utils.decorators.profile import profile
@login_and_domain_required
@profile('locations_download.prof')
def location_export(request, domain):
    response = HttpResponse(mimetype=Format.from_format('xlsx').mimetype)
    response['Content-Disposition'] = 'attachment; filename="locations.xlsx"'
    dump_locations(response, domain)
    return response
```

Now each time you load the page a raw dump file will be created with a timestamp of when it was run. These are created in /tmp/ by default, however you can change it by adding a value to your settings.py like so:

```
PROFILE_LOG_BASE = "/home/czue/profiling/"
```

Note that the files created are huge; this code should only be run locally.

Creating a more useful output from the dump file

The raw profile files are not human readable, and you need to use something like hotshot to make them useful. A script that will generate what is typically sufficient information to analyze these can be found in the commcarehq-scripts repository. You can read the source of that script to generate your own analysis, or just use it directly as follows:

```
$ ./reusable/convert_profile.py /path/to/profile_dump.prof
```

Reading the output of the analysis file

The analysis file is broken into two sections. The first section is an ordered breakdown of calls by the **cumulative** time spent in those functions. It also shows the number of calls and average time per call.

The second section is harder to read, and shows the callers to each function.

This analysis will focus on the first section. The second section is useful when you determine a huge amount of time is being spent in a function but it's not clear where that function is getting called.

Here is a sample start to that file:

```
loading profile stats for locations_download/commtrack-location-20140822T205905.prof
         361742 function calls (355960 primitive calls) in 8.838 seconds

   Ordered by: cumulative time, call count
   List reduced from 840 to 200 due to restriction <200>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    8.838    8.838 /home/czue/src/commcare-hq/corehq/apps/
↪locations/views.py:336(location_export)
        1    0.011    0.011    8.838    8.838 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:248(dump_locations)
      194    0.001    0.000    8.128    0.042 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:136(parent)
      190    0.002    0.000    8.121    0.043 /home/czue/src/commcare-hq/corehq/apps/
↪cachehq/mixins.py:35(get)
      190    0.003    0.000    8.021    0.042 submodules/dimagi-utils-src/dimagi/
↪utils/couch/cache/cache_core/api.py:65(cached_open_doc)
      190    0.013    0.000    7.882    0.041 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/client.py:362(open_doc)
```

```
      396    0.003    0.000    7.762    0.020 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/_socketio.py:56(readinto)
      396    7.757    0.020    7.757    0.020 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/_socketio.py:24(<lambda>)
      196    0.001    0.000    7.414    0.038 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/resource.py:40(json_body)
      196    0.011    0.000    7.402    0.038 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/restkit/wrappers.py:270(body_string)
      590    0.019    0.000    7.356    0.012 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/reader.py:19(readinto)
      198    0.002    0.000    0.618    0.003 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/resource.py:69(request)
      196    0.001    0.000    0.616    0.003 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/restkit/resource.py:105(get)
      198    0.004    0.000    0.615    0.003 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/restkit/resource.py:164(request)
      198    0.002    0.000    0.605    0.003 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/restkit/client.py:415(request)
      198    0.003    0.000    0.596    0.003 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/restkit/client.py:293(perform)
      198    0.005    0.000    0.537    0.003 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/restkit/client.py:456(get_response)
      396    0.001    0.000    0.492    0.001 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/http.py:135(headers)
      790    0.002    0.000    0.452    0.001 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/http.py:50(_check_headers_complete)
      198    0.015    0.000    0.450    0.002 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/http.py:191(__next__)
1159/1117    0.043    0.000    0.396    0.000 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/jsonobject/base.py:559(__init__)
    13691    0.041    0.000    0.227    0.000 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/jsonobject/base.py:660(__setitem__)
      103    0.005    0.000    0.219    0.002 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:65(location_custom_properties)
      103    0.000    0.000    0.201    0.002 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:70(<genexpr>)
  333/303    0.001    0.000    0.190    0.001 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/jsonobject/base.py:615(wrap)
      289    0.002    0.000    0.185    0.001 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:31(__init__)
        6    0.000    0.000    0.176    0.029 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/client.py:1024(_fetch_if_needed)
```

The most important thing to look at is the cumtime (cumulative time) column. In this example we can see that the vast majority of the time (over 8 of the 8.9 total seconds) is spent in the cached_open_doc function (and likely the library calls below are called by that function). This would be the first place to start when looking at improving profile performance. The first few questions that would be useful to ask include:

- Can we optimize the function?
- Can we reduce calls to that function?
- In the case where that function is hitting a database or a disk, can the code be rewritten to load things in bulk?

In this practical example, the function is clearly meant to already be caching (based on the name alone) so it's possible that the results would be different if caching was enabled and the cache was hot. It would be good to make sure we test with those two parameters true as well. This can be done by changing your localsettings file and setting the following two variables:

```
COUCH_CACHE_DOCS = True
COUCH_CACHE_VIEWS = True
```

Reloading the page twice (the first time to prime the cache and the second time to profile with a hot cache) will then produce a vastly different output:

```
loading profile stats for locations_download/commtrack-location-20140822T211654.prof
         303361 function calls (297602 primitive calls) in 0.484 seconds

   Ordered by: cumulative time, call count
   List reduced from 741 to 200 due to restriction <200>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.484    0.484 /home/czue/src/commcare-hq/corehq/apps/
↪locations/views.py:336(location_export)
        1    0.004    0.004    0.484    0.484 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:248(dump_locations)
1159/1117    0.017    0.000    0.160    0.000 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/jsonobject/base.py:559(__init__)
        4    0.000    0.000    0.128    0.032 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:62(filter_by_type)
        4    0.000    0.000    0.128    0.032 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/client.py:986(all)
      103    0.000    0.000    0.128    0.001 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/client.py:946(iterator)
        4    0.000    0.000    0.128    0.032 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/client.py:1024(_fetch_if_needed)
        4    0.000    0.000    0.128    0.032 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/client.py:995(fetch)
        9    0.000    0.000    0.124    0.014 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/_socketio.py:56(readinto)
        9    0.124    0.014    0.124    0.014 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/_socketio.py:24(<lambda>)
        4    0.000    0.000    0.114    0.029 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/couchdbkit/resource.py:40(json_body)
        4    0.000    0.000    0.114    0.029 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/restkit/wrappers.py:270(body_string)
       13    0.000    0.000    0.114    0.009 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/http_parser/reader.py:19(readinto)
      103    0.000    0.000    0.112    0.001 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:70(<genexpr>)
    13691    0.018    0.000    0.094    0.000 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/jsonobject/base.py:660(__setitem__)
      103    0.002    0.000    0.091    0.001 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:65(location_custom_properties)
      194    0.000    0.000    0.078    0.000 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:136(parent)
      190    0.000    0.000    0.076    0.000 /home/czue/src/commcare-hq/corehq/apps/
↪cachehq/mixins.py:35(get)
      103    0.000    0.000    0.075    0.001 submodules/dimagi-utils-src/dimagi/
↪utils/couch/database.py:50(iter_docs)
        4    0.000    0.000    0.075    0.019 submodules/dimagi-utils-src/dimagi/
↪utils/couch/bulk.py:81(get_docs)
        4    0.000    0.000    0.073    0.018 /home/czue/.virtualenvs/commcare-hq/
↪local/lib/python2.7/site-packages/requests/api.py:80(post)
```

Yikes! It looks like this is already quite fast with a hot cache! And there don't appear to be any obvious candidates for further optimization. If it is still a problem it may be an indication that we need to prime the cache better, or increase

the amount of data we are testing with locally to see more interesting results.

Aggregating data from multiple runs

In some cases it is useful to run a function a number of times and aggregate the profile data. To do this follow the steps above to create a set of '.prof' files (one for each run of the function) then use the 'gather_profile_stats.py' script included with django (lib/python2.7/site-packages/django/bin/profiling/gather_profile_stats.py) to aggregate the data.

This will produce a '.agg.prof' file which can be analysed with the prof.py script.

Line profiling

In addition to the above methods of profiling it is possible to do line profiling of code which attached profile data to individual lines of code as opposed to function names.

The easiest way to do this is to use the line_profile decorator.

Example output:

```
File: demo.py
Function: demo_follow at line 67
Total time: 1.00391 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    67                                           def demo_follow():
    68         1           34     34.0      0.0      r = random.randint(5, 10)
    69        11           81      7.4      0.0      for i in xrange(0, r):
    70        10      1003800 100380.0    100.0          time.sleep(0.1)
File: demo.py
Function: demo_profiler at line 72
Total time: 1.80702 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    72                                           @line_profile(follow=[demo_follow])
    73                                           def demo_profiler():
    74         1           17     17.0      0.0      r = random.randint(5, 10)
    75         9           66      7.3      0.0      for i in xrange(0, r):
    76         8       802921 100365.1     44.4          time.sleep(0.1)
    77
    78         1      1004013 1004013.0     55.6      demo_follow()
```

More details here:

- https://github.com/dmclain/django-debug-toolbar-line-profiler
- https://github.com/dcramer/django-devserver#devservermodulesprofilelineprofilermodule

Additional references

- http://django-extensions.readthedocs.org/en/latest/runprofileserver.html

Memory profiling

Refer to these resources which provide good information on memory profiling:

- Diagnosing memory leaks
- Using heapy
- Diving into python memory
- **Memory usage graphs with ps**
  - *while true; do ps -C python -o etimes=,pid=,%mem=,vsz= >> mem.txt; sleep 1; done*
- You can also use the "resident_set_size" decorator and context manager to print the amount of memory allocated to python before and after the method you think is causing memory leaks:

```
from dimagi.utils.decorators.profile import resident_set_size

@resident_set_size()
def function_that_uses_a_lot_of_memory:
    [u'{}'.format(x) for x in range(1,100000)]

def somewhere_else():
    with resident_set_size(enter_debugger=True):
        # the enter_debugger param will enter a pdb session after your method has␣
↪run so you can do more exploration
        # do memory intensive things
```

CHAPTER 15

ElasticSearch

Indexes

We have indexes for each of the following doc types:

- Applications - `hqapps`
- Cases - `hqcases`
- Domains - `hqdomains`
- Forms - `xforms`
- Groups - `hqgroups`
- Users - `hqusers`
- Report Cases - `report_cases`
- Report Forms - `report_xforms`
- SMS logs - `smslogs`
- TrialConnect SMS logs - `tc_smslogs`

The *Report* cases and forms indexes are only configured to run for a few domains, and they store additional mappings allowing you to query on form and case properties (not just metadata).

Each index has a corresponding mapping file in `corehq/pillows/mappings/`. Each mapping has a hash that reflects the current state of the mapping. This is appended to the index name so the index is called something like `xforms_1cce1f049a1b4d864c9c25dc42648a45`. Each type of index has an alias with the short name, so you should normally be querying just `xforms`, not the fully specified index+hash.

Whenever the mapping is changed, this hash should be updated. That will trigger the creation of a new index on deploy (by the `$ ./manage.py ptop_preindex` command). Once the new index is finished, the alias is *flipped* (`$ ./manage.py ptop_es_manage --flip_all_aliases`) to point to the new index, allowing for a relatively seamless transition.

Keeping indexes up-to-date

Pillowtop looks at the changes feed from couch and listens for any relevant new/changed docs. In order to have your changes appear in elasticsearch, pillowtop must be running:

```
$ ./manage.py run_ptop --all
```

You can also run a once-off reindex for a specific index:

```
$ ./manage.py ptop_reindexer_v2 user
```

Changing a mapping or adding data

If you're adding additional data to elasticsearch, you'll need modify that index's mapping file in order to be able to query on that new data.

Adding data to an index

Each pillow has a function or class that takes in the raw document dictionary and transforms it into the document that get's sent to ES. If for example, you wanted to store username in addition to user_id on cases in elastic, you'd add `username` to `corehq.pillows.mappings.case_mapping`, then modify `transform_case_for_elasticsearch` function to do the appropriate lookup. It accepts a `doc_dict` for the case doc and is expected to return a `doc_dict`, so just add the `username` to that.

Building the new index

Once you've made the change, you'll need to build a new index which uses that new mapping, so you'll have to update the hash at the top of the file. This can just be a random alphanumeric string. This will trigger a preindex as outlined in the *Indexes* section.

How to un-bork your broken indexes

Sometimes things get in a weird state and (locally!) it's easiest to just blow away the index and start over.

1. Delete the affected index. The easiest way to do this is with elasticsearch-head. You can delete multiple affected indices with `curl -X DELETE http://localhost:9200/*`. `*` can be replaced with any regex to delete matched indices, similar to bash regex.
2. Run `$ ./manage.py ptop_preindex && ./manage.py ptop_es_manage --flip_all_aliases`.
3. Try again

Querying Elasticsearch - Best Practices

Here are the most basic things to know if you want to write readable and reasonably performant code for accessing Elasticsearch.

CHAPTER 16

Use ESQuery when possible

Check out *ESQuery*

- Prefer the cleaner `.count()`, `.values()`, `.values_list()`, etc. execution methods to the more low level `.run().hits`, `.run().total`, etc. With the latter easier to make mistakes and fall into anti-patterns and it's harder to read.
- Prefer adding filter methods to using `set_query()` unless you really know what you're doing and are willing to make your code more error prone and difficult to read.

CHAPTER 17

Prefer “get” to “search”

Don’t use search to fetch a doc or doc fields by doc id; use “get” instead. Searching by id can be easily an order of magnitude (10x) slower. If done in a loop, this can effectively grind the ES cluster to a halt.

Bad::

```
POST /hqcases_2016-03-04/case/_search
{
  "query": {
    "filtered": {
      "filter": {
        "and": [{"terms": {"_id": [case_id]}}, {"match_all": {}}]
      },
      "query": {"match_all":{}}
    }
  },
  "_source": ["name"],
  "size":1000000
}
```

Good::

```
GET /hqcases_2016-03-04/case/<case_id>?_source_include=name
```

CHAPTER 18

Prefer scroll queries

Use a scroll query when fetching lots of records.

CHAPTER 19

Prefer filter to query

Don’t use `query` when you could use `filter` if you don’t need rank.

CHAPTER 20

Use size(0) with aggregations

Use `size(0)` when you're only doing aggregations thing—otherwise you'll get back doc bodies as well! Sometimes that's just abstractly wasteful, but often it can be a serious performance hit for the operation as well as the cluster.

The best way to do this is by using helpers like ESQuery's `.count()` that know to do this for you—your code will look better and you won't have to remember to check for that every time. (If you ever find *helpers* not doing this correctly, then it's definitely worth fixing.)

CHAPTER 21

ESQuery

CHAPTER 22

Analyzing Test Coverage

This page goes over some basic ways to analyze code coverage locally.

Using coverage.py

First thing is to install the coverage.py library:

```
$ pip install coverage
```

Now you can run your tests through the coverage.py program:

```
$ coverage run manage.py test commtrack
```

This will create a binary *commcare-hq/.coverage* file (that is already ignored by our *.gitignore*) which contains all the magic bits about what happened during the test run.

You can be as specific or generic as you'd like with what selection of tests you run through this. This tool will track which lines of code in the app have been hit during execution of the tests you run. If you're only looking to analyze (and hopefully increase) coverage in a specific model or utils file, it might be helpful to cut down on how many tests you're running.

Make an HTML view of the data

The simplest (and probably fastest) way to view this data is to build an HTML view of the code base with the coverage data:

```
$ coverage html
```

This will build a *commcare-hq/coverage-report/* directory with a ton of HTML files in it. The important one is *commcare-hq/coverage-report/index.html*.

View the result in Vim

Install coveragepy.vim (https://github.com/alfredodeza/coveragepy.vim) however you personally like to install plugins. This plugin is old and out of date (but seems to be the only reasonable option) so because of this I personally think the HTML version is better.

Then run *:Coveragepy report* in Vim to build the report (this is kind of slow).

You can then use *:Coveragepy hide* and *:Coveragepy show* to add/remove the view from your current buffer.

CHAPTER 23

Advanced App Features

See `corehq.apps.app_manager.suite_xml.SuiteGenerator` and `corehq.apps.app_manager.xform.XForm` for code.

Child Modules

In principle child modules is very simple. Making one module a child of another simply changes the `menu` elements in the *suite.xml* file. For example in the XML below module `m1` is a child of module `m0` and so it has its `root` attribute set to the ID of its parent.

```
<menu id="m0">
    <text>
        <locale id="modules.m0"/>
    </text>
    <command id="m0-f0"/>
</menu>
<menu id="m1" root="m0">
    <text>
        <locale id="modules.m1"/>
    </text>
    <command id="m1-f0"/>
</menu>
```

Menu structure

As described above the basic menu structure is quite simple however there is one property in particular that affects the menu structure: *module.put_in_root*

This property determines whether the forms in a module should be shown under the module's own menu item or under the parent menu item:

put_in_root	Resulting menu
True	id="<parent menu id>"
False	id="<module menu id>" root="<parent menu id>"

Notes:

- If the module has no parent then the parent is *root*.
- *root="root"* is equivalent to excluding the *root* attribute altogether.

Session Variables

This is all good and well until we take into account the way the Session works on the mobile which "prioritizes the most relevant piece of information to be determined by the user at any given time".

This means that if all the forms in a module require the same case (actually just the same session IDs) then the user will be asked to select the case before selecting the form. This is why when you build a module where *all forms require a case* the case selection happens before the form selection.

From here on we will assume that all forms in a module have the same case management and hence require the same session variables.

When we add a child module into the mix we need to make sure that the session variables for the child module forms match those of the parent in two ways, matching session variable names and adding in any missing variables.

Matching session variable names

For example, consider the session variables for these two modules:

module A:

```
case_id:            load mother case
```

module B child of module A:

```
case_id_mother:     load mother case
case_id_child:      load child case
```

You can see that they are both loading a mother case but are using different session variable names.

To fix this we need to adjust the variable name in the child module forms otherwise the user will be asked to select the mother case again:

case_id_mother -> case_id

module B final:

```
case_id:            load mother case
case_id_child:      load child case
```

Inserting missing variables

In this case imagine our two modules look like this:

module A:

```
case_id:            load patient case
case_id_new_visit:  id for new visit case ( uuid() )
```

module B child of module A:

```
case_id:            load patient case
case_id_child:      load child case
```

Here we can see that both modules load the patient case and that the session IDs match so we don't have to change anything there.

The problem here is that forms in the parent module also add a `case_id_new_visit` variable to the session which the child module forms do not. So we need to add it in:

module B final:

```
case_id:            load patient case
case_id_new_visit:  id for new visit case ( uuid() )
case_id_child:      load child case
```

Note that we can only do this for session variables that are automatically computed and hence does not require user input.

Shadow Modules

A shadow module is a module that piggybacks on another module's commands (the "source" module). The shadow module has its own name, case list configuration, and case detail configuration, but it uses the same forms as its source module.

This is primarily for clinical workflows, where the case detail is a list of patients and the clinic wishes to be able to view differently-filtered queues of patients that ultimately use the same set of forms.

Shadow modules are behind the feature flag **Shadow Modules**.

Scope

The shadow module has its own independent:

- Name
- Menu mode (display module & forms, or forms only)
- Media (icon, audio)
- Case list configuration (including sorting and filtering)
- Case detail configuration

The shadow module inherits from its source:

- case type
- commands (which forms the module leads to)
- end of form behavior

Limitations

A shadow module can neither **be** a parent module nor **have** a parent module

A shadow module's source can be a parent module (the shadow will include a copy of the children), or have a parent module (the shadow will appear as a child of that same parent)

Shadow modules are designed to be used with case modules. They may behave unpredictably if given an advanced module, reporting module, or careplan module as a source.

Shadow modules do not necessarily behave well when the source module uses custom case tiles. If you experience problems, make the shadow module's case tile configuration exactly matches the source module's.

Entries

A shadow module duplicates all of its parent's entries. In the example below, m1 is a shadow of m0, which has one form. This results in two unique entries, one for each module, which share several properties.

```
<entry>
    <form>
        http://openrosa.org/formdesigner/86A707AF-3A76-4B36-95AD-FF1EBFDD58D8
    </form>
    <command id="m0-f0">
        <text>
            <locale id="forms.m0f0"/>
        </text>
    </command>
</entry>
<entry>
    <form>
        http://openrosa.org/formdesigner/86A707AF-3A76-4B36-95AD-FF1EBFDD58D8
    </form>
    <command id="m1-f0">
        <text>
            <locale id="forms.m0f0"/>
        </text>
    </command>
</entry>
```

Menu structure

In the simplest case, shadow module menus look exactly like other module menus. In the example below, m1 is a shadow of m0. The two modules have their own, unique menu elements.

```
<menu id="m0">
    <text>
        <locale id="modules.m0"/>
    </text>
    <command id="m0-f0"/>
</menu>
<menu id="m1">
    <text>
        <locale id="modules.m1"/>
        </text>
    <command id="m1-f0"/>
</menu>
```

Menus get more complex when shadow modules are mixed with parent/child modules. In the following example, m0 is a basic module, m1 is a child of m0, and m2 is a shadow of m0. All three modules have *put_in_root=false* (see **Child Modules > Menu structure** above). The shadow module has its own menu and also a copy of the child

module's menu. This copy of the child module's menu is given the id *m1.m2* to distinguish it from *m1*, the original child module menu.

```
<menu id="m0">
    <text>
        <locale id="modules.m0"/>
    </text>
    <command id="m0-f0"/>
</menu>
<menu root="m0" id="m1">
    <text>
        <locale id="modules.m1"/>
    </text>
    <command id="m1-f0"/>
</menu>
<menu root="m2" id="m1.m2">
↪                                                    <text>
        <locale id="modules.m1"/>
    </text>
↪                                                    <command id="m1-f0"/>
</menu>
<menu id="m2">
↪                                                    <text>
        <locale id="modules.m2"/>
    </text>
↪                                                    <command id="m2-f0"/>
</menu>
```

CHAPTER 24

Using the shared NFS drive

On our production servers (and staging) we have an NFS drive set up that we can use for a number of things:

- store files that are generated asynchronously for retrieval in a later request * previously we needed to save these files to Redis so that they would be available to all the Django workers on the next request * doing this has the added benefit of allowing apache / nginx to handle the file transfer instead of Django
- store files uploaded by the user that require asynchronous processing

Using apache / nginx to handle downloads

```
import os
import tempfile
from wsgiref.util import FileWrapper
from django.conf import settings
from django.http import StreamingHttpResponse
from django_transfer import TransferHttpResponse

transfer_enabled = settings.SHARED_DRIVE_CONF.transfer_enabled
if transfer_enabled:
    path = os.path.join(settings.SHARED_DRIVE_CONF.transfer_dir, uuid.uuid4().hex)
else:
    _, path = tempfile.mkstemp()

make_file(path)

if transfer_enabled:
    response = TransferHttpResponse(path, content_type=self.zip_mimetype)
else:
    response = StreamingHttpResponse(FileWrapper(open(path)), content_type=self.zip_
↪mimetype)

response['Content-Length'] = os.path.getsize(fpath)
```

```
response["Content-Disposition"] = 'attachment; filename="%s"' % filename
return response
```

This also works for files that are generated asynchronously:

```
@task
def generate_download(download_id):
    use_transfer = settings.SHARED_DRIVE_CONF.transfer_enabled
    if use_transfer:
        path = os.path.join(settings.SHARED_DRIVE_CONF.transfer_dir, uuid.uuid4().hex)
    else:
        _, path = tempfile.mkstemp()

    generate_file(path)

    common_kwargs = dict(
        mimetype='application/zip',
        content_disposition='attachment; filename="{fname}"'.format(fname=filename),
        download_id=download_id,
    )
    if use_transfer:
        expose_file_download(
            path,
            use_transfer=use_transfer,
            **common_kwargs
        )
    else:
        expose_cached_download(
            FileWrapper(open(path)),
            expiry=(1 * 60 * 60),
            **common_kwargs
        )
```

Saving uploads to the NFS drive

For files that are uploaded and require asynchronous processing e.g. imports, you can also use the NFS drive:

```
from soil.util import expose_file_download, expose_cached_download

uploaded_file = request.FILES.get('Filedata')
if hasattr(uploaded_file, 'temporary_file_path') and settings.SHARED_DRIVE_CONF.temp_
↪dir:
    path = settings.SHARED_DRIVE_CONF.get_temp_file()
    shutil.move(uploaded_file.temporary_file_path(), path)
    saved_file = expose_file_download(path)
else:
    uploaded_file.file.seek(0)
    saved_file = expose_cached_download(uploaded_file.file.read(), expiry=(60 * 60))

process_uploaded_file.delay(saved_file.download_id)
```

CHAPTER 25

How to use and reference forms and cases programatically

With the introduction of the new architecture for form and case data it is now necessary to use generic functions and accessors to access and operate on the models.

This document provides a basic guide for how to do that.

Models

In the codebase there are now two models for form and case data.

Couch	SQL
CommCareCase	CommCareCaseSQL
CommCareCaseAction	CaseTransaction
CommCareCaseAttachment	CaseAttachmentSQL
CommCareCaseIndex	CommCareCaseIndexSQL
XFormInstance	XFormInstanceSQL
	XFormAttachment
XFormOperation	XFormOperationSQL
StockReport	
StockTransaction	LedgerTransaction
StockState	LedgerValue

Some of these models define a common interface that allows you to perform the same operations irrespective of the type. Some examples are shown below:

Form Instance

Property / method	Description
form.form_id	The instance ID of the form
form.is_normal form.is_deleted form.is_archived form.is_error form.is_deprecated form.is_duplicate form.is_submission_error_log	Replacement for checking the doc_type of a form
form.attachments	The form attachment objects
form.get_attachment	Get an attachment by name
form.archive	Archive a form
form.unarchive	Unarchive a form
form.to_json	Get the JSON representation of a form
form.form_data	Get the XML form data

Case

Property / method	Description
case.case_id	ID of the case
case.is_deleted	Replacement for doc_type check
case.case_name	Name of the case
case.get_attachment	Get attachment by name
case.dynamic_case_properties	Dictionary of dynamic case properties
case.get_subcases	Get subcase objects
case.get_index_map	Get dictionary of case indices

Model acessors

To access models from the database there are classes that abstract the actual DB operations. These classes are generally names `<type>Accessors` and must be instantiated with a domain

name in order to know which DB needs to be queried.

Forms

- FormAccessors(domain).get_form(form_id)
- FormAccessors(domain).get_forms(form_ids)
- FormAccessors(domain).iter_forms(form_ids)
- FormAccessors(domain).save_new_form(form)
  - only for new forms
- FormAccessors(domain).get_with_attachments(form)
  - Preload attachments to avoid having to the the DB again

Cases

- CaseAccessors(domain).get_case(case_id)
- CaseAccessors(domain).get_cases(case_ids)
- CaseAccessors(domain).iter_cases(case_ids)
- CaseAccessors(domain).get_case_ids_in_domain(type='dog')

Ledgers

- LedgerAccessors(domain).get_ledger_values_for_case(case_id)

For more details see:

- `corehq.form_processor.interfaces.dbaccessors.FormAccessors`
- `corehq.form_processor.interfaces.dbaccessors.CaseAccessors`
- `corehq.form_processor.interfaces.dbaccessors.LedgerAccessors`

Branching

In special cases code may need to be branched into SQL and Couch versions. This can be accomplished using the `should_use_sql_backend(domain)` function.:

```
if should_use_sql_backend(domain_name):
    # do SQL specifc stuff here
else:
    # do couch stuff here
```

Unit Tests

In most cases tests that use form / cases/ ledgers should be run on both backends as follows:

```
@run_with_all_backends
def test_my_function(self):
    ...
```

If you really need to run a test on only one of the backends you can do the following:

```
@override_settings(TESTS_SHOULD_USE_SQL_BACKEND=True)
def test_my_test(self):
    ...
```

To create a form in unit tests use the following pattern:

```
from corehq.form_processor.tests.utils import run_with_all_backends
from corehq.form_processor.utils import get_simple_wrapped_form, TestFormMetadata

@run_with_all_backends
def test_my_form_function(self):
    # This TestFormMetadata specifies properties about the form to be created
    metadata = TestFormMetadata(
        domain=self.user.domain,
        user_id=self.user._id,
    )
    form = get_simple_wrapped_form(
        form_id,
        metadata=metadata
    )
```

Creating cases can be done with the `CaseFactory`:

```
from corehq.form_processor.tests.utils import run_with_all_backends
from casexml.apps.case.mock import CaseFactory

@run_with_all_backends
def test_my_case_function(self):
    factory = CaseFactory(domain='foo')
    factory.create_case(
        case_type='my_case_type',
        owner_id='owner1',
        case_name='bar',
        update={'prop1': 'abc'}
    )
```

Cleaning up

Cleaning up in tests can be done using the `FormProcessorTestUtils1` class:

```
from corehq.form_processor.tests.utils import FormProcessorTestUtils

def tearDown(self):
    FormProcessorTestUtils.delete_all_cases()
    # OR
    FormProcessorTestUtils.delete_all_cases(
        domain=domain
    )

    FormProcessorTestUtils.delete_all_xforms()
    # OR
    FormProcessorTestUtils.delete_all_xforms(
        domain=domain
    )
```

CHAPTER 26

Messaging in CommCareHQ

The term "messaging" in CommCareHQ commonly refers to the set of frameworks that allow the following types of use cases:

- sending SMS to contacts
- receiving SMS from contacts and performing pre-configured actions based on the content
- scheduling reminders to contacts
- creating alerts based on configurable criteria
- sending outbound calls to contacts and initiating an Interactive Voice Response (IVR) session
- collecting data via SMS surveys
- sending email alerts to contacts

The purpose of this documentation is to show how all of those use cases are performed technically by CommCareHQ. The topics below cover this material and should be followed in the order presented below if you have no prior knowledge of the messaging frameworks used in CommCareHQ.

26.1 Messaging Definitions

General Messaging Terms

SMS Gateway a third party service that provides an API for sending and receiving SMS

Outbound SMS an SMS that is sent from the SMS Gateway to a contact

Inbound SMS an SMS that is sent from a contact to the SMS Gateway

Mobile Terminating (MT) SMS an outbound SMS

Mobile Originating (MO) SMS an inbound SMS

Dual Tone Multiple Frequencies (DTMF) tones: the tones made by a telephone when pressing a button such as number 1, number 2, etc.

Interactive Voice Response (IVR) Session: a phone call in which the user is prompted to make choices using DTMF tones and the flow of the call can change based on those choices

IVR Gateway a third party service that provides an API for handling IVR sessions

International Format (also referred to as E.164 Format) for a Phone Number: a format for a phone number which makes it so that it can be reached from any other country; the format typically starts with +, then the country code, then the number, though there may be some subtle operations to perform on the number before putting into international format, such as removing a leading zero

SMS Survey a way of collecting data over SMS that involves asking questions one SMS at a time and waiting for a contact's response before sending the next SMS

Structured SMS a way for collecting data over SMS that involves collecting all data points in one SMS rather than asking one question at a time as in an SMS Survey; for example: "REGISTER Joe 25" could be one way to define a Structured SMS that registers a contact named Joe whose age is 25.

Messaging Terms Commonly Used in CommCareHQ

SMS Backend the code which implements the API of a specific SMS Gateway

IVR Backend the code which implements the API of a specific IVR Gateway

Two-way Phone Number a phone number that the system has tied to a single contact in a single domain, so that the system can not only send oubound SMS to the contact, but the contact can also send inbound SMS and have the system process it accordingly; the system currently only considers a number to be two-way if there is a corehq.apps.sms.models.PhoneNumber entry for it that has verified = True

One-way Phone Number a phone number that has not been tied to a single contact, so that the system can only send outbound SMS to the number; one-way phone numbers can be shared across many contacts in many domains, but only one of those numbers can be a two-way phone number

Contacts

A contact is a single person that we want to interact with through messaging. In CommCareHQ, at the time of writing, contacts can either be users (CommCareUser, WebUser) or cases (CommCareCase).

In order for the messaging frameworks to interact with a contact, the contact must implement the corehq.apps.sms.mixin.CommCareMobileContactMixin.

Contacts have phone numbers which allows CommCareHQ to interact with them. All phone numbers for contacts must be stored in International Format, and the frameworks always assume a phone number is given in International Format.

Regarding the + sign before the phone number, the rule of thumb is to never store the + when storing phone numbers, and to always display it when displaying phone numbers.

Users

A user's phone numbers are stored as the phone_numbers attribute on the CouchUser class, which is just a list of strings.

At the time of writing, WebUsers are only allowed to have one-way phone numbers.

CommCareUsers are allowed to have two-way phone numbers, but in order to have a phone number be considered to be a two-way phone number, it must first be verified. The verification process is initiated

on the edit mobile worker page and involves sending an outbound SMS to the phone number and having it be acknowledged by receiving a validated response from it.

Cases

At the time of writing, cases are allowed to have only one phone number. The following case properties are used to define a case's phone number:

contact_phone_number the phone number, in International Format

contact_phone_number_is_verified must be set to 1 in order to consider the phone number a two-way phone number; the point here is that the health worker registering the case should verify the phone number and the form should set this case property to 1 if the health worker has identified the phone number as verified

If two cases are registered with the same phone number and both set the verified flag to 1, it will only be granted two-way phone number status to the case who registers it first.

If a two-way phone number can be granted for the case, a corehq.apps.sms.models.PhoneNumber entry with verified set to True is created for it. This happens automatically by running celery task corehq.apps.sms.tasks.sync_case_phone_number for a case each time a case is saved.

Future State

Forcing the verification workflows before granting a phone number two-way phone number status has proven to be challenging for our users. In a (hopefully soon) future state, we will be doing away with all verification workflows and automatically consider a phone number to be a two-way phone number for the contact who registers it first.

Outbound SMS

The SMS framework uses a queuing architecture to make it easier to scale SMS processing power horizontally.

The process to send an SMS from within the code is as follows. The only step you need to do is the first, and the rest happen automatically.

1. **Invoke one of the send_sms* functions found in corehq.apps.sms.api:**

   send_sms used to send SMS to a one-way phone number represented as a string

   send_sms_to_verified_number use to send SMS to a two-way phone number represented as a PhoneNumber object

   send_sms_with_backend used to send SMS with a specific SMS backend

   send_sms_with_backend_name used to send SMS with the given SMS backend name which will be resolved to an SMS backend

2. The framework creates a corehq.apps.sms.models.QueuedSMS object representing the SMS to be sent.
3. The SMS Queue polling process (python manage.py run_sms_queue), which runs as a supervisor process on one of the celery machines, picks up the QueuedSMS object and passes it to corehq.apps.sms.tasks.process_sms.

4. process_sms attempts to send the SMS. If an error happens, it is retried up to 2 more times on 5 minute intervals. After 3 total attempts, any failure causes the SMS to be marked with error = True.
5. Whether the SMS was processed successfully or not, the QueuedSMS object is deleted and replaced by an identical looking corehq.apps.sms.models.SMS object for reporting.

At a deeper level, process_sms performs the following important functions for outbound SMS. To find out other more detailed functionality provided by process_sms, see the code.

1. If the domain has restricted the times at which SMS can be sent, check those and requeue the SMS if it is not currently an allowed time.
2. **Select an SMS backend by looking in the following order:**
    - If using a two-way phone number, look up the SMS backend with the name given in the backend_id property
    - If the domain has a default SMS backend specified, use it
    - Look up an appropriate global SMS backend by checking the phone number's prefix against the global SQLMobileBackendMapping entries
    - Use the catch-all global backend (found from the global SQLMobileBackendMapping entry with prefix = '*')
3. If the SMS backend has configured rate limiting or load balancing across multiple numbers, enforce those constraints.
4. Pass the SMS to the send() method of the SMS Backend, which is an instance of corehq.apps.sms.models.SQLSMSBackend.

Inbound SMS

Inbound SMS uses the same queueing architecture as outbound SMS does.

The entry point to processing an inbound SMS is the corehq.apps.sms.api.incoming function. All SMS backends which accept inbound SMS call the incoming function.

From there, the following functions are performed at a high level:

1. The framework creates a corehq.apps.sms.models.QueuedSMS object representing the SMS to be processed.
2. The SMS Queue polling process (python manage.py run_sms_queue), which runs as a supervisor process on one of the celery machines, picks up the QueuedSMS object and passes it to corehq.apps.sms.tasks.process_sms.
3. process_sms attempts to process the SMS. If an error happens, it is retried up to 2 more times on 5 minute intervals. After 3 total attempts, any failure causes the SMS to be marked with error = True.
4. Whether the SMS was processed successfully or not, the QueuedSMS object is deleted and replaced by an identical looking corehq.apps.sms.models.SMS object for reporting.

At a deeper level, process_sms performs the following important functions for inbound SMS. To find out other more detailed functionality provided by process_sms, see the code.

1. Look up a two-way phone number for the given phone number string.
2. If a two-way phone number is found, pass the SMS on to each inbound SMS handler (defined in settings.SMS_HANDLERS) until one of them returns True, at which point processing stops.

3. If a two-way phone number is not found, try to pass the SMS on to the SMS handlers that don't require two-way phone numbers (the phone verification workflow, self-registration over SMS workflows)

SMS Backends

We have one SMS Backend class per SMS Gateway that we make available.

SMS Backends are defined by creating a new directory under corehq.messaging.smsbackends, and the code for each backend has two main parts:

- The outbound part of the backend which is represented by a class that subclasses corehq.apps.sms.models.SQLSMSBackend
- The inbound part of the backend which is represented by a view that subclasses corehq.apps.sms.views.IncomingBackendView

Outbound

The outbound part of the backend code is responsible for interacting with the SMS Gateway's API to send an SMS.

All outbound SMS backends are subclasses of SQLSMSBackend, and you can't use a backend until you've created an instance of it and saved it in the database. You can have multiple instances of backends, if for example, you have multiple accounts with the same SMS gateway.

Backend instances can either be global, in which case they are shared by all projects in CommCareHQ, or they can belong to a specific project. If belonged to a specific project, a backend can optionally be shared with other projects as well.

To write the outbound backend code:

1. Create a subclass of corehq.apps.sms.models.SQLSMSBackend and implement the unimplemented methods:

   get_api_id should return a string that uniquely identifies the backend type (but is shared across backend instances); we choose to not use the class name for this since class names can change but the api id should never change; the api id is only used for sms billing to look up sms rates for this backend type

   get_generic_name a displayable name for the backend

   get_available_extra_fields each backend likely needs to store additional information, such as a username and password for authenticating with the SMS gateway; list those fields here and they will be accessible via the backend's config property

   get_form_class should return a subclass of corehq.apps.sms.forms.BackendForm, which should:

   - have form fields for each of the fields in get_available_extra_fields, and
   - implement the gateway_specific_fields property, which should return a crispy forms rendering of those fields

   send takes a corehq.apps.sms.models.QueuedSMS object as an argument and is responsible for interfacing with the SMS Gateway's API to send the SMS; if you want the framework to retry the SMS, raise an exception in this method, otherwise if no exception is raised the framework takes that to mean the process was successful

2. Add the backend to sms.SMS_LOADED_SQL_BACKENDS
3. Add an outbound test for the backend in corehq.apps.sms.tests.test_backends. This will test that the backend is reachable by the framework, but any testing of the direct API connection with the gateway must be tested manually.

Once that's done, you should be able to create instances of the backend by navigating to Messaging -> SMS Connectivity (for domain-level backend instances) or Admin -> SMS Connectivity and Billing (for global backend instances). To test it out, set it as the default backend for a project and try sending an SMS through the Compose SMS interface.

Things to look out for:

- Make sure you use the proper encoding of the message when you implement the send() method. Some gateways are picky about the encoding needed. For example, some require everything to be UTF-8. Others might make you choose between ASCII and Unicode. And for the ones that accept Unicode, you might need to sometimes convert it to a hex representation. And remember that get/post data will be automatically url-encoded when you use python requests. Consult the documentation for the gateway to see what is required.
- The message limit for a single SMS is 160 7-bit structures. That works out to 140 bytes, or 70 words. That means the limit for a single message is typically 160 GSM characters, or 70 Unicode characters. And it's actually a little more complicated than that since some simple ASCII characters (such as '{') take up two GSM characters, and each carrier uses the GSM alphabet according to language.

  So the bottom line is, it's difficult to know whether the given text will fit in one SMS message or not. As a result, you should find out if the gateway supports Concatenated SMS, a process which seamlessly splits up long messages into multiple SMS and stiches them back up without you having to do any additional work. You may need to have the gateway enable a setting to do this or include an additional parameter when sending SMS to make this work.

Inbound

The inbound part of the backend code is responsible for exposing a view which implements the API that the SMS Gateway expects so that the gateway can connect to CommCareHQ and notify us of inbound SMS.

To write the inbound backend code:

1. Create a subclass of corehq.apps.sms.views.IncomingBackendView, and implement the unimplemented property:

   backend_class should return the subclass of SQLSMSBackend that was written above

2. Implement either the get() or post() method on the view based on the gateway's API. The only requirement of the framework is that this method call the corehq.apps.sms.api.incoming function, but you should also:

   - pass self.backend_couch_id as the backend_id kwarg to incoming()
   - if the gateway gives you a unique identifier for the SMS in their system, pass that identifier as the backend_message_id kwarg to incoming(); this can help later with debugging

3. Create a url for the view. The url pattern should accept an api key and look something like: r'^sms/(?P<api_key>[w-]+)/$' . The API key used will need to match the inbound_api_key of a backend instance in order to be processed.

4. Let the SMS Gateway know the url to connect to, including the API Key. To get the API Key, look at the value of the inbound_api_key property on the backend instance. This value is generated automatically when you first create a backend instance.

What happens behind the scenes is as follows:

1. A contact sends an inbound SMS to the SMS Gateway
2. The SMS Gateway connects to the URL configured above.
3. The view automatically looks up the backend instance by api key and rejects the request if one is not found.
4. Your get() or post() method is invoked which parses the parameters accordingly and passes the information to the inbound incoming() entry point.
5. The Inbound SMS framework takes it from there as described in the Inbound SMS section.

NOTE: The api key is part of the URL because it's not always easy to make the gateway send us an extra arbitrary parameter on each inbound SMS.

Rate Limiting

You may want (or need) to limit the rate at which SMS get sent from a given backend instance. To do so, just override the get_sms_rate_limit() method in your SQLSMSBackend, and have it return the maximum number of SMS that can be sent in a one minute period.

Load Balancing

If you want to load balance the Outbound SMS traffic automatically across multiple phone numbers, do the following:

1. Make your BackendForm subclass the corehq.apps.sms.forms.LoadBalancingBackendFormMixin
2. Make your SQLSMSBackend subclass the corehq.apps.sms.models.PhoneLoadBalancingMixin
3. Make your SQLSMSBackend's send method take a orig_phone_number kwarg. This will be the phone number to use when sending. This is always sent to the send() method, even if there is just one phone number to load balance over.

From there, the framework will automatically handle managing the phone numbers through the create/edit gateway UI and balancing the load across the numbers when sending. A simple round robin approach is taken when balancing the load.

If your backend uses load balancing and rate limiting, the framework applies the rate limit to each phone number separately as you would expect.

Reminders

The Reminders framework uses a queuing architecture similar to the SMS framework, to make it easier to scale reminders processing power horizontally.

To see how this works, we first have to see how the reminders models are setup.

Reminder Definition

A reminder definition, represented by a corehq.apps.reminders.models.CaseReminderHandler object, defines the rules for:

- what criteria cause a reminder to be triggered
- when the reminder should start once the criteria are fulfilled
- who the reminder should go to
- on what schedule and frequency the reminder should continue to be sent
- the content to send
- what causes the reminder to stop

Reminder Instance

A reminder instance, represented by a corehq.apps.reminders.models.CaseReminder, defines an instance of a reminder definition and keeps track of the state of the reminder instance throughout its lifetime.

For example, a reminder definition may define a rule for sending an SMS to a case of type patient, and sending an SMS appointment reminder to the case 2 days before the case's appointment_date case property.

As soon as a case is created or updated in the given project to meet the criteria of having type patient and having an appointment_date, the framework will create a reminder instance to track it. After the reminder is sent 2 days before the appointment_date, the reminder instance is deactivated to denote that it has completed the defined schedule and should not be sent again.

In order to keep reminder instances responsive to case changes, every time a case is saved, a corehq.apps.reminders.tasks.case_changed task is spawned to handle any changes. Similarly, any time a reminder definition is updated, a corehq.apps.reminders.tasks.process_reminder_rule task is spawned to rerun it against all cases in the project.

The aim of the framework is to always be completely responsive to all changes. So in the example above, if a case's appointment_date changes before the appointment reminder is actually sent, the framework will update the reminder instance automatically in order to reflect the new appointment date. And if the appointment reminder went out months ago but a new appointment_date value is given to the case for a new appointment, the same reminder instance is updated again to reflect a new reminder that must go out.

Similarly, if the reminder definition is updated to use a different case property other than appointment_date, all existing reminder instances are deleted and any new ones are created if they meet the criteria.

Queueing

All of the reminder instances in the database represent the queue of reminders that should be sent. The way a reminder is processed is as follows:

1. The reminder polling process (python manage.py run_reminder_queue), which runs as a supervisor process on one of the celery machines, constantly polls for reminders that should be processed by querying for reminder instances that have a next_fire property that is in the past.
2. Once a reminder that needs to be processed has been identified, the framework spawns a corehq.apps.reminders.tasks.fire_reminder task to handle it.

3. fire_reminder looks up the reminder definition that spawned the reminder instance, and instructs it to 1) take the appropriate action that has been configured (for example, send an sms), and 2) update the state of the reminder instance so that it gets scheduled for the next action it must take based on the reminder definition.

Event Handlers

A reminder definition sends content of one type. At the time of writing, the content a reminder definition can be configured to send includes:

- SMS
- SMS Survey
- Outbound IVR Session
- Emails

In the case of SMS Surveys or IVR Sessions, the survey content is defined using a form in an app which is then played to the recipients over SMS or IVR using touchforms (see corehq.apps.smsforms for this interface with touchforms).

New event handlers can be written and added to the current ones in corehq.apps.reminders.event_handlers, and each event handler is tied to a reminder definition through the reminder definition's method attribute and the corehq.apps.reminders.event_handlers.EVENT_HANDLER_MAP.

Keywords

A Keyword (corehq.apps.sms.models.Keyword) defines an action or set of actions to be taken when an inbound SMS is received whose first word matches the keyword configuration.

Any number of actions can be taken, which include:

- Replying with an SMS or SMS Survey
- Sending an SMS or SMS Survey to another contact or group of contacts
- Processing the SMS as a Structured SMS

Keywords tie into the Inbound SMS framework through the keyword handler (corehq.apps.sms.handlers.keyword.sms_keyword_handler, see settings.SMS_HANDLERS), and use the Reminders framework to carry out their action(s).

Behind the scenes, all actions besides processing Structured SMS create a reminder definition to be sent immediately. So any functionality provided by a reminder definition can be added to be supported as a Keyword action.

CHAPTER 27

Locations

CHAPTER 28

Tips for documenting

Documenting

Documentation is awesome. You should write it. Here's how.

All the CommCareHQ docs are stored in a `docs/` folder in the root of the repo. To add a new doc, make an appropriately-named rst file in the `docs/` directory. For the doc to appear in the table of contents, add it to the `toctree` list in `index.rst`.

Sooner or later we'll probably want to organize the docs into sub-directories, that's fine, you can link to specific locations like so: `` `Installation <intro/install>` ``.

For a more complete working set of documentation, check out Django's docs directory. This is used to build docs.djangoproject.com.

Index

1. *Sphinx* is used to build the documentation.
2. *Writing Documentation* - Some general tips for writing documentation
3. *reStructuredText* is used for markup.
4. *Editors* with RestructuredText support

Sphinx

Sphinx builds the documentation and extends the functionality of rst a bit for stuff like pointing to other files and modules.

To build a local copy of the docs (useful for testing changes), navigate to the `docs/` directory and run `make html`. Open `<path_to_commcare-hq>/docs/_build/html/index.html` in your browser and you should have access to the docs for your current version (I bookmarked it on my machine).

- Sphinx Docs

- Full index

Writing Documentation

For some great references, check out Jacob Kaplan-Moss's series Writing Great Documentation and this blog post by Steve Losh. Here are some takeaways:

- Use short sentences and paragraphs
- Break your documentation into sections to avoid text walls
- Avoid making assumptions about your reader's background knowledge
- Consider three types of documentation:
  1. Tutorials - quick introduction to the basics
  2. Topical Guides - comprehensive overview of the project; everything but the dirty details
  3. Reference Material - complete reference for the API

One aspect that Kaplan-Moss doesn't mention explicitly (other than advising us to "Omit fluff" in his Technical style piece) but is clear from both his documentation series and the Django documentation, is *what not to write*. It's an important aspect of the readability of any written work, but has other implications when it comes to technical writing.

Antoine de Saint Exupéry wrote, "... perfection is attained not when there is nothing more to add, but when there is nothing more to remove."

Keep things short and take stuff out where possible. It can help to get your point across, but, maybe more importantly with documentation, means there is less that needs to change when the codebase changes.

Think of it as an extension of the DRY principle.

reStructuredText

reStructuredText is a markup language that is commonly used for Python documentation. You can view the source of this document or any other to get an idea of how to do stuff (this document has hidden comments). Here are some useful links for more detail:

- rst quickreference
- Sphinx guide to rst
- reStructuredText full docs
- Referencing arbitrary locations and other documents

Editors

While you can use any text editor for editing RestructuredText documents, I find two particularly useful:

- PyCharm (or other JetBrains IDE, like IntelliJ), which has great syntax highlighting and linting.
- Sublime Text, which has a useful plugin for hard-wrapping lines called Sublime Wrap Plus. Hard-wrapped lines make documentation easy to read in a console, or editor that doesn't soft-wrap lines (i.e. most code editors).
- Vim has a command `gq` to reflow a block of text (`:help gq`). It uses the value of `textwidth` to wrap (`:setl tw=75`). Also check out `:help autoformat`. Syntastic has a rst linter. To make a line a header, just `yypVr=` (or whatever symbol you want).

Examples

Some basic examples adapted from 2 Scoops of Django:

Section Header

Sections are explained well here

emphasis (bold/strong)

italics

Simple link: http://commcarehq.org

Inline link: CommCareHQ

Fancier Link: CommCareHQ

1. An enumerated list item
2. Second item

- First bullet
- **Second bullet**
  - Indented Bullet
  - Note carriage return and indents

Literal code block:

```
def like():
    print("I like Ice Cream")

for i in range(10):
    like()
```

Python colored code block (requires pygments):

```
# You need to "pip install pygments" to make this work.

for i in range(10):
    like()
```

JavaScript colored code block:

```
console.log("Don't use alert()");
```

CHAPTER 29

Indices and tables

- genindex
- modindex
- search