CommCareHQ Documentation

Release 1.0

Dimagi

Jun 26, 2019

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.

1.1 Recommended approaches for building reports

Things to keep in mind:

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

1.1.1 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"))
```

(continues on next page)

(continued from previous page)

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

1.2 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,
    )),
)
```

1.3 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.

1.3.1 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.

class corehq.apps.reports.sqlreport.**SqlData**(*config=None*)

columns
Returns a list of Column objects. These are used to make up the from portion of the SQL query.

filter_values
Return a dict mapping the filter keys to actual values e.g. {"enddate": date(2013, 1, 1)}

filters
Returns a list of filter statements. Filters are instances of sqlagg.filters.SqlFilter. See the sqlagg.filters module for a list of standard filters.

e.g. [EQ('date', 'enddate')]

group_by
Returns a list of 'group by' column names.

keys
The list of report keys (e.g. users) or None to just display all the data returned from the query. Each value in this list should be a list of the same dimension as the 'group_by' list. If group_by is None then keys must also be None.

These allow you to specify which rows you expect in the output data. Its main use is to add rows for keys that don't exist in the data.

e.g. group_by = ['region', 'sub_region'] keys = [['region1', 'sub1'], ['region1', 'sub2'] ...]

table_name = None
The name of the table to run the query against.

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']]
```

(continues on next page)

(continued from previous page)

```
    @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)
```

1.4 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.

class corehq.apps.reports.api.**ReportDataSource**(*config=None*)

get_data(*start=None*, *limit=None*)
Intention: Override

Parameters **slugs** – List of slugs to return for each row. Return all values if slugs = None or [].

Returns A list of dictionaries mapping slugs to values.

e.g. [{'village': 'Mazu', 'births': 30, 'deaths': 28},{...}]

slugs()
Intention: Override

Returns A list of available slugs.

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()
```

1.5 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.

1.6 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

2.1 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)

2.2 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.

2.2.1 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).

2.2.2 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.

2.3 Publishing changes

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

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.4 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.

2.5 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

3.1 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.

3.2 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

3.2.1 Change Feed

Change feeds are documented in the Changes Feed section available on the left.

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

3.2.2 Checkpoint

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

3.2.3 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.

3.2.4 Change Event Handler

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

3.3 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.

3.4 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.

3.5 Troubleshooting

3.5.1 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

To scale UCR Pillows horizontally do the following:

1. Look for what pillows are behind. This can be found in the change feed dashboard or the hq admin system info page.
2. Ensure you have enough resources on the pillow server to scale the pillows This can be found through datadog.
3. Decide what topics need to have added partitions in kafka. There is no way to scale a couch pillow horizontally. You can also not remove partitions so you should attempt scaling in small increments. Also attempt to make sure pillows are able to split partitions easily. It's easiest to use powers of 2
4. Run *./manage.py add_kafka_partition <topic> <number partitions to have>*
5. In the commcare-cloud repo environments/<env>/app-processes.yml file change num_processes to the pillows you want to scale.
6. On the next deploy multiple processes will be used when starting pillows

The UCR pillows also 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.

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>*

3.5.2 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

4.1 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
```

4.1.1 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

5.1 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)

5.2 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`).

5.3 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*

5.4 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.

5.4.1 `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)

5.4.2 `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)

5.4.3 `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.

5.4.4 `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).

5.5 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).

5.6 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

Exports

Docs in corehq/apps/export/README.md

CHAPTER 7

UI Helpers

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

7.1 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.

7.1.1 The Basic Paginated View

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

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedViewMixin):
    # your template should extend hqwebapp/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_message = "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
```

(continues on next page)

(continued from previous page)

```
    @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 page_context(self):
        # This should at least include the pagination_context that␣
↪CRUDPaginatedViewMixin provides
        return self.pagination_context

    @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.paginate_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 'hqwebapp/base_paginated_crud.html' %}

{% block pagination_templates %}
<script type="text/html" id="base-puppy-template">
```

(continues on next page)

(continued from previous page)

```
    <td data-bind="text: name"></td>
    <td data-bind="text: breed"></td>
    <td data-bind="text: age"></td>
</script>
{% endblock %}
```

7.1.2 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"),
```

(continues on next page)

(continued from previous page)

```
                css_class='btn-primary',
                type='submit'
            )
        )

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

7.1.3 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',
```

(continues on next page)

(continued from previous page)

```
        }
```

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">
```

(continues on next page)

(continued from previous page)

```
                <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>
```

7.1.4 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="fa fa-remove"></i> Delete Puppy
        </button>

        <div class="modal fade"
             data-bind="
                attr: {
                    id: 'delete-puppy-' + id
                }
             ">
             <div class="modal-dialog">
                <div class="modal-content">
                    <div class="modal-header">
```

(continues on next page)

(continued from previous page)

```
                        <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 class="lead">
                            Yes, delete the puppy named <strong data-bind="text: name
↪"></strong>.
                        </p>
                    </div>
                    <div class="modal-footer">
                        <button type="button"
                                class="btn btn-default"
                                data-dismiss="modal">
                            Cancel
                        </button>
                        <button type="button"
                                class="btn btn-danger delete-item-confirm"
                                data-loading-text="Deleting Puppy...">
                            <i class="fa fa-remove"></i> Delete Puppy
                        </button>
                    </div>
                </div>
            </div>
        </div>
    </td>
</script>
```

7.1.5 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 8

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.

8.1 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).

8.1.1 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.
```

(continues on next page)

(continued from previous page)

```
        # 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 *hqwebapp/base_page.html*

It might look something like:

```
{% extends 'hqwebapp/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 %}
```

8.1.2 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 *hqwebapp/base_section.html*

It might look something like:

```
{% extends 'hqwebapp/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 *hqwebapp/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.

8.2 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),
)
```

8.3 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
```

8.4 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)
```

8.5 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...
```

(continues on next page)

(continued from previous page)

```
        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
```

8.5.1 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 9

Testing best practices

9.1 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):
```

(continues on next page)

(continued from previous page)

```
    @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.

9.2 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.

9.3 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.

9.4 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 10

Forms in HQ

See the HQ Style Guide for guidance on form UI, whether you're creating a custom HTML form or using crispy forms.

10.1 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 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
6. 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
7. 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
8. 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.

CHAPTER 11

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_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>

make_superuser Make a new superuser or make an existing user a superuser. Usage:: $ ./manage.py make_superuser [options] <email>

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 12

Migrating Database Definitions

There are currently three persistent data stores in CommCare that can be migrated. Each of these have slightly different steps that should be followed.

12.1 General

For all ElasticSearch and CouchDB changes, add a "reindex/migration" flag to your PR. These migrations generally have some gotchas and require more planning for deploy than a postgres migration.

12.2 Adding Data

12.2.1 Postgres

Add the column as a nullable column. Creating NOT NULL constraints can lock the table and take a very long time to complete. If you wish to have the column be NOT NULL, you should add the column as nullable and migrate data to have a value before adding a NOT NULL constraint.

12.2.2 ElasticSearch

You only need to add ElasticSearch mappings if you want to search by the field you are adding. There are two ways to do this:

a. Change the mapping's name, add the field, and using ptop_preindex.

b. Add the field, reset the mapping, and using ptop_preindex with an *in-place* flag.

If you change the mapping's name, you should add reindex/migration flag to your PR and coordinate your PR to run ptop_preindex in a private release directory. Depending on the index and size, this can take somewhere between minutes and days.

12.2.3 CouchDB

You can add fields as needed to couch documents, but take care to handle the previous documents not having this field defined.

12.3 Removing Data

12.3.1 General

Removing columns, fields, SQL functions, or views should always be done in multiple steps.

1. Remove any references to the field/function/view in application code
2. Wait until this code has been deployed to all relevant environments.
3. Remove the column/field/function/view from the database.

It's generally not enough to remove these at the same time because any old processes could still reference the to be deleted entity.

12.3.2 Couch

A separate prune_couch_views will need to be run to remove the view from couch

12.3.3 ElasticSearch

If you're removing an index, you can use *prune_es_indices* to remove all indices that are no longer referenced in code.

12.4 Querying Data

12.4.1 Postgres

Creating an index can lock the table and cause it to not respond to queries. If the table is large, an index is going to take a long time. In that case:

1. Create the migration normally using django.
2. On all large environments, create the index concurrently. One way to do this is to use ./manage.py run_sql ... to apply the SQL to the database.
3. Once finished, fake the migration. Avoid this by using *CREATE INDEX IF NOT EXISTS ...* in the migration if possible.
4. Merge your PR.

12.4.2 Couch

Changing views can block our deploys due to the way we sync our couch views. If you're changing a view, please sync with someone else who understands this process and coordinate with the team to ensure we can rebuild the view without issue.

12.5 Migration Patterns and Best Practices

- auto-managed-migration-pattern

CHAPTER 13

CommTrack

13.1 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 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

13.2 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 14

CloudCare

14.1 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.

14.1.1 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.

14.1.2 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.

14.2 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.

14.3 Offline Cloudcare

14.3.1 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!*

14.3.2 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.

14.3.3 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 15

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.

15.1 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}."
```

15.1.1 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
```

(continues on next page)

(continued from previous page)

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

15.2 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") %}
```

15.3 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 16

Profiling

16.1 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

16.1.1 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
```

16.1.2 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.

16.1.3 Profiling in production

The same method can be used to profile functions in production. Obviously we want to be able to turn this on and off and possibly only profile a limited number of function calls.

This can be accomplished by using an environment variable to set the probability of profiling a function. Here's an example:

```
@profile_prod('locations_download.prof', probability=float(os.getenv('PROFILE_
↪LOCATIONS_EXPORT', 0))
def location_export(request, domain):
    ....
```

By default this wil not do any profiling but if the *PROFILE_LOCATIONS_EXPORT* environment variable is set to a value between 0 and 1 and the Django process is restarted then the function will get profiled. The number of profiles that are done will depend on the value of the environment variable. Values closer to 1 will get more profiling.

You can also limit the total number of profiles to be recorded using the *limit* keyword argument. You could also expose this via an environment variable or some other method to make it configurable:

```
@profile_prod('locations_download.prof', 1, limit=10)
def location_export(request, domain):
    ....
```

Warning: In a production environment the *limit* may not apply absolutely since there are likely multiple processes running in which case the limit will get applied to each one. Also, the limit will be reset if the processes are restarted.

Any profiling in production should be closely monitored to ensure that it does not adversely affect performance or fill up available disk space.

16.1.4 Creating a more useful output from the dump file

The raw profile files are not human readable, and you need to use something like cProfile 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
```

16.1.5 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)
```

(continues on next page)

(continued from previous page)

```
      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)
```

See also:

Description of columns

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__)
```

(continues on next page)

(continued from previous page)

```
        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.

16.1.6 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 to aggregate the data.

This will produce a file which can be analysed with the convert_profile.py script.

16.1.7 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

16.1.8 Additional references

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

16.2 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)]
```

(continues on next page)

(continued from previous page)

```
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 17

ElasticSearch

17.1 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.

17.2 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
```

17.3 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.

17.3.1 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.

17.3.2 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.

17.3.3 Updating indexes in a production environment

Updates in a production environment should be done in two steps, so to not show incomplete data.

1. Setup a release of your branch using cchq <env> setup_limited_release:keep_days=n_days
2. In your release directory, kick off a index using `./mange.py ptop_preindex`
3. Verify that the reindex has completed successfully - This is a weak point in our current migration process - This can be done by using ES head or the ES APIs to compare document counts to the previous index. - You should also actively look for errors in the ptop_preindex command that was ran
4. Merge your PR and deploy your latest master branch.

17.4 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

17.5 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.

17.5.1 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.

17.5.2 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
```

17.5.3 Prefer scroll queries

Use a scroll query when fetching lots of records.

17.5.4 Prefer filter to query

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

17.5.5 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 18

ESQuery

18.1 ESQuery

ESQuery is a library for building elasticsearch queries in a friendly, more readable manner.

18.1.1 Basic usage

There should be a file and subclass of ESQuery for each index we have.

Each method returns a new object, so you can chain calls together like SQLAlchemy. Here's an example usage:

```
q = (FormsES()
     .domain(self.domain)
     .xmlns(self.xmlns)
     .submitted(gte=self.datespan.startdate_param,
                lt=self.datespan.enddateparam)
     .fields(['xmlns', 'domain', 'app_id'])
     .sort('received_on', desc=False)
     .size(self.pagination.count)
     .start(self.pagination.start)
     .terms_aggregation('babies.count', 'babies_saved'))
result = q.run()
total_docs = result.total
hits = result.hits
```

Generally useful filters and queries should be abstracted away for re-use, but you can always add your own like so:

```
q.filter({"some_arbitrary_filter": {...}})
q.set_query({"fancy_query": {...}})
```

For debugging or more helpful error messages, you can use `query.dumps()` and `query.pprint()`, both of which use `json.dumps()` and are suitable for pasting in to ES Head or Marvel or whatever

18.1.2 Filtering

Filters are implemented as standalone functions, so they can be composed and nested `q.OR(web_users(), mobile_users())`. Filters can be passed to the `query.filter` method: `q.filter(web_users())`

There is some syntactic sugar that lets you skip this boilerplate and just call the filter as if it were a method on the query class: `q.web_users()` In order to be available for this shorthand, filters are added to the `builtin_filters` property of the main query class. I know that's a bit confusing, but it seemed like the best way to make filters available in both contexts.

Generic filters applicable to all indices are available in `corehq.apps.es.filters`. (But most/all can also be accessed as a query method, if appropriate)

18.1.3 Filtering Specific Indices

There is a file for each elasticsearch index (if not, feel free to add one). This file provides filters specific to that index, as well as an appropriately-directed ESQuery subclass with references to these filters.

These index-specific query classes also have default filters to exclude things like inactive users or deleted docs. These things should nearly always be excluded, but if necessary, you can remove these with `remove_default_filters`.

18.1.4 Running against production

Since the ESQuery library is read-only, it's mostly safe to run against production. You can define alternate elasticsearch hosts in your localsettings file in the `ELASTICSEARCH_DEBUG_HOSTS` dictionary and pass in this host name as the `debug_host` to the constructor:

```
>>> CaseES(debug_host='prod').domain('dimagi').count()
120
```

18.1.5 Language

- es_query - the entire query, filters, query, pagination
- filters - a list of the individual filters
- query - the query, used for searching, not filtering
- field - a field on the document. User docs have a 'domain' field.
- lt/gt - less/greater than
- lte/gte - less/greater than or equal to

class corehq.apps.es.es_query.**ESQuery**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)

This query builder only outputs the following query structure:

```
{
    "query": {
        "filtered": {
            "filter": {
                "and": [
                    <filters>
                ]
            },
```

(continues on next page)

(continued from previous page)

```
            "query": <query>
        }
    },
    <size, sort, other params>
}
```

__init__(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)
x.__init__(...) initializes x; see help(type(x)) for signature

add_query(*new_query*, *clause*)
Add a query to the current list of queries

aggregation(*aggregation*)
Add the passed-in aggregation to the query

builtin_filters
A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

count()
Performs a minimal query to get the count of matching documents

dumps(*pretty=False*)
Returns the JSON query that will be sent to elasticsearch.

exclude_source()
Turn off _source retrieval. Mostly useful if you just want the doc_ids

fields(*fields*)
Restrict the fields returned from elasticsearch

Deprecated. Use *source* instead.

filter(*filter*)
Add the passed-in filter to the query. All filtering goes through this class.

filters
Return a list of the filters used in this query, suitable if you want to reproduce a query with additional filtering.

get_ids()
Performs a minimal query to get the ids of the matching documents

For very large sets of IDs, use `scroll_ids` instead

nested_sort(*path*, *field_name*, *nested_filter*, *desc=False*, *reset_sort=True*)
Order results by the value of a nested field

pprint()
pretty prints the JSON query that will be sent to elasticsearch.

remove_default_filter(*default*)
Remove a specific default filter by passing in its name.

remove_default_filters()
Sensible defaults are provided. Use this if you don't want 'em

run(*include_hits=False*)
Actually run the query. Returns an ESQuerySet object.

scroll()
Run the query against the scroll api. Returns an iterator yielding each document that matches the query.

scroll_ids()
Returns a generator of all matching ids

search_string_query(*search_string*, *default_fields=None*)
Accepts a user-defined search string

set_query(*query*)
Set the query. Most stuff we want is better done with filters, but if you actually want Levenshtein distance or prefix querying...

set_sorting_block(*sorting_block*)
To be used with *get_sorting_block*, which interprets datatables sorting

size(*size*)
Restrict number of results returned. Analagous to SQL limit.

sort(*field*, *desc=False*, *reset_sort=True*)
Order the results by field.

source(*include*, *exclude=None*)
Restrict the output of _source in the queryset. This can be used to return an object in a queryset

start(*start*)
Pagination. Analagous to SQL offset.

values(**fields*)
modeled after django's QuerySet.values

class corehq.apps.es.es_query.**ESQuerySet**(*raw*, *query*)

The object returned from `ESQuery.run`

- ESQuerySet.raw is the raw response from elasticsearch
- ESQuerySet.query is the ESQuery object

__init__(*raw*, *query*)
x.__init__(...) initializes x; see help(type(x)) for signature

doc_ids
Return just the docs ids from the response.

hits
Return the docs from the response.

static normalize_result(*query*, *result*)
Return the doc from an item in the query response.

total
Return the total number of docs matching the query.

class corehq.apps.es.es_query.**HQESQuery**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)
Query logic specific to CommCareHQ

18.2 Available Filters

The following filters are available on any ESQuery instance - you can chain any of these on your query.

Note also that the `term` filter accepts either a list or a single element. Simple filters which match against a field are based on this filter, so those will also accept lists. That means you can do `form_query.xmlns(XMLNS1)` or `form_query.xmlns([XMLNS1, XMLNS2, ...])`.

Contributing: Additions to this file should be added to the `builtin_filters` method on either ESQuery or HQESQuery, as appropriate (is it an HQ thing?).

`corehq.apps.es.filters.`**`AND`**(**filters*)
Filter docs to match all of the filters passed in

`corehq.apps.es.filters.`**`NOT`**(*filter_*)
Exclude docs matching the filter passed in

`corehq.apps.es.filters.`**`OR`**(**filters*)
Filter docs to match any of the filters passed in

`corehq.apps.es.filters.`**`date_range`**(*field*, *gt=None*, *gte=None*, *lt=None*, *lte=None*)
Range filter that accepts datetime objects as arguments

`corehq.apps.es.filters.`**`doc_id`**(*doc_id*)
Filter by doc_id. Also accepts a list of doc ids

`corehq.apps.es.filters.`**`doc_type`**(*doc_type*)
Filter by doc_type. Also accepts a list

`corehq.apps.es.filters.`**`domain`**(*domain_name*)
Filter by domain.

`corehq.apps.es.filters.`**`empty`**(*field*)
Only return docs with a missing or null value for `field`

`corehq.apps.es.filters.`**`exists`**(*field*)
Only return docs which have a value for `field`

`corehq.apps.es.filters.`**`missing`**(*field*, *exist=True*, *null=True*)
Only return docs missing a value for `field`

`corehq.apps.es.filters.`**`nested`**(*path*, *filter_*)
Query nested documents which normally can't be queried directly

`corehq.apps.es.filters.`**`non_null`**(*field*)
Only return docs with a real, non-null value for `field`

`corehq.apps.es.filters.`**`range_filter`**(*field*, *gt=None*, *gte=None*, *lt=None*, *lte=None*)
Filter `field` by a range. Pass in some sensible combination of `gt` (greater than), `gte` (greater than or equal to), `lt`, and `lte`.

`corehq.apps.es.filters.`**`term`**(*field*, *value*)
Filter docs by a field 'value' can be a singleton or a list.

18.3 Available Queries

Queries are used for actual searching - things like relevancy scores, Levenstein distance, and partial matches.

View the elasticsearch documentation to see what other options are available, and put 'em here if you end up using any of 'em.

`corehq.apps.es.queries.`**`filtered`**(*query*, *filter_*)
Filtered query for performing both filtering and querying at once

`corehq.apps.es.queries.`**`match_all`**()
No-op query used because a default must be specified

`corehq.apps.es.queries.`**`nested`**(*path*, *query*, **args*, ***kwargs*)
Creates a nested query for use with nested documents

Keyword arguments such as score_mode and others can be added.

corehq.apps.es.queries.**nested_filter**(*path*, *filter_*, **args*, ***kwargs*)
Creates a nested query for use with nested documents

Keyword arguments such as score_mode and others can be added.

corehq.apps.es.queries.**search_string_query**(*search_string*, *default_fields=None*)
Allows users to use advanced query syntax, but if `search_string` does not use the ES query string syntax, default to doing an infix search for each term. (This may later change to some kind of fuzzy matching).

This is also available via the main ESQuery class.

18.4 Aggregate Queries

Aggregations are a replacement for Facets

Here is an example used to calculate how many new pregnancy cases each user has opened in a certain date range.

```
res = (CaseES()
       .domain(self.domain)
       .case_type('pregnancy')
       .date_range('opened_on', gte=startdate, lte=enddate))
       .aggregation(TermsAggregation('by_user', 'opened_by')
       .size(0)

buckets = res.aggregations.by_user.buckets
buckets.user1.doc_count
```

There's a bit of magic happening here - you can access the raw json data from this aggregation via `res.aggregation('by_user')` if you'd prefer to skip it.

The `res` object has a `aggregations` property, which returns a namedtuple pointing to the wrapped aggregation results. The name provided at instantiation is used here (`by_user` in this example).

The wrapped `aggregation_result` object has a `result` property containing the aggregation data, as well as utilties for parsing that data into something more useful. For example, the `TermsAggregation` result also has a `counts_by_bucket` method that returns a `{bucket: count}` dictionary, which is normally what you want.

As of this writing, there's not much else developed, but it's pretty easy to add support for other aggregation types and more results processing

class corehq.apps.es.aggregations.**AggregationRange**
Note that a range includes the "start" value and excludes the "end" value. i.e. start <= X < end

Parameters

- **start** – range start
- **end** – range end
- **key** – optional key name for the range

class corehq.apps.es.aggregations.**AggregationTerm**(*name*, *field*)

field
Alias for field number 1

name
Alias for field number 0

class corehq.apps.es.aggregations.**DateHistogram**(*name*, *datefield*, *interval*, *timezone=None*)

Aggregate by date range. This can answer questions like "how many forms were created each day?".

This class can be instantiated by the `ESQuery.date_histogram` method.

Parameters

- **name** – what do you want to call this aggregation
- **datefield** – the document's date field to look at
- **interval** – the date interval to use: "year", "quarter", "month", "week", "day", "hour", "minute", "second"
- **timezone** – do bucketing using this time zone instead of UTC

__init__(*name*, *datefield*, *interval*, *timezone=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**ExtendedStatsAggregation**(*name*, *field*, *script=None*)

Extended stats aggregation that computes an extended stats aggregation by field

class corehq.apps.es.aggregations.**FilterAggregation**(*name*, *filter*)

Bucket aggregation that creates a single bucket for the specified filter

Parameters

- **name** – aggregation name
- **filter** – filter body

__init__(*name*, *filter*)

x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**FiltersAggregation**(*name*, *filters=None*)

Bucket aggregation that creates a bucket for each filter specified using the filter name.

Parameters **name** – aggregation name

__init__(*name*, *filters=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

add_filter(*name*, *filter*)

Parameters

- **name** – filter name
- **filter** – filter body

class corehq.apps.es.aggregations.**MinAggregation**(*name*, *field*)

Bucket aggregation that returns the minumum value of a field

Parameters

- **name** – aggregation name
- **field** – name of the field to min

class corehq.apps.es.aggregations.**MissingAggregation**(*name*, *field*)

A field data based single bucket aggregation, that creates a bucket of all documents in the current document set context that are missing a field value (effectively, missing a field or having the configured NULL value set).

Parameters

- **name** – aggregation name

- **field** – name of the field to bucket on

__init__(*name*, *field*)
x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**NestedAggregation**(*name*, *path*)
A special single bucket aggregation that enables aggregating nested documents.

Parameters **path** – Path to nested document

__init__(*name*, *path*)
x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**NestedTermAggregationsHelper**(*base_query*, *terms*, *inner_most_aggregation=None*)
Helper to run nested term-based queries (equivalent to SQL group-by clauses). This is not at all related to the ES 'nested aggregation'. The final aggregation defaults to a count of documents, though can also be used to sum a different field of the document.

Example usage:

```
# counting all forms submitted in a domain grouped by app id and user id

NestedTermAggregationsHelper(
    base_query=FormES().domain(domain_name),
    terms=[
        AggregationTerm('app_id', 'app_id'),
        AggregationTerm('user_id', 'form.meta.userID'),
    ]
).get_data()

# summing the balances of ledger values, grouped by the entry id

NestedTermAggregationsHelper(
    base_query=LedgerES().domain(domain).section(section_id),
    terms=[
        AggregationTerm('entry_id', 'entry_id'),
    ],
    inner_most_aggregation=SumAggregation('balance', 'balance'),
).get_data()
```

This works by bucketing docs first by one terms aggregation, then within that bucket, bucketing further by the next term, and so on. This is then flattened out to appear like a group-by-multiple.

__init__(*base_query*, *terms*, *inner_most_aggregation=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**RangeAggregation**(*name*, *field*, *ranges=None*, *keyed=True*)
Bucket aggregation that creates one bucket for each range :param name: the aggregation name :param field: the field to perform the range aggregations on :param ranges: list of AggregationRange objects :param keyed: set to True to have the results returned by key instead of as a list (see RangeResult.normalized_buckets)

__init__(*name*, *field*, *ranges=None*, *keyed=True*)
x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**StatsAggregation**(*name*, *field*, *script=None*)
Stats aggregation that computes a stats aggregation by field

Parameters

- **name** – aggregation name
- **field** – name of the field to collect stats on
- **script** – an optional field to allow you to script the computed field

__init__(*name*, *field*, *script=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**SumAggregation**(*name*, *field*)
Bucket aggregation that sums a field

Parameters

- **name** – aggregation name
- **field** – name of the field to sum

__init__(*name*, *field*)
x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**TermsAggregation**(*name*, *field*, *size=None*)
Bucket aggregation that aggregates by field

Parameters

- **name** – aggregation name
- **field** – name of the field to bucket on
- **size** –

__init__(*name*, *field*, *size=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

class corehq.apps.es.aggregations.**TopHitsAggregation**(*name*, *field=None*, *is_ascending=True*, *size=1*, *include=None*)
A top_hits metric aggregator keeps track of the most relevant document being aggregated This aggregator is intended to be used as a sub aggregator, so that the top matching documents can be aggregated per bucket.

Parameters

- **name** – Aggregation name
- **field** – This is the field to sort the top hits by. If None, defaults to sorting by score.
- **is_ascending** – Whether to sort the hits in ascending or descending order.
- **size** – The number of hits to include. Defaults to 1.
- **include** – An array of fields to include in the hit. Defaults to returning the whole document.

__init__(*name*, *field=None*, *is_ascending=True*, *size=1*, *include=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

18.5 AppES

class corehq.apps.es.apps.**AppES**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)

builtin_filters

index = u'apps'

corehq.apps.es.apps.**app_id**(*app_id*)

corehq.apps.es.apps.**build_comment**(*comment*)

corehq.apps.es.apps.**cloudcare_enabled**(*cloudcare_enabled*)

corehq.apps.es.apps.**created_from_template**(*from_template=True*)

corehq.apps.es.apps.**is_build**(*build=True*)

corehq.apps.es.apps.**is_released**(*released=True*)

corehq.apps.es.apps.**uses_case_sharing**(*case_sharing=True*)

corehq.apps.es.apps.**version**(*version*)

18.6 UserES

Here's an example adapted from the case list report - it gets a list of the ids of all unknown users, web users, and demo users on a domain.

```
from corehq.apps.es import users as user_es

user_filters = [
    user_es.unknown_users(),
    user_es.web_users(),
    user_es.demo_users(),
]

query = (user_es.UserES()
         .domain(self.domain)
         .OR(*user_filters)
         .show_inactive())

owner_ids = query.get_ids()
```

class corehq.apps.es.users.**UserES**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)

builtin_filters

default_filters = {u'active': {u'term': {u'is_active': True}}, u'not_deleted': {u'

index = u'users'

show_inactive()
Include inactive users, which would normally be filtered out.

show_only_inactive()

corehq.apps.es.users.**admin_users**()
Return only AdminUsers. Admin users are mock users created from xform submissions with unknown user ids whose username is "admin".

corehq.apps.es.users.**analytics_enabled**(*enabled=True*)

corehq.apps.es.users.**created**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.users.**demo_users**()
Matches users whose username is demo_user

corehq.apps.es.users.**domain**(*domain*)

corehq.apps.es.users.**is_active**(*active=True*)

corehq.apps.es.users.**is_practice_user**(*practice_mode=True*)

corehq.apps.es.users.**last_logged_in**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.users.**location**(*location_id*)

corehq.apps.es.users.**mobile_users**()

corehq.apps.es.users.**primary_location**(*location_id*)

corehq.apps.es.users.**role_id**(*role_id*)

corehq.apps.es.users.**unknown_users**()
Return only UnknownUsers. Unknown users are mock users created from xform submissions with unknown user ids.

corehq.apps.es.users.**user_ids**(*user_ids*)

corehq.apps.es.users.**username**(*username*)

corehq.apps.es.users.**web_users**()

18.7 CaseES

Here's an example getting pregnancy cases that are either still open or were closed after May 1st.

```
from corehq.apps.es import cases as case_es

q = (case_es.CaseES()
     .domain('testproject')
     .case_type('pregnancy')
     .OR(case_es.is_closed(False),
         case_es.closed_range(gte=datetime.date(2015, 05, 01))))
```

class corehq.apps.es.cases.**CaseES**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)

builtin_filters

index = u'cases'

corehq.apps.es.cases.**active_in_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)
Restricts cases returned to those with actions during the range

corehq.apps.es.cases.**case_ids**(*case_ids*)

corehq.apps.es.cases.**case_type**(*type_*)

corehq.apps.es.cases.**closed_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**is_closed**(*closed=True*)

corehq.apps.es.cases.**modified_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**open_case_aggregation**(*name=u'open_case'*, *gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**opened_by**(*user_id*)

corehq.apps.es.cases.**opened_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**owner**(*owner_id*)

corehq.apps.es.cases.**owner_type**(*owner_type*)

corehq.apps.es.cases.**server_modified_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**touched_total_aggregation**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**user**(*user_id*)

corehq.apps.es.cases.**user_ids_handle_unknown**(*user_ids*)

18.8 FormES

class corehq.apps.es.forms.**FormES**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)

builtin_filters

completed_histogram(*timezone=None*)

default_filters = {u'has_domain': {u'not': {u'missing': {u'field': u'domain'}}}, u

domain_aggregation()

index = u'forms'

only_archived()
Include only archived forms, which are normally excluded

submitted_histogram(*timezone=None*)

user_aggregation()

corehq.apps.es.forms.**app**(*app_ids*)

corehq.apps.es.forms.**completed**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.forms.**j2me_submissions**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.forms.**submitted**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.forms.**updating_cases**(*case_ids*)
return only those forms that have case blocks that touch the cases listed in *case_ids*

corehq.apps.es.forms.**user_id**(*user_ids*)

corehq.apps.es.forms.**user_ids_handle_unknown**(*user_ids*)

corehq.apps.es.forms.**user_type**(*user_types*)

corehq.apps.es.forms.**xmlns**(*xmlnss*)

18.9 DomainES

Here's an example generating a histogram of domain creations (that's a type of faceted query), filtered by a provided list of domains and a report date range.

```
from corehq.apps.es import DomainES

domains_after_date = (DomainES()
                      .in_domains(domains)
                      .created(gte=datespan.startdate, lte=datespan.enddate)
                      .date_histogram('date', 'date_created', interval)
                      .size(0))
histo_data = domains_after_date.run().aggregations.date.buckets_list
```

class corehq.apps.es.domains.**DomainES**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)

builtin_filters

default_filters = {u'not_snapshot': {u'not': {u'term': {u'is_snapshot': True}}}}

index = u'domains'

only_snapshots()

Normally snapshots are excluded, instead, return only snapshots

corehq.apps.es.domains.**commcare_domains**()

corehq.apps.es.domains.**commconnect_domains**()

corehq.apps.es.domains.**commtrack_domains**()

corehq.apps.es.domains.**created**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.domains.**created_by_user**(*creating_user*)

corehq.apps.es.domains.**in_domains**(*domains*)

corehq.apps.es.domains.**incomplete_domains**()

corehq.apps.es.domains.**is_active**(*is_active=True*)

corehq.apps.es.domains.**is_active_project**(*is_active=True*)

corehq.apps.es.domains.**last_modified**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.domains.**non_test_domains**()

corehq.apps.es.domains.**real_domains**()

corehq.apps.es.domains.**self_started**()

18.10 SMSES

class corehq.apps.es.sms.**SMSES**(*index=None*, *debug_host=None*, *es_instance_alias=u'default'*)

builtin_filters

index = u'sms'

user_aggregation()

corehq.apps.es.sms.**direction**(*direction_*)

corehq.apps.es.sms.**incoming_messages**()

corehq.apps.es.sms.**outgoing_messages**()

corehq.apps.es.sms.**processed**(*processed=True*)

corehq.apps.es.sms.**processed_or_incoming_messages**()

corehq.apps.es.sms.**received**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.sms.**to_commcare_case**()

corehq.apps.es.sms.**to_commcare_user**()

corehq.apps.es.sms.**to_commcare_user_or_case**()

corehq.apps.es.sms.**to_couch_user**()

corehq.apps.es.sms.**to_web_user**()

CHAPTER 19

Analyzing Test Coverage

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

19.1 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.

19.1.1 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*.

19.1.2 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 20

Advanced App Features

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

20.1 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>
```

HQ's app manager only allows users to configure one level of nesting; that is, it does not allow for "grandchild" modules. Although CommCare mobile supports multiple levels of nesting, beyond two levels it quickly gets prohibitively complex for the user to understand the implications of their app design and for for HQ to determine a logical set of session variables for every case. The modules could have all different case types, all the same, or a mix, and for modules that use the same case type, that case type may have a different meanings (e.g., a "person" case type that is sometimes a mother and sometimes a child), which all makes it difficult for HQ to determine the user's intended application design. See below for more on how session variables are generated with child modules.

20.1.1 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.

20.1.2 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.

20.2 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**.

20.2.1 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

20.2.2 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 or reporting 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.

20.2.3 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>
```

20.2.4 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>
```

(continues on next page)

(continued from previous page)

```
        <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 21

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

21.1 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)
```

(continues on next page)

(continued from previous page)

```
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
        )
```

21.2 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, expiry=60 * 60)
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 22

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.

22.1 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
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

22.2 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`

22.3 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
```

22.4 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'}
    )
```

22.4.1 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 23

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.

23.1 Messaging Definitions

23.1.1 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.

23.1.2 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

23.2 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.

23.2.1 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.

23.2.2 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.

23.2.3 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.

23.3 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.

23.4 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)

23.5 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

23.5.1 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 settings.HQ_APPS and settings.SMS_LOADED_SQL_BACKENDS
3. Run ./manage.py makemigrations sms; Django will just create a proxy model for the backend model, but no database changes will occur
4. 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.
- If this gateway has a phone number that people can reply to (whether a long code or short code), you'll want to add an entry to the sms.Phoneblacklist model for the gateway's phone number so that the system won't allow sending SMS to this number as a precaution. You can do so in the Django admin, and you'll want to make sure that send_sms and can_opt_in are both False on the record.

23.5.2 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.

23.5.3 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.

23.5.4 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. When choosing the originating phone number, the destination number is hashed and that hash is used to choose from the list of load balancing phone numbers, so that a recipient always receives messages from the same originating number.

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

23.6 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.

23.6.1 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

23.6.2 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.

23.6.3 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.

23.6.4 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.

23.7 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 24

Locations

24.1 Location Permissions

24.1.1 Normal Access

Location Types - Users who can edit apps on the domain can edit location types. Locations - There is an "edit_locations" and a "view_locations" permission.

24.1.2 Restricted Access and Whitelist

Many large projects have mid-level users who should have access to a subset of the project based on the organization's hierarchy.

This is handled by a special permission called "Full Organization Access" which is enabled by default on all user roles. To restrict data access based on a user's location, projects may create a user role with this permission disabled.

This is checked like so:

```
user.has_permission(domain, 'access_all_locations')
```

We have whitelisted portions of HQ that have been made to correctly handle these restricted users. Anything not explicitly whitelisted is inaccessible to restricted users.

24.1.3 Whitelist Implementation

There is `LocationAccessMiddleware` which controls this whitelist. It intercepts every request, checks if the user has restricted access to the domain, and if so, only allows requests to whitelisted views. This middleware also guarantees that restricted users have a location assigned. That is, if a user should be restricted, but does not have an assigned location, they can't see anything. This is to prevent users from obtaining full access in the event that their location is deleted or improperly assigned.

The other component of this is uitabs. The menu bar and the sidebar on HQ are composed of a bunch of links and names, essentially. We run the url for each of these links against the same check that the middleware uses to see if it should be visible to the user. In this way, they only see menu and sidebar links that are accessible.

To mark a view as location safe, you apply the `@location_safe` decorator to it. This can be applied directly to view functions, view classes, HQ report classes, or tastypie resources (see implentation and existing usages for examples).

UCR and Report Builder reports will be automatically marked as location safe if the report contains a location choice provider. This is done using the `conditionally_location_safe` decorator, which is provided with a function that in this case checks that the report has at least one location choice provider.

When marking a view as location safe, you must also check for restricted users by using either `request.can_access_all_locations` or `user.has_permission(domain, 'access_all_locations')` and limit the data returned accordingly.

You should create a user who is restricted and click through the desired workflow to make sure it still makes sense, there could be for instance, ajax requests that must also be protected, or links to features the user shouldn't see.

CHAPTER 25

Caching and Memoization

There are two primary ways of caching in CommCareHQ - using the decorators `@quickcache` and `@memoized`. At their core, these both do the same sort of thing - they store the results of function, like this simplified version:

```
cache = {}

def get_object(obj_id):
    if obj_id not in cache:
        obj = expensive_query_for_obj(obj_id)
        cache[obj_id] = obj
    return cache[obj_id]
```

In either case, it is important to remember that the body of the function being cached is not evaluated at all when the cache is hit. This results in two primary concerns - what to cache and how to identify it. You should cache only functions which are referentially transparent, that is, "pure" functions which return the same result when called multiple times with the same set of parameters.

This document describes the use of these two utilities.

25.1 Memoized

Memoized is an in-memory cache. At its simplest, it's a replacement for the two common patterns used in this example class:

```
class MyClass(object):

    def __init__(self):
        self._all_objects = None
        self._objects_by_key = {}

    @property
    def all_objects(self):
        if self._all_objects is None:
```

(continues on next page)

(continued from previous page)

```
            result = do_a_bunch_of_stuff()
            self._all_objects = result
        return self._all_objects

    def get_object_by_key(self, key):
        if key not in self._objects_by_key:
            result = do_a_bunch_of_stuff(key)
            self._objects_by_key[key] = result
        return self._objects_by_key[key]
```

With the memoized decorator, this becomes:

```
from memoized import memoized

class MyClass(object):

    @property
    @memoized
    def all_objects(self):
        return do_a_bunch_of_stuff()

    @memoized
    def get_object_by_key(self, key):
        return do_a_bunch_of_stuff(key)
```

When decorating a class method, `@memoized` stores the results of calls to those methods on the class instance. It stores a result for every unique set of arguments passed to the decorated function. This persists as long as the class does (or until you manually invalidate), and will be garbage collected along with the instance.

You can decorate any callable with `@memoized` and the cache will persist for the life of the callable. That is, if it isn't an instance method, the cache will probably be stored in memory for the life of the process. This should be used sparingly, as it can lead to memory leaks. However, this can be useful for lazily initializing singleton objects. Rather than computing at module load time:

```
def get_classes_by_doc_type():
    # Look up all subclasses of Document
    return result

classes_by_doc_type = get_classes_by_doc_type()
```

You can memoize it, and only compute if and when it's needed. Subsequent calls will hit the cache.

```
@memoized
def get_classes_by_doc_type():
    # Look up all subclasses of Document
    return result
```

25.2 Quickcache

`@quickcache` behaves much more like a normal cache. It stores results in a caching backend (Redis, in CCHQ) for a specified timeout (5 minutes, by default). This also means they can be shared across worker machines. Quickcache also caches objects in local memory (10 seconds, by default). This is faster to access than Redis, but its not shared across machines.

Quickcache requires you to specify which arguments to "vary on", that is, which arguments uniquely identify a cache

For examples of how it's used, check out the repo. For background, check out Why we made quickcache

25.3 The Differences

Memoized returns the same actual python object that was originally returned by the function. That is, `id(obj1) == id(obj2)` and `obj1 is obj2`. Quickcache, on the other hand, saves a copy (however, if you're within the `memoized_timeout`, you'll get the original object, but don't write code which depends on it.).

Memoized is a python-only library with no other dependencies; quickcache is configured on a per-project basis to use whatever cache backend is being used, in our case django-cache backends.

Incidentally, quickcache also uses some inspection magic that makes it not work in a REPL context (i.e. from running *python* interactively or *./manage.py shell*)

25.4 Lifecycle

Memoized on instance method: The cache lives on the instance itself, so it gets garbage collected along with the instance

Memoized on any other function/callable: The cache lives on the callable, so if it's globally scoped and never gets garbage collected, neither does the cache

Quickcache: Garbage collection happens based on the timeouts specified: memoize_timeout for the local cache and timeout for redis

25.5 Scope

In-memory caching (memoized or quickcache) is scoped to a single process on a single machine. Different machines or different processes on the same machine do not share these caches between them.

For this reason, memoized is usually used when you want to cache things only for duration of a request, or for globally scoped objects that need to be always available for very fast retrieval from memory.

Redis caching (quickcache only) is globally shared between processes on all machines in an environment. This is used to share a cache across multiple requests and webworkers (although quickcache also provides a short-duration, lightning quick, in-memory cache like @memoized, so you should never need to use both).

25.6 Decorating various things

Memoized is more flexible here - it can be used to decorate any callable, including a class. In practice, it's much more common and practical to limit ourselves to normal functions, class methods, and instance methods. Technically, if you do use it on a class, it has the effect of caching the result of calling the class to create an instance, so instead of creating a new instance, if you call the class twice with the same arguments, you'll get the same (*obj1 is obj2*) python object back.

Quickcache must go on a function—whether standalone or within a class—and does not work on other callables like a class or other custom callable. In practice this is not much of a limitation.

25.7 Identifying cached values

Cached functions usually have a set of parameters passed in, and will return different results for different sets of parameters.

Best practice here is to use as small a set of parameters as possible, and to use simple objects as parameters when possible (strings, booleans, integers, that sort of thing).

```
@quickcache(['domain_obj.name', 'user._id'], timeout=10)
def count_users_forms_by_device(domain_obj, user):
    return {
        FormAccessors(domain_obj.name).count_forms_by_device(device.device_id)
        for device in user.devices
    }
```

The first argument to `@quickcache` is an argument called `vary_on` which is a list of the parameters used to identify each result stored in the cache. Taken together, the variables specified in vary_on should constitute all inputs that would change the value of the output. You may be thinking "Well, isn't that just all of the arguments?" Often, yes. However, also very frequently, a function depends not on the exact object being passed in, but merely on one or a few properties of that object. In the example above, we want the function to return the same result when called with the same domain name and user ID, not necessarily the same exact objects. Quickcache handles this by allowing you to pass in strings like `parameter.attribute`. Additionally, instead of a list of parameters, you may pass in a function, which will be called with the arguments of the cached function to return a cache key.

Memoized does not provide these capabilities, and instead always uses all of the arguments passed in. For this reason, you should only memoize functions with simple arguments. At a minimum, all arguments to memoized must be hashable. You'll notice that the above function doesn't actually use anything on the `domain_obj` other than name, so you could just refactor it to accept `domain` instead (this also means code calling this function won't need to fetch the domain object to pass to this function, only to discard everything except the name anyways).

You don't need to let this consideration muck up your function's interface. A common practice is to make a helper function with simple arguments, and decorate that. You can then still use the top-level function as you see fit. For example, let's pretend the above function is an instance method and you want to use memoize rather than quickcache. You could split it apart like this:

```
@memoized
def _count_users_forms_by_device(self, domain, device_id):
    return FormAccessors(domain).count_forms_by_device(device_id)

def count_users_forms_by_device(self, domain_obj, user):
    return {
        self._count_users_forms_by_device(domain_obj.name, device.device_id)
        for device in user.devices
    }
```

25.8 What can be cached

Memoized: All arguments must be hashable; notably, lists and dicts are not hashable, but tuples are.

Return values can be anything.

Quickcache: All vary_on values must be "basic" types (all the way down, if they are collections): string types, bool, number, list/tuple (treated as interchangeable), dict, set, None. Arbitrary objects are not allowed, nor are lists/tuples/dicts/sets containing objects, etc.

Return values can be anything that's pickleable. More generally, quickcache dictates what values you can vary_on, but leaves what values you can return up to your caching backend; since we use django cache, which uses pickle, our return values have to be pickleable.

25.9 Invalidation

"There are only two hard problems in computer science - cache invalidation and naming things" (and off-by-one errors)

Memoized doesn't allow invalidation except by blowing away the whole cache for all parameters. Use `<function>.reset_cache()`

One of quickcache's killer features is the ability to invalidate the cache for a specific function call. To invalidate the cache for `<function>(*args, **kwargs)`, use `<function>.clear(*args, **kwargs)`. Appropriately selecting your args makes this easier.

To sneakily prime the cache of a particular call with a preset value, you can use `<function>.set_cached_value(*args, **kwargs).to(value)`. This is useful when you are already holding the answer to an expensive computation in your hands and want to do the next caller the favor of not making them do it. It's also useful for when you're dealing with a backend that has delayed refresh as is the case with Elasticsearch (when configured a certain way).

25.10 Other ways of caching

Redis is sometimes accessed manually or through other wrappers for special purposes like locking. Some of those are:

RedisLockableMixIn Provides `get_locked_obj`, useful for making sure only one instance of an object is accessible at a time.

CriticalSection Similar to the above, but used in a `with` construct - makes sure a block of code is never run in parallel with the same identifier.

QuickCachedDocumentMixin Intended for couch models - quickcaches the `get` method and provides automatic invalidation on save or delete.

CachedCouchDocumentMixin Subclass of QuickCachedDocumentMixin which also caches some couch views

CHAPTER 26

Device Restore Optimization

This document is based on the definitions and requirements for restore logic outlined in new-idea-for-extension-cases.md.

Important terms from that document that are also used in this document:

A case is **available** if

- it is **open** and not an **extension** case
- it is **open** and is the **extension** of an **available** case.

A case is **live** if any of the following are true:

- it is **owned** and **available**
- it has a **live child**
- it has a **live extension**
- it is **open** and is the **exension** of a **live** case

26.1 Dealing with shards

Observation: the decision to shard by case ID means that the number of levels in a case hierarchy impacts restore performance. The minimum number of queries needed to retrieve all *live* cases for a device can equal the number of levels in the hierarchy. The maximum is unbounded.

Since cases are sharded by case ID. . .

- Quotes from How Sharding Works
  - Non-partitioned queries do not scale with respect to the size of cluster, thus they are discouraged.
  - Queries spanning multiple partitions . . . tend to be inefficient, so such queries should be done sparingly.
  - A particular cross-partition query may be required frequently and efficiently. In this case, data needs to be stored in multiple partitions to support efficient reads.

- Potential optimizations to allow PostgreSQL to do more of the heavy lifting for us.
  - Shard `case_index` by domain.
    - Probably not? Some domains are too large.
  - Copy related case index data into all relevant shards to allow a query to run on a single shard.
    - Nope. Effectively worse than sharding by domain: would copy entire case index to every shard because in a given set of *live* cases that is the same size as or larger than the number of shards, each case will probably live in a different shard.
  - Re-shard based on ownership rather than case ID
    - Maybe use hierarchical keys since ownership is strictly hierarchical. This may simplify the sharding function.
    - Copy or move data between shards when ownership changes.

26.2 Data Structure

Simplified/minimal table definitions used in sample queries below.

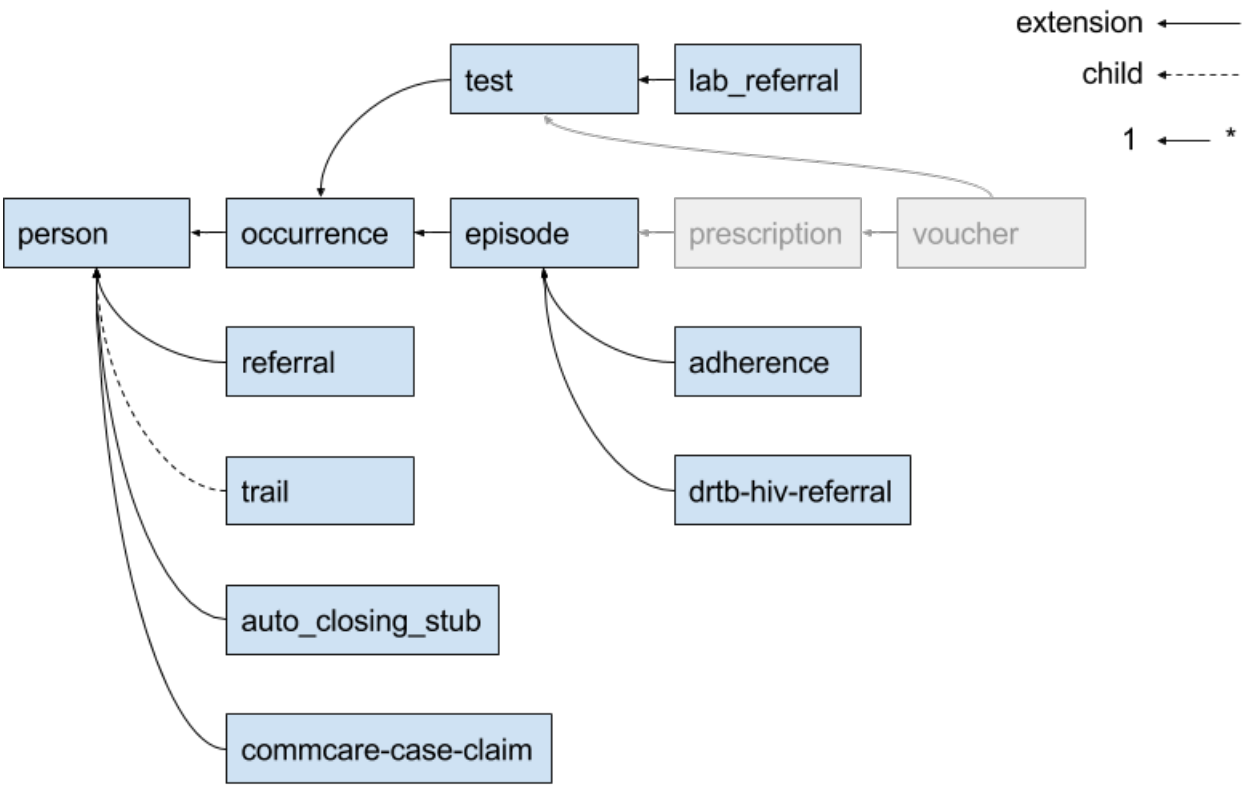
```
cases
  domain        char
  case_id       char
  owner_id      char
  is_open       bool

case_index
  domain        char
  parent_id     char
  child_id      char
  child_type    enum (CHILD|EXTENSION)
```

Presence of a row in the `case_index` adjacency list table implies that the referenced cases are *available*. The `case_index` is updated when new data is received during a device sync: new case relationships are inserted and relationships for closed cases are deleted. All information in the `case_index` table is also present in the `CommCareCaseIndexSQL` and `CommCareCaseSQL` tables. Likewise for the `cases` table, which is a subset of `CommCareCaseSQL`.

26.3 Case Study: UATBC case structure

Sources: eNikshay App Design and Feedback - Case Structure and case_utils.py. These sources contain conflicting information. For example:

- case_utils.py references *prescription* and *voucher* while the sheet does not.
- case_utils.py has *referral* related to *episode*, not *person* as in the sheet.



With the current sharding (by case ID) configuration, the maximum number of queries needed to get all *live* cases for a device is 5 because there are 5 levels in the case hierarchy. Update: this is wrong; it could be more than 5. Example: if a case retrieved in the 5th query has unvisited children, then at least one more query is necessary. Because any given case may have multiple parents, the maximum number of queries is unbounded.

26.4 Algorithm to minimize queries while sharding on case ID

The algorithm (Python):

```
next_ids = get_cases_owned_by_device(owner_ids)
live_ids = set(next_ids)
while next_ids:
    related_ids = set(get_related_cases(next_ids))
    if not related_ids:
        break
    next_ids = related_ids - live_ids
    live_ids.update(related_ids)
```

All queries below are simplified for the purposes of demonstration. They use the simplified table definitions from the *Data Structure* section in this document, and they only return case IDs. If this algorithm is implemented it will likely make sense to expand the queries to retrieve all case data, including case relationship data, and to query directly from `CommCareCaseIndexSQL` and `CommCareCaseSQL`.

The term "child" is a general term used to refer to a case that is related to another case by retaining a reference to the other case in its set of parent indices. It does not refer to the more restrictive "child" relationship type.

Definitions:

- `OWNER_DOMAIN` - the domain for which the query is being executed.

- `OWNER_IDS` - a set of user and group IDs for the device being restored.
- `NEXT_IDS` - a set of *live* case IDs.

`get_cases_owned_by_device()` retrieves all open cases that are not extension cases given a set of owner IDs for a device. That is, it retrieves all *live* cases that are directly owned by a device (user and groups). The result of this function can be retrieved with a single query:

```
select cx.case_id
from cases cx
  left outer join case_index ci
    on ci.domain = cx.domain and ci.child_id = cx.case_id
where
  cx.domain = OWNER_DOMAIN and
  cx.owner_id in OWNER_IDS and
  (ci.child_id is null or ci.child_type != EXTENSION) and
  cx.is_open = true
```

`get_related_cases()` retrieves all *live* cases related to the given set of *live* case IDs. The result of this function can be retrieved with a single query:

```
-- parent cases (outgoing)
select parent_id, child_id, child_type
from case_index
where domain = OWNER_DOMAIN
  and child_id in NEXT_IDS
union
-- child cases (incoming)
select parent_id, child_id, child_type
from case_index
where domain = OWNER_DOMAIN
  and parent_id in NEXT_IDS
  and child_type = EXTENSION
```

The `IN` operator used to filter on case ID sets should be optimized since case ID sets may be large.

Each of the above queries is executed on all shards and the results from each shard are merged into the final result set.

26.5 One query to rule them all.

Objective: retrieve all *live* cases for a device with a single query. This query answers the question Which cases end up on a user's phone? The sharding structure will need to be changed if we want to use something like this.

```
with owned_case_ids as (
  select case_id
  from cases
  where
    domain = OWNER_DOMAIN and
    owner_id in OWNER_IDS and
    is_open = true
), recursive parent_tree as (
  -- parent cases (outgoing)
  select parent_id, child_id, child_type, array[child_id] as path
  from case_index
  where domain = OWNER_DOMAIN
    and child_id in owned_case_ids
```

(continues on next page)

(continued from previous page)

```
  union
  -- parents of parents (recursive)
  select ci.parent_id, ci.child_id, ci.child_type, path || ci.child_id
  from case_index ci
    inner join parent_tree as refs on ci.child_id = refs.parent_id
  where ci.domain = OWNER_DOMAIN
    and not (ci.child_id = any(refs.path)) -- stop infinite recursion
), recursive child_tree as (
  -- child cases (incoming)
  select parent_id, child_id, child_type, array[parent_id] as path
  from case_index
  where domain = OWNER_DOMAIN
    and (parent_id in owned_case_ids or parent_id in parent_tree)
    and child_type = EXTENSION
  union
  -- children of children (recursive)
  select
    ci.parent_id,
    ci.child_id,
    ci.child_type,
    path || ci.parent_id
  from case_index ci
    inner join child_tree as refs on ci.parent_id = refs.child_id
  where ci.domain = OWNER_DOMAIN
    and not (ci.parent_id = any(refs.path)) -- stop infinite recursion
    and child_type = EXTENSION
)
select
  case_id as parent_id,
  null as child_id,
  null as child_type,
  null as path
from owned_case_ids
union
select * from parent_tree
union
select * from child_tree
```

26.6 Q & A

- Do we have documentation on existing restore logic?
  - Yes: new-idea-for-extension-cases.md
  - See also child/extension test cases
- new-idea-for-extension-cases.md: "[an extension case has] the ability (like a child case) to go out in the world and live its own life."

  What does it mean for an extension case to "live its own life"? Is it meaningful to have an extension case apart from the parent of which it is an extension? How are the attributes of an extension case "living its own life" different from one that is not living it's own life (I'm assuming *not living its own life* means it has the same lifecycle as its parent).
  - Danny Roberts:

    > haha i mean that may have been a pretty loosely picked phrase

I think I specifically just meant you can assign it an owner separate from its parent's

- Is there an ERD or something similar for UATBC cases and their relationships?
  - Case structure diagram (outdated)
  - SDD _EY Comments_v5_eq.docx (page 24, outdated)
  - eNikshay App Design and Feedback - Case Structure - Kriti
  - case_utils.py - Farid

CHAPTER 27

Playing nice with Cloudant/CouchDB

We have a lot of views:

```
$ find . -path *_design*/map.js | wc -l
     159
```

Things to know about views:

1. Every time you create or update a doc, each map function is run on it and the btree for the view is updated based on the change in what the maps emit for that doc. Deleting a doc causes the btree to be updated as well.
2. Every time you update a view, all views in the design doc need to be run, from scratch, in their entirety, on every single doc in the database, regardless of doc_type.

Things to know about our Cloudant cluster:

1. It's slow. You have to wait in line just to say "hi". Want to fetch a single doc? So does everyone else. Get in line, I'll be with you in just 1000ms.
2. That's pretty much it.

Takeaways:

1. Don't save docs! If nothing changed in the doc, just don't save it. Couchdb isn't smart enough to realize that nothing changed, so saving it incurs most of the overhead of saving a doc that actually changed.
2. Don't make http requests! If you need a bunch of docs by id, get them all in one request or a few large requests using `dimagi.utils.couch.database.iter_docs`.
3. Don't make http requests! If you want to save a bunch of docs, save them all at once (after excluding the ones that haven't changed and don't need to be saved!) using `MyClass.get_db().bulk_save(docs)`. If you're writing application code that touches a number of related docs in a number of different places, you want to bulk save them, and you understand the warning in its docstring, you can use `dimagi.utils.couch.bulk.CouchTransaction`. Note that this isn't good for saving thousands of documents, because it doesn't do any chunking.
4. Don't save too many docs in too short a time! To give the views time to catch up, rate-limit your saves if going through hundreds of thousands of docs. One way to do this is to save N docs and then make a tiny request to the view you think will be slowest to update, and then repeat.

5. Use different databases! All forms and cases save to the main database, but there is a *_meta* database we have just added for new doc or migrated doc types. When you use a different database you create two advantages: a) Documents you save don't contribute to the view indexing load of all of the views in the main database. b) Views you add don't have to run on all forms and cases.
6. Split views! When a single view changes, the **entire design doc** has to reindex. If you make a new view, it's much better to make a new design doc for it than to put it in with some other big, possibly expensive views. We use the *couchapps* folder/app for this.

CHAPTER 28

Celery

Official Celery documentation: http://docs.celeryproject.org/en/latest/ What is it ==========

Celery is a library we use to perform tasks outside the bounds of an HTTP request.

28.1 How to use celery

All celery tasks should go into a tasks.py file or tasks module in a django app. This ensures that `autodiscover_tasks` can find the task and register it with the celery workers.

These tasks should be decorated with one of the following:

1. `@task` defines a task that is called manually (with `task_function_name.delay` in code)
2. `@periodic_task` defines a task that is called at some interval (specified by `crontab` in the decorator)
3. `@serial_task` defines a task that should only ever have one job running at one time

28.2 Best practices

Do not pass objects to celery. Instead, IDs can be passed and the celery task can retrieve the object from the database using the ID. This keeps message lengths short and reduces burden on RabbitMQ as well as preventing tasks from operating on stale data.

Do not specify `serializer='pickle'` for new tasks. This is a deprecated message serializer and by default, we now use JSON.

28.3 Queues

Table 1: Queues

Queue	I/O Bound?	Target max time-to-start	Target max time-to-start comments	Description of usage	How long does the typical task take to complete?	Best practices / Notes
send_report_throttled		hours	30 minutes: reports should be sent as close to schedule as possible. EDIT: this queue only affects mvp-* and ews-ghana	This is used specifically for domains who are abusing Scheduled Reports and overwhelming the background queue. See settings.THROTTLE_SCHED_REPORTS_PATTERNS		
submission_reprocessing_queue	no?	hours	1 hour: not critical if this gets behind as long as it can keep up within a few hours	Reprocess form submissions that errored in ways that can be handled by HQ. Triggered by 'submission_reprocessing_queue' process.	seconds	
sumologic_logs_queue	yes	hours	1 hour: OK for this to get behind	Forward device logs to sumologic. Triggered by device log submission from mobile.	seconds	Non-essential queue
analytics_queue	yes	minutes		Used to run tasks related to external analytics tools like HubSpot. Triggered by user actions on the site.	instantaneous (seconds)	
reminder_case_update_queue	yes	minutes		Run reminder tasks related to case changes. Triggered by case change signal.	seconds	
reminder_queue	yes	minutes	15 minutes: since these are scheduled it can be	Runs the reminder rule tasks for reminders that	seconds	

28.4 Soil

Soil is a Dimagi utility to provide downloads that are backed by celery.

To use soil:

```
from soil import DownloadBase
from soil.progress import update_task_state
from soil.util import expose_cached_download

@task
def my_cool_task():
    DownloadBase.set_progress(my_cool_task, 0, 100)

    # do some stuff

    DownloadBase.set_progress(my_cool_task, 50, 100)

    # do some more stuff

    DownloadBase.set_progress(my_cool_task, 100, 100)

    expose_cached_download(payload, expiry, file_extension)
```

For error handling update the task state to failure and provide errors, HQ currently supports two options:

28.4.1 Option 1

This option raises a celery exception which tells celery to ignore future state updates. The resulting task result will not be marked as "successful" so `task.successful()` will return `False` If calling with `CELERY_ALWAYS_EAGER = True` (i.e. a dev environment), and with `.delay()`, the exception will be caught by celery and `task.result` will return the exception.

```
from celery.exceptions import Ignore
from soil import DownloadBase
from soil.progress import update_task_state
from soil.util import expose_cached_download

@task
def my_cool_task():
    try:
        # do some stuff
    except SomeError as err:
        errors = [err]
        update_task_state(my_cool_task, states.FAILURE, {'errors': errors})
        raise Ignore()
```

28.4.2 Option 2

This option raises an exception which celery does not catch. Soil will catch this and set the error to the error message in the exception. The resulting task will be marked as a failure meaning `task.failed()` will return `True` If calling with `CELERY_ALWAYS_EAGER = True` (i.e. a dev environment), the exception will "bubble up" to the calling code.

```
from soil import DownloadBase
from soil.progress import update_task_state
from soil.util import expose_cached_download

@task
def my_cool_task():
    # do some stuff
    raise SomeError("my uncool error")
```

28.5 Testing

As noted in the [celery docs](http://docs.celeryproject.org/en/v4.2.1/userguide/testing.html) testing tasks in celery is not the same as in production. In order to test effectively, mocking is required.

An example of mocking with Option 1 from the soil documentation:

```
@patch('my_cool_test.update_state')
def my_cool_test(update_state):
    res = my_cool_task.delay()
    self.assertIsInstance(res.result, Ignore)
    update_state.assert_called_with(
        state=states.FAILURE,
        meta={'errors': ['my uncool errors']}
    )
```

28.6 Other references

https://docs.google.com/presentation/d/1iiiVZDiOGXoLeTvEIgM_rGgw6Me5_wM_Cyc64bl7zns/edit#slide=id.g1d621cb6fc_0_372

https://docs.google.com/spreadsheets/d/10uv0YBVTGi88d6mz6xzwXRLY5OZLW1FJ0iarHI6Orck/edit?ouid=112475836275787837666&usp=sheets_home&ths=true

CHAPTER 29

User Configurable Reporting

An overview of the design, API and data structures used here.

29.1 Data Flow

Reporting is handled in multiple stages. Here is the basic workflow.

Raw data (form or case) → [Data source config] → Row in database table → [Report config] → Report in HQ

Both the data source config and report config are JSON documents that live in the database. The data source config determines how raw data (forms and cases) gets mapped to rows in an intermediary table, while the report config(s) determine how that report table gets turned into an interactive report in the UI.

29.2 Data Sources

Each data source configuration maps a filtered set of the raw data to indicators. A data source configuration consists of two primary sections:

1. A filter that determines whether the data is relevant for the data source
2. A list of indicators in that data source

In addition to these properties there are a number of relatively self-explanatory fields on a data source such as the `table_id` and `display_name`, and a few more nuanced ones. The full list of available fields is summarized in the following table:

Field	Description
filter	Determines whether the data is relevant for the data source
indicators	List of indicators to save
table_id	A unique ID for the table
display_name	A display name for the table that shows up in UIs
base_item_expression	Used for making tables off of repeat or list data
named_expressions	A list of named expressions that can be referenced in other filters and indicators
named_filters	A list of named filters that can be referenced in other filters and indicators

29.2.1 Data Source Filtering

When setting up a data source configuration, filtering defines what data applies to a given set of indicators. Some example uses of filtering on a data source include:

- Restricting the data by document type (e.g. cases or forms). This is a built-in filter.
- Limiting the data to a particular case or form type
- Excluding demo user data
- Excluding closed cases
- Only showing data that meets a domain-specific condition (e.g. pregnancy cases opened for women over 30 years of age)

Filter type overview

There are currently four supported filter types. However, these can be used together to produce arbitrarily complicated expressions.

Filter Type	Description
boolean_expression	A expression / logic statement (more below)
and	An "and" of other filters - true if all are true
or	An "or" of other filters - true if any are true
not	A "not" or inverse expression on a filter

To understand the `boolean_expression` type, we must first explain expressions.

Expressions

An *expression* is a way of representing a set of operations that should return a single value. Expressions can basically be thought of as functions that take in a document and return a value:

Expression: `function(document) → value`

In normal math/python notation, the following are all valid expressions on a `doc` (which is presumed to be a `dict` object:

- `"hello"`
- `7`
- `doc["name"]`
- `doc["child"]["age"]`
- `doc["age"] < 21`
- `"legal" if doc["age"] > 21 else "underage"`

In user configurable reports the following expression types are currently supported (note that this can and likely will be extended in the future):

Expression Type	Description	Example
identity	Just returns whatever is passed in	`doc`
constant	A constant	`"hello"` `, ``4, 2014-12-20`
property_name	A reference to the property in a document	`doc["nam e"]`
property_path	A nested reference to a property in a document	`doc["chi ld"]["age" ]`
conditional	An if/else expression	`"legal" if doc["ag e"] > 21 e lse "under age"`
switch	A switch statement	‘‘if doc[” age”] == 2 1: “legal” ‘‘ ‘‘elif doc [“age”] == 60: . . . ‘‘ `else: .. .`
array_index	An index into an array	`doc[1]`
split_string	Splitting a string and grabbing a specific element from it by index	‘‘**doc[“foo** bar”].spl it(‘ ‘)[0] ‘‘
iterator	Combine multiple expressions into a list	`[doc.nam e, doc.age , doc.gend er]`
related_doc	A way to reference something in another document	‘‘form.cas e.owner_id ‘‘
root_doc	A way to reference the root document explicitly (only needed when making a data source from repeat/child data)	‘‘repeat.p arent.name ‘‘
ancestor_location	A way to retrieve the ancestor of a particular type from a location	
nested	A way to chain any two expressions together	`f1(f2(do c))`
dict	A way to emit a dictionary of key/value pairs	‘‘**{“name”:** “test”, “ value”: f(doc)}‘‘
add_days	A way to add days to a date	‘‘my_date + timedelt a(days=15) ‘‘
add_months	A way to add months to a date	`my_date + relative delta(mont hs=15)`
month_start_date	First day in the month of a date	`2015-01- 20 -> 2015-01-01`
month_end_date	Last day in the month of a date	`2015-01- 20 -> 2015-01-31`
diff_days	A way to get duration in days between two dates	‘‘**(to_date** • from-da te).days‘‘
evaluator	A way to do arithmetic operations	`a + b*c / d`
base_iteration_number	Used with ‘‘*base_item_expression* ‘ <#saving-multiple-ro ws-per-caseform>‘__ - a way to get the current iteration number (starting from 0).	`loop.ind ex`

Following expressions act on a list of objects or a list of lists (for e.g. on a repeat list) and return another list or value.

These expressions can be combined to do complex aggregations on list data.

Expression Type	Description	Example
filter_items	Filter a list of items to make a new list	‘‘[1, 2, 3 , -1, -2, -3] -> [1, 2, 3]‘‘ (filter numbers greater than zero)
map_items	Map one list to another list	‘‘[{‘name’ : ‘a’, gen der: ‘f’}, {‘name’: ‘b, gender : ‘m’}]‘‘ -> `['a', 'b']` (list of names from list of child data)
sort_items	Sort a list based on an expression	`[{'name' : 'a', age : 5}, {'na me': 'b, a ge: 3}]` -> ‘‘[{‘name’ : ‘b, age: 3}, {‘nam e’: ‘a’, a ge: 5}]‘‘ (sort child data by age)
reduce_items	Aggregate a list of items into one value	sum on `[1, 2, 3 ]` -> `6`
flatten	Flatten multiple lists of items into one list	‘‘**[[1, 2],** [4, 5]]‘‘ -> `[1, 2, 4 , 5]`

JSON snippets for expressions

Here are JSON snippets for the various expression types. Hopefully they are self-explanatory.

Constant Expression

class corehq.apps.userreports.expressions.specs.**ConstantGetterSpec**

There are two formats for constant expressions. The simplified format is simply the constant itself. For example `"hello"`, or `5`.

The complete format is as follows. This expression returns the constant `"hello"`:

```
{
    "type": "constant",
    "constant": "hello"
}
```

Property Name Expression

class corehq.apps.userreports.expressions.specs.**PropertyNameGetterSpec**

This expression returns `doc["age"]`:

```
{
    "type": "property_name",
    "property_name": "age"
}
```

An optional `"datatype"` attribute may be specified, which will attempt to cast the property to the given data type. The options are "date", "datetime", "string", "integer", and "decimal". If no datatype is specified, "string" will be used.

Property Path Expression

class corehq.apps.userreports.expressions.specs.**PropertyPathGetterSpec**

This expression returns `doc["child"]["age"]`:

```
{
    "type": "property_path",
    "property_path": ["child", "age"]
}
```

An optional `"datatype"` attribute may be specified, which will attempt to cast the property to the given data type. The options are "date", "datetime", "string", "integer", and "decimal". If no datatype is specified, "string" will be used.

Conditional Expression

class corehq.apps.userreports.expressions.specs.**ConditionalExpressionSpec**

This expression returns `"legal" if doc["age"] > 21 else "underage"`:

Note that this expression contains other expressions inside it! This is why expressions are powerful. (It also contains a filter, but we haven't covered those yet - if you find the `"test"` section confusing, keep reading...)

Note also that it's important to make sure that you are comparing values of the same type. In this example, the expression that retrieves the age property from the document also casts the value to an integer. If this datatype is not specified, the expression will compare a string to the `21` value, which will not produce the expected results!

Switch Expression

class corehq.apps.userreports.expressions.specs.**SwitchExpressionSpec**

This expression returns the value of the expression for the case that matches the switch on expression. Note that case values may only be strings at this time.

```
{
    "type": "switch",
    "switch_on": {
        "type": "property_name",
        "property_name": "district"
    },
    "cases": {
        "north": {
            "type": "constant",
            "constant": 4000
        },
        "south": {
            "type": "constant",
            "constant": 2500
        },
        "east": {
            "type": "constant",
```

(continues on next page)

(continued from previous page)

```
            "constant": 3300
        },
        "west": {
            "type": "constant",
            "constant": 65
        },
    },
    "default": {
        "type": "constant",
        "constant": 0
    }
}
```

Coalesce Expression

class corehq.apps.userreports.expressions.specs.**CoalesceExpressionSpec**

This expression returns the value of the expression provided, or the value of the default_expression if the expression provided evalutes to a null or blank string.

```
{
    "type": "coalesce",
    "expression": {
        "type": "property_name",
        "property_name": "district"
    },
    "default_expression": {
        "type": "constant",
        "constant": "default_district"
    }
}
```

Array Index Expression

class corehq.apps.userreports.expressions.specs.**ArrayIndexExpressionSpec**

This expression returns `doc["siblings"][0]`:

```
{
    "type": "array_index",
    "array_expression": {
        "type": "property_name",
        "property_name": "siblings"
    },
    "index_expression": {
        "type": "constant",
        "constant": 0
    }
}
```

It will return nothing if the siblings property is not a list, the index isn't a number, or the indexed item doesn't exist.

Split String Expression

class corehq.apps.userreports.expressions.specs.**SplitStringExpressionSpec**

This expression returns `(doc["foo bar"]).split(' ')[0]`:

```
{
    "type": "split_string",
    "string_expression": {
        "type": "property_name",
        "property_name": "multiple_value_string"
    },
    "index_expression": {
        "type": "constant",
        "constant": 0
    },
    "delimiter": ","
}
```

The delimiter is optional and is defaulted to a space. It will return nothing if the string_expression is not a string, or if the index isn't a number or the indexed item doesn't exist. The index_expression is also optional. Without it, the expression will return the list of elements.

Iterator Expression

class corehq.apps.userreports.expressions.specs.**IteratorExpressionSpec**

```
{
    "type": "iterator",
    "expressions": [
        {
            "type": "property_name",
            "property_name": "p1"
        },
        {
            "type": "property_name",
            "property_name": "p2"
        },
        {
            "type": "property_name",
            "property_name": "p3"
        },
    ],
    "test": {}
}
```

This will emit `[doc.p1, doc.p2, doc.p3]`. You can add a `test` attribute to filter rows from what is emitted - if you don't specify this then the iterator will include one row per expression it contains regardless of what is passed in. This can be used/combined with the `base_item_expression` to emit multiple rows per document.

Base iteration number expressions

class corehq.apps.userreports.expressions.specs.**IterationNumberExpressionSpec**

These are very simple expressions with no config. They return the index of the repeat item starting from 0 when

used with a `base_item_expression`.

```
{
    "type": "base_iteration_number"
}
```

Related document expressions

class corehq.apps.userreports.expressions.specs.**RelatedDocExpressionSpec**

This can be used to lookup a property in another document. Here's an example that lets you look up `form.case.owner_id` from a form.

```
{
    "type": "related_doc",
    "related_doc_type": "CommCareCase",
    "doc_id_expression": {
        "type": "property_path",
        "property_path": ["form", "case", "@case_id"]
    },
    "value_expression": {
        "type": "property_name",
        "property_name": "owner_id"
    }
}
```

Ancestor location expression

class corehq.apps.locations.ucr_expressions.**AncestorLocationExpression**

This is used to return a json object representing the ancestor of the given type of the given location. For instance, if we had locations configured with a hierarchy like `country -> state -> county -> city`, we could pass the location id of Cambridge and a location type of state to this expression to get the Massachusetts location.

```
{
    "type": "ancestor_location",
    "location_id": {
        "type": "property_name",
        "name": "owner_id"
    },
    "location_type": {
        "type": "constant",
        "constant": "state"
    }
}
```

If no such location exists, returns null.

Optionally you can specifiy `location_property` to return a single property of the location.

```
{
    "type": "ancestor_location",
    "location_id": {
        "type": "property_name",
        "name": "owner_id"
```

(continues on next page)

(continued from previous page)

```
    },
    "location_type": {
        "type": "constant",
        "constant": "state"
    },
    "location_property": "site_code"
}
```

Nested expressions

class corehq.apps.userreports.expressions.specs.**NestedExpressionSpec**

These can be used to nest expressions. This can be used, e.g. to pull a specific property out of an item in a list of objects.

The following nested expression is the equivalent of a `property_path` expression to `["outer", "inner"]` and demonstrates the functionality. More examples can be found in the practical examples.

```
{
    "type": "nested",
    "argument_expression": {
        "type": "property_name",
        "property_name": "outer"
    },
    "value_expression": {
        "type": "property_name",
        "property_name": "inner"
    }
}
```

Dict expressions

class corehq.apps.userreports.expressions.specs.**DictExpressionSpec**

These can be used to create dictionaries of key/value pairs. This is only useful as an intermediate structure in another expression since the result of the expression is a dictionary that cannot be saved to the database.

See the practical examples for a way this can be used in a `base_item_expression` to emit multiple rows for a single form/case based on different properties.

Here is a simple example that demonstrates the structure. The keys of `properties` must be text, and the values must be valid expressions (or constants):

```
{
    "type": "dict",
    "properties": {
        "name": "a constant name",
        "value": {
            "type": "property_name",
            "property_name": "prop"
        },
        "value2": {
            "type": "property_name",
            "property_name": "prop2"
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

"Add Days" expressions

class corehq.apps.userreports.expressions.date_specs.**AddDaysExpressionSpec**

Below is a simple example that demonstrates the structure. The expression below will add 28 days to a property called "dob". The date_expression and count_expression can be any valid expressions, or simply constants.

```
{
    "type": "add_days",
    "date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
    "count_expression": 28
}
```

"Add Months" expressions

class corehq.apps.userreports.expressions.date_specs.**AddMonthsExpressionSpec**

`add_months` offsets given date by given number of calendar months. If offset results in an invalid day (for e.g. Feb 30, April 31), the day of resulting date will be adjusted to last day of the resulting calendar month.

The date_expression and months_expression can be any valid expressions, or simply constants, including negative numbers.

```
{
    "type": "add_months",
    "date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
    "months_expression": 28
}
```

"Diff Days" expressions

class corehq.apps.userreports.expressions.date_specs.**DiffDaysExpressionSpec**

`diff_days` returns number of days between dates specified by `from_date_expression` and `to_date_expression`. The from_date_expression and to_date_expression can be any valid expressions, or simply constants.

```
{
    "type": "diff_days",
    "from_date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
    "to_date_expression": "2016-02-01"
}
```

"Month Start Date" and "Month End Date" expressions

class corehq.apps.userreports.expressions.date_specs.**MonthStartDateExpressionSpec**

`month_start_date` returns date of first day in the month of given date and `month_end_date` returns date of last day in the month of given date.

The `date_expression` can be any valid expression, or simply constant

```
{
    "type": "month_start_date",
    "date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
}
```

"Evaluator" expression

class corehq.apps.userreports.expressions.specs.**EvalExpressionSpec**

`evaluator` expression can be used to evaluate statements that contain arithmetic (and simple python like statements). It evaluates the statement specified by `statement` which can contain variables as defined in `context_variables`.

```
{
    "type": "evaluator",
    "statement": "a + b - c + 6",
    "context_variables": {
        "a": 1,
        "b": 20,
        "c": 2
    }
}
```

This returns 25 (1 + 20 - 2 + 6).

`statement` can be any statement that returns a valid number. All python math operators except power operator are available for use.

`context_variables` is a dictionary of Expressions where keys are names of variables used in the `statement` and values are expressions to generate those variables. Variables can be any valid numbers (Python datatypes `int`, `float` and `long` are considered valid numbers.) or also expressions that return numbers. In addition to numbers the following types are supported:

- `date`
- `datetime`

Only the following functions are permitted:

- `rand()`: generate a random number between 0 and 1
- `randint(max)`: generate a random integer between 0 and `max`
- `int(value)`: convert `value` to an int. Value can be a number or a string representation of a number
- `float(value)`: convert `value` to a floating point number
- `str(value)`: convert `value` to a string

- `timedelta_to_seconds(time_delta)`: convert a TimeDelta object into seconds. This is useful for getting the number of seconds between two dates.
  - e.g. `timedelta_to_seconds(time_end - time_start)`
- `range(start, [stop], [skip])`: the same as the python `range` function <https://docs.python.org/2/library/functions.html#range>`__. Note that for performance reasons this is limited to 100 items or less.

'Get Case Sharing Groups' expression

class corehq.apps.userreports.expressions.specs.**CaseSharingGroupsExpressionSpec**

`get_case_sharing_groups` will return an array of the case sharing groups that are assigned to a provided user ID. The array will contain one document per case sharing group.

```
{
    "type": "get_case_sharing_groups",
    "user_id_expression": {
        "type": "property_path",
        "property_path": ["form", "meta", "userID"]
    }
}
```

'Get Reporting Groups' expression

class corehq.apps.userreports.expressions.specs.**ReportingGroupsExpressionSpec**

`get_reporting_groups` will return an array of the reporting groups that are assigned to a provided user ID. The array will contain one document per reporting group.

```
{
    "type": "get_reporting_groups",
    "user_id_expression": {
        "type": "property_path",
        "property_path": ["form", "meta", "userID"]
    }
}
```

Filter, Sort, Map and Reduce Expressions

We have following expressions that act on a list of objects or list of lists. The list to operate on is specified by `items_expression`. This can be any valid expression that returns a list. If the `items_expression` doesn't return a valid list, these might either fail or return one of empty list or `None` value.

map_items Expression

class corehq.apps.userreports.expressions.list_specs.**MapItemsExpressionSpec**

`map_items` performs a calculation specified by `map_expression` on each item of the list specified by `items_expression` and returns a list of the calculation results. The `map_expression` is evaluated relative to each item in the list and not relative to the parent document from which the list is specified. For e.g. if `items_expression` is a path to repeat-list of children in a form document, `map_expression` is a path relative to the repeat item.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`map_expression` can be any valid expression relative to the items in above list.

```
{
    "type": "map_items",
    "items_expression": {
        "datatype": "array",
        "type": "property_path",
        "property_path": ["form", "child_repeat"]
    },
    "map_expression": {
        "type": "property_path",
        "property_path": ["age"]
    }
}
```

Above returns list of ages. Note that the `property_path` in `map_expression` is relative to the repeat item rather than to the form.

filter_items Expression

class corehq.apps.userreports.expressions.list_specs.**FilterItemsExpressionSpec**

`filter_items` performs filtering on given list and returns a new list. If the boolean expression specified by `filter_expression` evaluates to a `True` value, the item is included in the new list and if not, is not included in the new list.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`filter_expression` can be any valid boolean expression relative to the items in above list.

```
{
    "type": "filter_items",
    "items_expression": {
        "datatype": "array",
        "type": "property_name",
        "property_name": "family_repeat"
    },
    "filter_expression": {
       "type": "boolean_expression",
        "expression": {
            "type": "property_name",
            "property_name": "gender"
        },
        "operator": "eq",
        "property_value": "female"
    }
}
```

sort_items Expression

class corehq.apps.userreports.expressions.list_specs.**SortItemsExpressionSpec**

`sort_items` returns a sorted list of items based on sort value of each item.The sort value of an item is specified by `sort_expression`. By default, list will be in ascending order. Order can be changed by adding optional `order` expression with one of `DESC` (for descending) or `ASC` (for ascending) If a sort-value of an item is `None`, the item will appear in the start of list. If sort-values of any two items can't be compared, an empty list is returned.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`sort_expression` can be any valid expression relative to the items in above list, that returns a value to be used as sort value.

```
{
    "type": "sort_items",
    "items_expression": {
        "datatype": "array",
        "type": "property_path",
        "property_path": ["form", "child_repeat"]
    },
    "sort_expression": {
        "type": "property_path",
        "property_path": ["age"]
    }
}
```

reduce_items Expression

class corehq.apps.userreports.expressions.list_specs.**ReduceItemsExpressionSpec**

`reduce_items` returns aggregate value of the list specified by `aggregation_fn`.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list, `aggregation_fn` will be applied on an empty list. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`aggregation_fn` is one of following supported functions names.

Function Name	Example
`count`	`['a', 'b']` -> 2
`sum`	`[1, 2, 4]` -> 7
`min`	`[2, 5, 1]` -> 1
`max`	`[2, 5, 1]` -> 5
`first_item`	`['a', 'b']` -> 'a'
`last_item`	`['a', 'b']` -> 'b'

```
{
    "type": "reduce_items",
    "items_expression": {
        "datatype": "array",
        "type": "property_name",
        "property_name": "family_repeat"
```

(continues on next page)

(continued from previous page)

```
    },
    "aggregation_fn": "count"
}
```

This returns number of family members

flatten expression

class corehq.apps.userreports.expressions.list_specs.**FlattenExpressionSpec**

`flatten` takes list of list of objects specified by `items_expression` and returns one list of all objects.

`items_expression` is any valid expression that returns a list of lists. It this doesn't evaluate to a list of lists an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

```
{
    "type": "flatten",
    "items_expression": {},
```

}

Named Expressions

class corehq.apps.userreports.expressions.specs.**NamedExpressionSpec**

Last, but certainly not least, are named expressions. These are special expressions that can be defined once in a data source and then used throughout other filters and indicators in that data source. This allows you to write out a very complicated expression a single time, but still use it in multiple places with a simple syntax.

Named expressions are defined in a special section of the data source. To reference a named expression, you just specify the type of `"named"` and the name as follows:

```
{
    "type": "named",
    "name": "my_expression"
}
```

This assumes that your named expression section of your data source includes a snippet like the following:

```
{
    "my_expression": {
        "type": "property_name",
        "property_name": "test"
    }
}
```

This is just a simple example - the value that `"my_expression"` takes on can be as complicated as you want *as long as it doesn't reference any other named expressions*.

Boolean Expression Filters

A `boolean_expression` filter combines an *expression*, an *operator*, and a *property value* (a constant), to produce a statement that is either `True` or `False`. *Note: in the future the constant value may be replaced with a second*

expression to be more general, however currently only constant property values are supported.

Here is a sample JSON format for simple `boolean_expression` filter:

```
{
    "type": "boolean_expression",
    "expression": {
        "type": "property_name",
        "property_name": "age",
        "datatype": "integer"
    },
    "operator": "gt",
    "property_value": 21
}
```

This is equivalent to the python statement: `doc["age"] > 21`

The following operators are currently supported:

Operator	Description	Value type	Example
`eq`	is equal	constant	`doc["age "] == 21`
`not_eq`	is not equal	constant	`doc["age "] != 21`
`in`	single value is in a list	list	`doc["col or"] in [" red", "blu e"]`
`in_multi`	a value is in a multi select	list	`selected (doc["colo r"], "red" )`
‘‘any_in_multi‘ ‘	one of a list of values in in a multiselect	list	`selected (doc["colo r"], ["red ", "blue"] )`
`lt`	is less than	number	`doc["age "] < 21`
`lte`	is less than or equal	number	`doc["age "] <= 21`
`gt`	is greater than	number	`doc["age "] > 21`
`gte`	is greater than or equal	number	`doc["age "] >= 21`

Compound filters

Compound filters build on top of `boolean_expression` filters to create boolean logic. These can be combined to support arbitrarily complicated boolean logic on data. There are three types of filters, *and*, *or*, and *not* filters. The JSON representation of these is below. Hopefully these are self explanatory.

The following filter represents the statement: `doc["age"] < 21 and doc["nationality"] == "american"`:

```
{
    "type": "and",
    "filters": [
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "age",
                "datatype": "integer"
            },
            "operator": "lt",
            "property_value": 21
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "nationality",
            },
            "operator": "eq",
            "property_value": "american"
        }
    ]
}
```

The following filter represents the statement: `doc["age"] > 21 or doc["nationality"] == "european"`:

```
{
    "type": "or",
    "filters": [
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "age",
                "datatype": "integer",
            },
            "operator": "gt",
            "property_value": 21
        },
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "nationality",
            },
            "operator": "eq",
            "property_value": "european"
        }
    ]
}
```

The following filter represents the statement: `!(doc["nationality"] == "european")`:

```
{
    "type": "not",
    "filter": [
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "nationality",
            },
            "operator": "eq",
            "property_value": "european"
        }
    ]
}
```

Note that this could be represented more simply using a single filter with the ‘‘not_eq‘‘ operator, but “not” filters can represent more complex logic than operators generally, since the filter itself can be another compound filter.

Practical Examples

See practical examples for some practical examples showing various filter types.

29.2.2 Indicators

Now that we know how to filter the data in our data source, we are still left with a very important problem: *how do we know what data to save*? This is where indicators come in. Indicators are the data outputs - what gets computed and put in a column in the database.

A typical data source will include many indicators (data that will later be included in the report). This section will focus on defining a single indicator. Single indicators can then be combined in a list to fully define a data source.

The overall set of possible indicators is theoretically any function that can take in a single document (form or case) and output a value. However the set of indicators that are configurable is more limited than that.

Indicator Properties

All indicator definitions have the following properties:

Property	Description
type	A specified type for the indicator. It must be one of the types listed below.
column_id	The database column where the indicator will be saved.
display_name	A display name for the indicator (not widely used, currently).
comment	A string describing the indicator

Additionally, specific indicator types have other type-specific properties. These are covered below.

Indicator types

The following primary indicator types are supported:

Indicator Type	Description
boolean	Save `1` if a filter is true, otherwise `0`.
expression	Save the output of an expression.
choice_list	Save multiple columns, one for each of a predefined set of choices
ledger_balances	Save a column for each product specified, containing ledger data

Note/todo: there are also other supported formats, but they are just shortcuts around the functionality of these ones they are left out of the current docs.

Now we see again the power of our filter framework defined above! Boolean indicators take any arbitrarily complicated filter expression and save a `1` to the database if the expression is true, otherwise a `0`. Here is an example boolean indicator which will save `1` if a form has a question with ID `is_pregnant` with a value of `"yes"`:

```
{
    "type": "boolean",
    "column_id": "col",
    "filter": {
        "type": "boolean_expression",
        "expression": {
            "type": "property_path",
            "property_path": ["form", "is_pregnant"],
        },
        "operator": "eq",
        "property_value": "yes"
    }
}
```

Similar to the boolean indicators - expression indicators leverage the expression structure defined above to create arbitrarily complex indicators. Expressions can store arbitrary values from documents (as opposed to boolean indicators which just store `0`'s and `1`'s). Because of this they require a few additional properties in the definition:

Property	Description
datatype	The datatype of the indicator. Current valid choices are: "date", "datetime", "string", "decimal", "integer", and "small_integer".
is_nullable	Whether the database column should allow null values.
is_primary_key	Whether the database column should be (part of?) the primary key. (TODO: this needs to be confirmed)
create_index	Creates an index on this column. Only applicable if using the SQL backend
expression	Any expression.
transform	(optional) transform to be applied to the result of the expression. (see "Report Columns > Transforms" section below)

Here is a sample expression indicator that just saves the "age" property to an integer column in the database:

```
{
    "type": "expression",
    "expression": {
        "type": "property_name",
        "property_name": "age"
    },
    "column_id": "age",
    "datatype": "integer",
    "display_name": "age of patient"
}
```

Choice list indicators take a single choice column (select or multiselect) and expand it into multiple columns where each column represents a different choice. These can support both single-select and multi-select quesitons.

A sample spec is below:

```
{
    "type": "choice_list",
    "column_id": "col",
    "display_name": "the category",
    "property_name": "category",
    "choices": [
        "bug",
```

(continues on next page)

(continued from previous page)

```
        "feature",
        "app",
        "schedule"
    ],
    "select_style": "single"
}
```

Ledger Balance indicators take a list of product codes and a ledger section, and produce a column for each product code, saving the value found in the corresponding ledger.

Property	Description
ledger_section	The ledger section to use for this indicator, for example, "stock"
product_codes	A list of the products to include in the indicator. This will be used in conjunction with the `column_id` to produce each column name.
case_id_expression	An expression used to get the case where each ledger is found. If not specified, it will use the row's doc id.

```
{
    "type": "ledger_balances",
    "column_id": "soh",
    "display_name": "Stock On Hand",
    "ledger_section": "stock",
    "product_codes": ["aspirin", "bandaids", "gauze"],
    "case_id_expression": {
        "type": "property_name",
        "property_name": "_id"
    }
}
```

This spec would produce the following columns in the data source:

soh_aspirin	soh_bandaids	soh_gauze
20	11	5
67	32	9

If the ledger you're using is a due list and you wish to save the dates instead of integers, you can change the "type" from "ledger_balances" to "due_list_dates".

Practical notes for creating indicators

These are some practical notes for how to choose what indicators to create.

All indicators output single values. Though fractional indicators are common, these should be modeled as two separate indicators (for numerator and denominator) and the relationship should be handled in the report UI config layer.

29.2.3 Saving Multiple Rows per Case/Form

You can save multiple rows per case/form by specifying a root level `base_item_expression` that describes how to get the repeat data from the main document. You can also use the `root_doc` expression type to reference parent properties and the `base_iteration_number` expression type to reference the current index of the item. This can be combined with the `iterator` expression type to do complex data source transforms. This is not described in

detail, but the following sample (which creates a table off of a repeat element called "time_logs" can be used as a guide). There are also additional examples in the practical examples:

```
{
    "domain": "user-reports",
    "doc_type": "DataSourceConfiguration",
    "referenced_doc_type": "XFormInstance",
    "table_id": "sample-repeat",
    "display_name": "Time Logged",
    "base_item_expression": {
        "type": "property_path",
        "property_path": ["form", "time_logs"]
    },
    "configured_filter": {
    },
    "configured_indicators": [
        {
            "type": "expression",
            "expression": {
                "type": "property_name",
                "property_name": "start_time"
            },
            "column_id": "start_time",
            "datatype": "datetime",
            "display_name": "start time"
        },
        {
            "type": "expression",
            "expression": {
                "type": "property_name",
                "property_name": "end_time"
            },
            "column_id": "end_time",
            "datatype": "datetime",
            "display_name": "end time"
        },
        {
            "type": "expression",
            "expression": {
                "type": "property_name",
                "property_name": "person"
            },
            "column_id": "person",
            "datatype": "string",
            "display_name": "person"
        },
        {
            "type": "expression",
            "expression": {
                "type": "root_doc",
                "expression": {
                    "type": "property_name",
                    "property_name": "name"
                }
            },
            "column_id": "name",
            "datatype": "string",
            "display_name": "name of ticket"
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

29.2.4 Data Cleaning and Validation

Note this is only available for "static" data sources that are created in the HQ repository.

When creating a data source it can be valuable to have strict validation on the type of data that can be inserted. The attribute `validations` at the top level of the configuration can use UCR expressions to determine if the data is invalid. If an expression is deemed invalid, then the relevant error is stored in the `InvalidUCRData` model.

```
{
    "domain": "user-reports",
    "doc_type": "DataSourceConfiguration",
    "referenced_doc_type": "XFormInstance",
    "table_id": "sample-repeat",
    "base_item_expression": {},
    "validations": [{
         "name": "is_starred_valid",
         "error_message": "is_starred has unexpected value",
         "expression": {
             "type": "boolean_expression",
             "expression": {
                 "type": "property_name",
                 "property_name": "is_starred"
             },
             "operator": "in",
             "property_value": ["yes", "no"]
         }
    }],
    "configured_filter": {...},
    "configured_indicators": [...]
}
```

29.3 Report Configurations

A report configuration takes data from a data source and renders it in the UI. A report configuration consists of a few different sections:

1. *Report Filters* - These map to filters that show up in the UI, and should translate to queries that can be made to limit the returned data.
2. *Aggregation* - This defines what each row of the report will be. It is a list of columns forming the *primary key* of each row.
3. *Report Columns* - Columns define the report columns that show up from the data source, as well as any aggregation information needed.
4. *Charts* - Definition of charts to display on the report.
5. *Sort Expression* - How the rows in the report are ordered.

29.3.1 Samples

Here are some sample configurations that can be used as a reference until we have better documentation.

- Dimagi chart report
- GSID form report

29.3.2 Report Filters

The documentation for report filters is still in progress. Apologies for brevity below.

A note about report filters versus data source filters

Report filters are *completely* different from data source filters. Data source filters limit the global set of data that ends up in the table, whereas report filters allow you to select values to limit the data returned by a query.

Numeric Filters

Numeric filters allow users to filter the rows in the report by comparing a column to some constant that the user specifies when viewing the report. Numeric filters are only intended to be used with numeric (integer or decimal type) columns. Supported operators are =, , <, , >, and .

ex:

```
{
  "type": "numeric",
  "slug": "number_of_children_slug",
  "field": "number_of_children",
  "display": "Number of Children"
}
```

Date filters

Date filters allow you filter on a date. They will show a datepicker in the UI.

```
{
  "type": "date",
  "slug": "modified_on",
  "field": "modified_on",
  "display": "Modified on",
  "required": false
}
```

Date filters have an optional `compare_as_string` option that allows the date filter to be compared against an indicator of data type `string`. You shouldn't ever need to use this option (make your column a `date` or `datetime` type instead), but it exists because the report builder needs it.

Quarter filters

Quarter filters are similar to date filters, but a choice is restricted only to the particular quarter of the year. They will show inputs for year and quarter in the UI.

```
{
  "type": "quarter",
  "slug": "modified_on",
  "field": "modified_on",
  "display": "Modified on",
  "required": false
}
```

Pre-Filters

Pre-filters offer the kind of functionality you get from *data source filters*. This makes it easier to use one data source for many reports, especially if some of those reports just need the data source to be filtered slightly differently. Pre-filters do not need to be configured by app builders in report modules; fields with pre-filters will not be listed in the report module among the other fields that can be filtered.

A pre-filter's `type` is set to "pre":

```
{
  "type": "pre",
  "field": "at_risk_field",
  "slug": "at_risk_slug",
  "datatype": "string",
  "pre_value": "yes"
}
```

If `pre_value` is scalar (i.e. `datatype` is "string", "integer", etc.), the filter will use the "equals" operator. If `pre_value` is null, the filter will use "is null". If `pre_value` is an array, the filter will use the "in" operator. e.g.

```
{
  "type": "pre",
  "field": "at_risk_field",
  "slug": "at_risk_slug",
  "datatype": "array",
  "pre_value": ["yes", "maybe"]
}
```

(If `pre_value` is an array and `datatype` is not "array", it is assumed that `datatype` refers to the data type of the items in the array.)

You can optionally specify the operator that the prevalue filter uses by adding a pre_operator argument. e.g.

```
{
  "type": "pre",
  "field": "at_risk_field",
  "slug": "at_risk_slug",
  "datatype": "array",
  "pre_value": ["maybe", "yes"],
  "pre_operator": "between"
}
```

Note that instead of using `eq`, `gt`, etc, you will need to use =, >, etc.

Dynamic choice lists

Dynamic choice lists provide a select widget that will generate a list of options dynamically.

The default behavior is simply to show all possible values for a column, however you can also specify a `choice_provider` to customize this behavior (see below).

Simple example assuming "village" is a name:

```
{
  "type": "dynamic_choice_list",
  "slug": "village",
  "field": "village",
  "display": "Village",
  "datatype": "string"
}
```

Currently the supported `choice_providers` are supported:

Field	Description
location	Select a location by name
user	Select a user
owner	Select a possible case owner owner (user, group, or location)

Location choice providers also support three additional configuration options:

- "include_descendants" - Include descendants of the selected locations in the results. Defaults to `false`.
- "show_full_path" - Display the full path to the location in the filter. Defaults to `false`. The default behavior shows all locations as a flat alphabetical list.
- "location_type" - Includes locations of this type only. Default is to not filter on location type.

Example assuming "village" is a location ID, which is converted to names using the location `choice_provider`:

```
{
  "type": "dynamic_choice_list",
  "slug": "village",
  "field": "location_id",
  "display": "Village",
  "datatype": "string",
  "choice_provider": {
      "type": "location",
      "include_descendants": true,
      "show_full_path": true,
      "location_type": "district"
  }
}
```

Choice lists

Choice lists allow manual configuration of a fixed, specified number of choices and let you change what they look like in the UI.

```
{
  "type": "choice_list",
  "slug": "role",
  "field": "role",
  "choices": [
    {"value": "doctor", "display": "Doctor"},
    {"value": "nurse"}
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

Drilldown by Location

This filter allows selection of a location for filtering by drilling down from top level.

```
{
  "type": "location_drilldown",
  "slug": "by_location",
  "field": "district_id",
  "include_descendants": true,
  "max_drilldown_levels": 3
}
```

- "include_descendants" - Include descendant locations in the results. Defaults to `false`.
- "max_drilldown_levels" - Maximum allowed drilldown levels. Defaults to 99

Internationalization

Report builders may specify translations for the filter display value. Also see the sections on internationalization in the Report Column and the *translations transform*.

```
{
    "type": "choice_list",
    "slug": "state",
    "display": {"en": "State", "fr": "État"},
    ...
}
```

29.3.3 Report Columns

Reports are made up of columns. The currently supported column types ares:

- *field* which represents a single value
- *percent* which combines two values in to a percent
- *aggregate_date* which aggregates data by month
- *expanded* which expands a select question into multiple columns
- *expression* which can do calculations on data in other columns

Field columns

Field columns have a type of `"field"`. Here's an example field column that shows the owner name from an associated `owner_id`:

```
{
    "type": "field",
    "field": "owner_id",
    "column_id": "owner_id",
    "display": "Owner Name",
    "format": "default",
    "transform": {
        "type": "custom",
        "custom_type": "owner_display"
    },
    "aggregation": "simple"
}
```

Percent columns

Percent columns have a type of `"percent"`. They must specify a `numerator` and `denominator` as separate field columns. Here's an example percent column that shows the percentage of pregnant women who had danger signs.

```
{
  "type": "percent",
  "column_id": "pct_danger_signs",
  "display": "Percent with Danger Signs",
  "format": "both",
  "denominator": {
    "type": "field",
    "aggregation": "sum",
    "field": "is_pregnant",
    "column_id": "is_pregnant"
  },
  "numerator": {
    "type": "field",
    "aggregation": "sum",
    "field": "has_danger_signs",
    "column_id": "has_danger_signs"
  }
}
```

The following percentage formats are supported.

Format	Description	example
percent	A whole number percentage (the default format)	33%
fraction	A fraction	1/3
both	Percentage and fraction	33% (1/3)
numeric_percent	Percentage as a number	33
decimal	Fraction as a decimal number	.333

AggregateDateColumn

AggregateDate columns allow for aggregating data by month over a given date field. They have a type of `"aggregate_date"`. Unlike regular fields, you do not specify how aggregation happens, it is automatically grouped by month.

Here's an example of an aggregate date column that aggregates the `received_on` property for each month (allowing you to count/sum things that happened in that month).

```
{
   "column_id": "received_on",
   "field": "received_on",
   "type": "aggregate_date",
   "display": "Month"
 }
```

AggregateDate supports an optional "format" parameter, which accepts the same format string as *Date formatting*. If you don't specify a format, the default will be "%Y-%m", which will show as, for example, "2008-09".

Keep in mind that the only variables available for formatting are `year` and `month`, but that still gives you a fair range, e.g.

format	Example result
"%Y-%m"	"2008-09"
"%B, %Y"	"September, 2008"
"%b (%y)"	"Sep (08)"

ConditionalAggregationColumn

NOTE This feature is only available to static UCR reports maintained by Dimagi developers.

Conditional aggregation columns allow you to define a series of conditional expressions with corresponding names, then group together rows which which meet the same conditions. They have a type of `"conditional_aggregation"`.

Here's an example that groups children based on their age at the time of registration:

```
{
    "display": "age_range",
    "column_id": "age_range",
    "type": "conditional_aggregation",
    "whens": {
        "0 <= age_at_registration AND age_at_registration < 12": "infant",
        "12 <= age_at_registration AND age_at_registration < 36": "toddler",
        "36 <= age_at_registration AND age_at_registration < 60": "preschooler"
    },
    "else_": "older"
}
```

The `"whens"` attribute maps conditional expressions to labels. If none of the conditions are met, the row will receive the `"else_"` value, if provided.

Here's a more complex example which uses SQL functions to dynamically calculate ranges based on a date property:

```
{
    "display": "Age Group",
    "column_id": "age_group",
    "type": "conditional_aggregation",
    "whens": {
        "extract(year from age(dob))*12 + extract(month from age(dob)) BETWEEN 0 and 5
↪": "0_to_5",
        "extract(year from age(dob))*12 + extract(month from age(dob)) BETWEEN 6 and␣
↪11": "6_to_11",
        "extract(year from age(dob))*12 + extract(month from age(dob)) BETWEEN 12 and␣
↪35": "12_to_35",
```

(continues on next page)

(continued from previous page)

```
        "extract(year from age(dob))*12 + extract(month from age(dob)) BETWEEN 36 and␣
↪59": "36_to_59",
        "extract(year from age(dob))*12 + extract(month from age(dob)) BETWEEN 60 and␣
↪71": "60_to_71"
    }
}
```

Expanded Columns

Expanded columns have a type of `"expanded"`. Expanded columns will be "expanded" into a new column for each distinct value in this column of the data source. For example:

If you have a data source like this:

```
+---------|----------|-------------+
| Patient | district | test_result |
+---------|----------|-------------+
| Joe     | North    | positive    |
| Bob     | North    | positive    |
| Fred    | South    | negative    |
+---------|----------|-------------+
```

and a report configuration like this:

```
aggregation columns:
["district"]

columns:
[
  {
    "type": "field",
    "field": "district",
    "column_id": "district",
    "format": "default",
    "aggregation": "simple"
  },
  {
    "type": "expanded",
    "field": "test_result",
    "column_id": "test_result",
    "format": "default"
  }
]
```

Then you will get a report like this:

```
+----------|----------------------|----------------------+
| district | test_result-positive | test_result-negative |
+----------|----------------------|----------------------+
| North    | 2                    | 0                    |
| South    | 0                    | 1                    |
+----------|----------------------|----------------------+
```

Expanded columns have an optional parameter `"max_expansion"` (defaults to 10) which limits the number of columns that can be created. WARNING: Only override the default if you are confident that there will be no adverse performance implications for the server.

Expression columns

Expression columns can be used to do just-in-time calculations on the data coming out of reports. They allow you to use any UCR expression on the data in the report row. These can be referenced according to the `column_id`s from the other defined column. They can support advanced use cases like doing math on two different report columns, or doing conditional logic based on the contents of another column.

A simple example is below, which assumes another called "number" in the report and shows how you could make a column that is 10 times that column.

```
{
    "type": "expression",
    "column_id": "by_tens",
    "display": "Counting by tens",
    "expression": {
        "type": "evaluator",
        "statement": "a * b",
        "context_variables": {
            "a": {
                "type": "property_name",
                "property_name": "number"
            },
            "b": 10
        }
    }
}
```

The "aggregation" column property

The aggregation column property defines how the column should be aggregated. If the report is not doing any aggregation, or if the column is one of the aggregation columns this should always be `"simple"` (see *Aggregation* below for more information on aggregation).

The following table documents the other aggregation options, which can be used in aggregate reports.

Format	Description
simple	No aggregation
avg	Average (statistical mean) of the values
count_unique	Count the unique values found
count	Count all rows
min	Choose the minimum value
max	Choose the maximum value
sum	Sum the values

Column IDs in percentage fields *must be unique for the whole report*. If you use a field in a normal column and in a percent column you must assign unique `column_id` values to it in order for the report to process both.

Calculating Column Totals

To sum a column and include the result in a totals row at the bottom of the report, set the `calculate_total` value in the column configuration to `true`.

Not supported for the following column types: - expression

Internationalization

Report columns can be translated into multiple languages. To translate values in a given column check out the *translations transform* below. To specify translations for a column header, use an object as the `display` value in the configuration instead of a string. For example:

```
{
    "type": "field",
    "field": "owner_id",
    "column_id": "owner_id",
    "display": {
        "en": "Owner Name",
        "he": ""
    },
    "format": "default",
    "transform": {
        "type": "custom",
        "custom_type": "owner_display"
    },
    "aggregation": "simple"
}
```

The value displayed to the user is determined as follows: - If a display value is specified for the users language, that value will appear in the report. - If the users language is not present, display the `"en"` value. - If `"en"` is not present, show an arbitrary translation from the `display` object. - If `display` is a string, and not an object, the report shows the string.

Valid `display` languages are any of the two or three letter language codes available on the user settings page.

29.3.4 Aggregation

Aggregation in reports is done using a list of columns to aggregate on. This defines how indicator data will be aggregated into rows in the report. The columns represent what will be grouped in the report, and should be the `column_id`s of valid report columns. In most simple reports you will only have one level of aggregation. See examples below.

No aggregation

Note that if you use `is_primary_key` in any of your columns, you must include all primary key columns here.

```
["doc_id"]
```

Aggregate by 'username' column

```
["username"]
```

Aggregate by two columns

```
["column1", "column2"]
```

29.3.5 Transforms

Transforms can be used in two places - either to manipulate the value of a column just before it gets saved to a data source, or to transform the value returned by a column just before it reaches the user in a report. Here's an example of a transform used in a report config 'field' column:

```
{
    "type": "field",
    "field": "owner_id",
    "column_id": "owner_id",
    "display": "Owner Name",
    "format": "default",
    "transform": {
        "type": "custom",
        "custom_type": "owner_display"
    },
    "aggregation": "simple"
}
```

The currently supported transform types are shown below:

Translations and arbitrary mappings

The translations transform can be used to give human readable strings:

```
{
    "type": "translation",
    "translations": {
        "lmp": "Last Menstrual Period",
        "edd": "Estimated Date of Delivery"
    }
}
```

And for translations:

```
{
    "type": "translation",
    "translations": {
        "lmp": {
            "en": "Last Menstrual Period",
            "es": "Fecha Última Menstruación",
        },
        "edd": {
            "en": "Estimated Date of Delivery",
            "es": "Fecha Estimada de Parto",
        }
    }
}
```

To use this in a mobile ucr, set the `'mobile_or_web'` property to `'mobile'`

```
{
    "type": "translation",
    "mobile_or_web": "mobile",
    "translations": {
        "lmp": "Last Menstrual Period",
```

(continues on next page)

(continued from previous page)

```
        "edd": "Estimated Date of Delivery"
    }
}
```

Displaying username instead of user ID

```
{
    "type": "custom",
    "custom_type": "user_display"
}
```

Displaying username minus @domain.commcarehq.org instead of user ID

```
{
    "type": "custom",
    "custom_type": "user_without_domain_display"
}
```

Displaying owner name instead of owner ID

```
{
    "type": "custom",
    "custom_type": "owner_display"
}
```

Displaying month name instead of month index

```
{
    "type": "custom",
    "custom_type": "month_display"
}
```

Rounding decimals

Rounds decimal and floating point numbers to two decimal places.

```
{
    "type": "custom",
    "custom_type": "short_decimal_display"
}
```

Generic number formatting

Rounds numbers using Python's built in formatting.

See below for a few simple examples. Read the docs for complex ones. The input to the format string will be a *number* not a string.

If the format string is not valid or the input is not a number then the original input will be returned.

```
{
    "type": "number_format",
    "format_string": "{0:.0f}"
}
```

```
{
    "type": "number_format",
    "format_string": "{0:.3f}"
}
```

Date formatting

Formats dates with the given format string. See here for an explanation of format string behavior. If there is an error formatting the date, the transform is not applied to that value.

```
{
   "type": "date_format",
   "format": "%Y-%m-%d %H:%M"
}
```

Converting an ethiopian date string to a gregorian date

Converts a string in the YYYY-MM-DD format to a gregorian date. For example, 2009-09-11 is converted to date(2017, 5, 19). If it is unable to convert the date, it will return an empty string.

```
{
   "type": "custom",
   "custom_type": "ethiopian_date_to_gregorian_date"
}
```

Converting a gregorian date string to an ethiopian date

Converts a string in the YYYY-MM-DD format to an ethiopian date. For example, 2017-05-19 is converted to date(2009, 09, 11). If it is unable to convert the date, it will return an empty string.

```
{
   "type": "custom",
   "custom_type": "gregorian_date_to_ethiopian_date"
}
```

29.3.6 Charts

There are currently three types of charts supported. Pie charts, and two types of bar charts.

Pie charts

A pie chart takes two inputs and makes a pie chart. Here are the inputs:

Field	Description
aggregation_colu mn	The column you want to group - typically a column from a select question
value_column	The column you want to sum - often just a count

Here's a sample spec:

```
{
    "type": "pie",
    "title": "Remote status",
    "aggregation_column": "remote",
    "value_column": "count"
}
```

Aggregate multibar charts

An aggregate multibar chart is used to aggregate across two columns (typically both of which are select questions). It takes three inputs:

Field	Description
primary_aggregation	The primary aggregation. These will be the x-axis on the chart.
secondary_aggregati on	The secondary aggregation. These will be the slices of the bar (or individual bars in "grouped" format)
value_column	The column you want to sum - often just a count

Here's a sample spec:

```
{
    "type": "multibar-aggregate",
    "title": "Applicants by type and location",
    "primary_aggregation": "remote",
    "secondary_aggregation": "applicant_type",
    "value_column": "count"
}
```

Multibar charts

A multibar chart takes a single x-axis column (typically a user, date, or select question) and any number of y-axis columns (typically indicators or counts) and makes a bar chart from them.

Field	Description
x_axis_column	This will be the x-axis on the chart.
y_axis_columns	These are the columns to use for the secondary axis. These will be the slices of the bar (or individual bars in "grouped" format).

Here's a sample spec:

```
{
    "type": "multibar",
    "title": "HIV Mismatch by Clinic",
    "x_axis_column": "clinic",
    "y_axis_columns": [
```

(continues on next page)

(continued from previous page)

```
        {
            "column_id": "diagnoses_match_no",
            "display": "No match"
        },
        {
            "column_id": "diagnoses_match_yes",
            "display": "Match"
        }
    ]
}
```

29.3.7 Sort Expression

A sort order for the report rows can be specified. Multiple fields, in either ascending or descending order, may be specified. Example:

Field should refer to report column IDs, not database fields.

```
[
  {
    "field": "district",
    "order": "DESC"
  },
  {
    "field": "date_of_data_collection",
    "order": "ASC"
  }
]
```

29.4 Mobile UCR

Mobile UCR is a beta feature that enables you to make application modules and charts linked to UCRs on mobile. It also allows you to send down UCR data from a report as a fixture which can be used in standard case lists and forms throughout the mobile application.

The documentation for Mobile UCR is very sparse right now.

29.4.1 Filters

On mobile UCR, filters can be automatically applied to the mobile reports based on hardcoded or user-specific data, or can be displayed to the user.

The documentation of mobile UCR filters is incomplete. However some are documented below.

Custom Calendar Month

When configuring a report within a module, you can filter a date field by the 'CustomMonthFilter'. The choice includes the following options: - Start of Month (a number between 1 and 28) - Period (a number between 0 and n with 0 representing the current month).

Each custom calendar month will be "Start of the Month" to ("Start of the Month" - 1). For example, if the start of the month is set to 21, then the period will be the 21th of the month -> 20th of the next month.

Examples: Assume it was May 15: Period 0, day 21, you would sync April 21-May 15th Period 1, day 21, you would sync March 21-April 20th Period 2, day 21, you would sync February 21 -March 20th

Assume it was May 20: Period 0, day 21, you would sync April 21-May 20th Period 1, day 21, you would sync March 21-April 20th Period 2, day 21, you would sync February 21-March 20th

Assume it was May 21: Period 0, day 21, you would sync May 21-May 21th Period 1, day 21, you would sync April 21-May 20th Period 2, day 21, you would sync March 21-April 20th

29.5 Export

A UCR data source can be exported, to back an excel dashboard, for instance. The URL for exporting data takes the form https://www.commcarehq.org/a/{[}domain{]}/configurable_reports/data_sources/export/{[}data source id]/ The export supports a "$format" parameter which can be any of the following options: html, csv, xlsx, xls. The default format is csv.

This export can also be filtered to restrict the results returned. The filtering options are all based on the field names:

URL parameter	Value	Description
{field_name}	{exact value}	require an exact match
{field_name}-range	{start}..{end}	return results in range
{field_name}-lastndays	{number}	restrict to the last n days

This is configured in `export_data_source` and tested in `test_export`. It should be pretty straightforward to add support for additional filter types.

Let's say you want to restrict the results to only cases owned by a particular user, opened in the last 90 days, and with a child between 12 and 24 months old as an xlsx file. The querystring might look like this:

```
?$format=xlsx&owner_id=481069n24myxk08hl563&opened_on-lastndays=90&child_age-range=12.
↪.24
```

29.6 Practical Notes

Some rough notes for working with user configurable reports.

29.6.1 Getting Started

The easiest way to get started is to start with sample data and reports.

First copy the dimagi domain to your developer machine. You only really need forms, users, and cases:

```
./manage.py copy_domain https://<your_username>:<your_password>@commcarehq.cloudant.
↪com/commcarehq dimagi --include=CommCareCase,XFormInstance,CommCareUser
```

Then load and rebuild the data table:

```
./manage.py load_spec corehq/apps/userreports/examples/dimagi/dimagi-case-data-source.
↪json --rebuild
```

Then load the report:

```
./manage.py load_spec corehq/apps/userreports/examples/dimagi/dimagi-chart-report.json
```

Fire up a browser and you should see the new report in your domain. You should also be able to navigate to the edit UI, or look at and edit the example JSON files. There is a second example based off the "gsid" domain as well using forms.

The tests are also a good source of documentation for the various filter and indicator formats that are supported.

29.6.2 Static data sources

As well as being able to define data sources via the UI which are stored in the database you can also define static data sources which live as JSON documents in the source repository.

These are mainly useful for custom reports.

They conform to a slightly different style:

```
{
    "domains": ["live-domain", "test-domain"],
    "config": {
        ... put the normal data source configuration here
    }
}
```

Having defined the data source you need to add the path to the data source file to the `STATIC_DATA_SOURCES` setting in `settings.py`. Now when the static data source pillow is run it will pick up the data source and rebuild it.

Changes to the data source require restarting the pillow which will rebuild the SQL table. Alternately you can use the UI to rebuild the data source (requires Celery to be running).

29.6.3 Static configurable reports

Configurable reports can also be defined in the source repository. Static configurable reports have the following style:

```
{
    "domains": ["my-domain"],
    "data_source_table": "my_table",
    "report_id": "my-report",
    "config": {
        ... put the normal report configuration here
    }
}
```

29.6.4 Custom configurable reports

Sometimes a client's needs for a rendered report are outside of the scope of the framework. To render the report using a custom Django template or with custom Excel formatting, define a subclass of `ConfigurableReportView` and override the necessary functions. Then include the python path to the class in the field `custom_configurable_report` of the static report and don't forget to include the static report in `STATIC_DATA_SOURCES` in `settings.py`.

29.6.5 Extending User Configurable Reports

When building a custom report for a client, you may find that you want to extend UCR with custom functionality. The UCR framework allows developers to write custom expressions, and register them with the framework. To do so, simply add a tuple to the `CUSTOM_UCR_EXPRESSIONS` setting list. The first item in the tuple is the name of the expression type, the second item is the path to a function with a signature like conditional_expression(spec, context) that returns an expression object. e.g.:

```
# settings.py

CUSTOM_UCR_EXPRESSIONS = [
    ('abt_supervisor', 'custom.abt.reports.expressions.abt_supervisor'),
]
```

Following are some custom expressions that are currently available.

- `location_type_name`: A way to get location type from a location document id.
- `location_parent_id`: A shortcut to get a location's parent ID a location id.
- `get_case_forms`: A way to get a list of forms submitted for a case.
- `get_subcases`: A way to get a list of subcases (child cases) for a case.
- `indexed_case`: A way to get an indexed case from another case.

You can find examples of these in practical examples.

29.6.6 Scaling UCR

Profiling data sources

You can use `./manage.py profile_data_source <domain> <data source id> <doc id>` to profile a datasource on a particular doc. It will give you information such as functions that take the longest and number of database queries it initiates.

Faster Reporting

If reports are slow, then you can add `create_index` to the data source to any columns that have filters applied to them.

Asynchronous Indicators

If you have an expensive data source and the changes come in faster than the pillow can process them, you can specify `asynchronous: true` in the data source. This flag puts the document id in an intermediary table when a change happens which is later processed by a celery queue. If multiple changes are submitted before this can be processed, a new entry is not created, so it will be processed once. This moves the bottle neck from kafka/pillows to celery.

The main benefit of this is that documents will be processed only once even if many changes come in at a time. This makes this approach ideal datasources that don't require 'live' data or where the source documents change very frequently.

It is also possible achieve greater parallelization than is currently available via pillows since multiple Celery workers can process the changes.

A diagram of this workflow can be found here

29.6.7 Inspecting database tables

The easiest way to inspect the database tables is to use the sql command line utility.

This can be done by runnning `./manage.py dbshell` or using `psql`.

The naming convention for tables is: `config_report_[domain name]_[table id]_[hash]`.

In postgres, you can see all tables by typing `\dt` and use sql commands to inspect the appropriate tables.

CHAPTER 30

Dimagi JavaScript Guide

Dimagi's internal JavaScript guide for use in the CommCare HQ project

30.1 Table of contents

CHAPTER 31

Tips for documenting

31.1 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.

31.1.1 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

31.1.2 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

31.1.3 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.

31.1.4 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

31.1.5 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 32

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X