Open Data Kit 2 Documentation

Open Data Kit 2

Jan 13, 2019

CONTENTS

1 Learn More 3

2 Perspectives 5

3 List of Tools 7

4 Trying It Out 9

The *ODK 2 Tool Suite* is a new set of ODK tools that will co-exist with the existing ODK 1 Tool Suite. It targets advanced users who find themselves limited by the ODK 1 data collection workflows. It provides:

- **Fully customizable layout of prompts on the Android device**. The ODK 2 tools use HTML, JavaScript, and CSS to specify the layout of nearly all the screens viewed by the data collectors. This enables individuals and organizations with basic web development skills to modify and customize the appearance of their surveys and workflow. At the same time, we retain the easy-to-use spreadsheet-based definition of the survey questions (however, this XLSX Converter mechanism is not cross-compatible with XLSForm).
- **More flexible, user-directed, navigation of a survey**. The ODK 2 tools do not impose a strict sequential advancement through a form like ODK Collect. Form designers can allow users to traverse a form in any order, yet impose validation of collected data prior to traversing into subsequent steps in a workflow.
- **Improved treatment of repeat-groups**. In the ODK 2 tools, we have eliminated the concept of a repeat-group. In its place, we provide prompts that enable you to open and edit other surveys with links back to the originating survey (if desired). These prompts can describe a sub-form (nested) relationship among the surveys (for example: household and household-member) or they can represent arbitrary relational linkages across your data (for example: tea-houses and tea-types).
- **Bi-directional synchronization of data across devices**. The ODK 2 tools support the collaborative sharing of survey data across devices, as well as the updating and submission of changes to previously collected data (for example: follow-up surveys) via a bi-directional synchronization protocol. This contrasts with the unidirectional device-to-server submission pathway of ODK Collect / ODK Aggregate / ODK Briefcase.
- **Data curation and visualization on the device**. ODK Tables gives organizations the ability to investigate and visualize entire datasets directly on the Android devices through graphical and non-graphical displays and through filtered views.
- **Row-level access filters**. The visibility of the data and the ability to edit and/or delete data can be restricted for different users and groups.

Note: The ODK 2 tool suite is targeted at advanced users who are unable to complete their workflows with the ODK 1 tools. If you find that the ODK 1 tools meet your needs then there is no reason to switch.

ODK 2 is a platform that will run your *data management applications*. With ODK 1 you create survey forms that ODK Collect renders and are used to collect data and submit it to the ODK Aggregate server. In ODK 2, you create *data management applications* that consist of survey forms (similar to those used in Collect) as well as Javascript based web apps that allow you to render a fully customized user interface and implement business logic in order to collect, manage, and visualize data all on the Android device.

CHAPTER

ONE

LEARN MORE

View a brief feature comparison in the guide for *Selecting the Appropriate Tool Suite*.

CHAPTER

TWO

PERSPECTIVES

The ODK 2 documentation is organized into different perspectives to allow readers to focus on the sections most relevant to their needs.

These perspectives are:

- *User*: A user of the data management application running on the Android tools. This person collects and interacts with data on the Android device. Examples include field workers, clinic staff, and census workers.
- *Deployment Architect*: An author of a data management application or a consumer of collected data. This person might create forms and edit Javascript on their computer to deploy to the Android device. Or they might download data from the server and use Excel to perform analysis. Examples include technical staff and data analytics staff.
- *System Administrator*: The person or team that manages the web servers in the cloud. This person might want to ensure high availability of the server or coordinate across multiple regions.
- *Platform Developer*: A programmer that intends to modify the source code of the ODK 2 tools themselves. This person might want to add a new view type or a fix a bug.

Note: We expect most organizations to have users, deployment architects, and system administrators (though these roles might overlap). But most organizations can safely ignore the platform developer sections.

CHAPTER

THREE

LIST OF TOOLS

The ODK 2 Tool Suite consists of:

- *ODK Application Designer* - a design environment for creating, customizing, and previewing your forms, data curation, and visualization applications.
- *ODK Survey* - a data collection application based upon HTML, CSS, JavaScript.
- *ODK Tables* - a data curation and visualization application running on your mobile device.
- *ODK Services* - an application for handling database access, file access, and data synchronization services between all the ODK 2 applications. It allows you to synchronize data collected by the ODK 2 Android tools with a cloud endpoint.
- *ODK Cloud Endpoints* - a cloud server to host data and application files, and to support bi-directional data synchronization across disconnected mobile devices.
- *ODK Suitcase* - a desktop tool for synchronizing data with a cloud endpoint.

CHAPTER

FOUR

TRYING IT OUT

The first step to get a feel for the ODK 2 tools and how they fit together is to do the *Getting Started User Guide*. This guide walks you through the process of using a basic geotagging application and submitting data to the server. After this is completed, the guide provides a list of *Next Steps* for the user.

Deployment Architects should follow this up with the *Getting Started Deployment Architect Guide* to get an introduction into revising and editing their own forms. That guide walks you through modifying the *Geotagger* demo application to add a new field to it. Similar to the user guide, this guide provides a list of *Next Steps* for the Deployment Architect.

System Administrators should learn about *ODK Cloud Endpoints*.

Platform Developers should already be familiar with everything from the User and Deployment Architect sections, and can learn more in the *Platform Developer Advanced Topics* documentation.

4.1 Selecting the Appropriate Tool Suite

Generally, we suggest starting with the ODK 1 tool suite. If it does not fulfill your requirements then move on to the more flexible, but also more complex, ODK 2 tool suite.

Note: It is tempting to look at the version number and assume the latest is the greatest, but this is not always the case. The ODK 2 tool suite was designed to co-exist with the ODK 1 tool suite and does not replace any 1.x tools. In general, the ODK 1 tools are easier to use, require less setup, and are widely adopted. However, if you have a complex longitudinal study and possess some technical skills, then the ODK 2 tools may be better suited to your needs.

The feature comparison table below illustrates the differences between the ODK 1 and ODK 2 tools.

Table 4.1: Feature Comparison Table

Feature	ODK 1	ODK 2
Stage of technology lifecycle	Maturity	Introductory
Collect data with mobile device	x	x
Widely adopted	x	
Drag and drop tool to create forms	x	
Transmit collected data from device to server	x	x
Ability to capture rich data types (e.g. GPS, Images, Audio, Video)	x	x
One to **one** mapping of a question to database fields (except for GPS)	x	x
One to **many** mapping of a question to database fields		x
Static input contraint checks	x	x
Dynamic input contraint		x

4.1.1 Trying Them Out

- Try the 1.x tools with the Getting Started With ODK guide.
- Try the 2.x tools with the *Getting Started User Guide*.

4.2 Getting Started User Guide

The ODK 2 tools are intended to address limitations of the existing tool set. The 2 Tool Suite consists of:

- *ODK Services* - an application that handles database access, file access, and data synchronization services between all of the ODK 2 applications. It also allows you to synchronize data collected by the ODK 2 tools using the 2 protocol with an ODK Aggregate instance.
- *ODK Survey* - a data collection application based upon HTML, CSS, and JavaScript.
- *ODK Tables* - a data collection and visualization application running on your device.
- *ODK Cloud Endpoints* - a ready-to-deploy server and data repository with enhancements to support bi-directional data synchronization across disconnected devices.
- *ODK Application Designer* - a design environment for creating, customizing, and previewing your forms.
- *ODK Suitcase* - a desktop tool for synchronizing data from an ODK 2 server so the data can be exported to CSV format.

This page provides a brief end-to-end walk-through of the ODK 2 tools. It will cover the following topics:

- *ODK Data Management Applications*
- *Joining a device to an Existing Aggregate Server*
- *Tour of the Simple Demo Application*
  - *Checkpoint Resolution*
  - *Conflict Resolution*
- *Next Steps*

4.2.1 ODK Data Management Applications

The ODK 2 Android tools (ODK Survey, ODK Tables, ODK Services, ODK Scan, ODK Sensors Framework, and various ODK Sensor implementations) are APKs that are designed to work together to create a coherent tailored application experience for an end-user.

Note: Together the ODK 2 tools create a platform, on top of which you can build your own data management applications.

ODK 2 tools access configuration files and store data under sub-directories of the `opendatakit` directory in the `sdcard` root directory (whether your device has a physical SD card or not): `/sdcard/opendatakit`. User applications constructed using the ODK 2 tools are identified by the name of the sub-directory holding those configuration and data files. Thus, `/sdcard/opendatakit/mytestapp` would contain all the files and data for the *mytestapp* application. The name of that sub-directory, *mytestapp,* is referred to as the **AppName** of that application. The default **AppName** for the ODK tools is *default.* However, when configured appropriately, the ODK tools can run under another **AppName**, accessing configuration and saving data in a different subdirectory under opendatakit.

This is handled in such a way that each user application is isolated from all other user applications, with separate configurations, data tables, and server settings. This allows one device to run multiple user applications built on top of the ODK 2 tools without any coordination among the teams developing those applications.

A major goal of the 2 tools was to eliminate the need for any software engineering skills (for example: Java programming, Android software development environment, source code version control systems) when designing data management applications. The skills required to build a data management application range from scripting a form definition in XLSX (similar to constructing ODK Collect forms using XLSX files processed by the XLSForm tool), to simple web programming – modifying boilerplate HTML and JavaScript for custom presentations of the collected data. Advanced web programmers can also easily implement entirely custom web pages.

4.2.2 Joining a device to an Existing Aggregate Server

The steps for joining a device to an existing Aggregate server are straightforward.

1. Install the APKs your application uses.
2. Launch the *home screen* APK, either ODK Survey or ODK Tables.
3. Click on the three vertical dots in the upper right corner of the menu bar and choose *Sync* to launch the ODK Services sync activity in the context of your *home screen* APK.
4. Configure ODK Services to point to the ODK Aggregate instance you want to join.

5. Choose *Sync now* to make the device mirror the contents on that ODK Aggregate server.

Follow the steps described above to join the ODK Aggregate server hosting our simple demo, which uses ODK Tables as its *home_screen* APK. The detailed steps are:

1. Download and install ODK Services, ODK Tables, and ODK Survey.
2. Launch ODK Tables (the *home_screen* APK).
3. Click on the three vertical dots in the upper right corner of the menu bar and choose *Sync* to launch the ODK Services.
4. The default Sync Configuration should be *https://open-data-kit.appspot.com* and *None (anonymous access)*. You will need to change that. It will also default to *Fully Sync Attachments*.
5. Click on the three vertical dots in the menu bar, select *Settings* → *Server Settings*.
6. Click on *Server URL* and replace the default server with *https://opendatakit-simpledemo.appspot.com* then click *OK*.
7. Back out of settings then choose *Sync Now*.

The synchronization process will now occur.

Note: If there is an error, check to make sure the server URL is correct, then choose *Sync Now* again until it completes successfully.

Once successful, back out of ODK Services, returning to ODK Tables. And back out of ODK Tables. Then relaunch ODK Tables.

4.2.3 Tour of the Simple Demo Application

You should now see the custom home screen for the *Geotagger* demo:

This demo is based upon the *geotagger* data table and form. It allows users to record the date, time, GPS coordinates, description, and picture of their current location.

When you launch the demo by clicking the blue launch button, you see a map showing the collected data points, indicated with markers. By clicking on a marker, you bring its data record to the top of the list of records above the map. Clicking on the record header will expand or contract that item to show the coordinates and photo of that location. For example, if we click on the *Phinney Ridge* marker, its color changes from blue to green, and, if we then touch the *Phinney Ridge* heading, it expands to show the coordinates and image of that location:

You can add a new data record by choosing the + icon in top menu bar. This opens ODK Survey.

Note: Since ODK Survey is being opened for the first time, it will initialize itself. This may take a few moments.

Advance through and finalize this form. Upon finalizing the form, you will be returned to ODK Tables and its map view. You can then highlight the marker you added and view the image in the list view:

If you then click or tap in the list item details area (on the image), a detail view of the item will be displayed.

From here, if you were to choose the pencil icon, ODK Survey would be launched to edit this record.

You can also view the data in a list view or spreadsheet view by choosing the sheet icon in the menu bar and selecting the view you want:

Tip: These other views can be useful if you need to access and complete data records that do not yet have location data and cannot therefore be displayed on a map. Try these other views now.

Now back out of the *geotagger* table view and return to the custom home screen. Choose the three-horizontal-line icon on the top menu bar and choose *Sync*. This opens up ODK Services in its sync activity. Sync your device with the server (choose *Sync Noaw*). This will push your newly added record to the server. You can see this by browsing to https://opendatakit-simpledemo.appspot.com click on the *ODK Tables* tab, choose the *View Table* sub-tab, and select the *geotagger* table.

If you then repeat these steps with a different device, you can see that the two devices can share and exchange data, and revisions to this data, whenever they synchronize to the server.

Note: During this process, there are two problem-resolution screens you are likely to encounter:

- *Checkpoint Resolution* - if ODK Survey exits without the user explicitly saving their additions or changes.
- *Conflict Resolution* - if ODK Services detects a change on the server to a data record that was also changed on the device.

Checkpoint Resolution

The checkpoint resolution screen can be triggered a variety of ways. For this tour, choose the + icon then back out of ODK Survey:

When presented with this screen, there are three choices:

- Cancel and continue editing the form.
- Ignore changes and discard the entire partially filled-out form.
- Save it even though it is incomplete. In this case, since there is no entered data for this record, we can ignore changes.

In rare cases, a second form of checkpoint resolution screen can be triggered. This most often happens if ODK Survey experiences a failure and closes. In this case, you may have several data records with unsaved checkpoint changes (changes that the user has not explicitly saved as incomplete or finalized). This will lead to a screen like:

Clicking a row will display details about that individual checkpoint:

In all of these screens, you can choose whether to save the changes as incomplete or to discard them.

Conflict Resolution

The conflict resolution screen is triggered when another device has edited one or more rows and synchronized its changes to the server before your edits to those same rows have been synchronized. In this case, your synchronization attempt will end with an error, and a *Conflicts Detected* error will appear:

Once you click *OK*, the conflict resolution screen will be presented. If there are multiple rows in conflict, this screen will display the rows that are in conflict:

Clicking a row will display details about the conflict:

And if only a single row is in conflict, the list-of-rows screen will be bypassed.

The conflict details screen displays the values of the field(s) in conflict, with the field value on the device (Local) appearing first. In this case, the *Description* field is in conflict. The device has *Kite hill at Gasworks* and the server has *Kite Hill … Gasworks*. You can select either to take your device values (*Take Local Version*) or take the server's values (*Take Server Version*) or pick-and-choose among the changes and merge them (the *Merge Changes as Indicated Below* button will be enabled after all fields have had either their Local or Server value picked for the merge). After selecting the local version or choosing to merge, you must again synchronize with the server to push that change up to the server.

> **Warning:** When you resolve a conflict, your decision does not only affect you. The value you choose becomes the new true value and the next time you sync it will be written to the server.

This concludes the tour of the *Geotagger* example application's screens, and the functionality within ODK Tables. For larger tours of sample applications, try the *ODK Survey: Sample Application* and *ODK Tables: Sample Application*.

4.2.4 Next Steps

Users can browse the user guides for the Android tools. Tables and Survey's documentation each guide you through the use of sample application to better familiarize with the workflow of each tool.

- *ODK Survey*
- *ODK Tables*
- *ODK Services*

Development Architects should continue this tour in the *Getting Started Deployment Architect Guide*.

4.3 Getting Started Deployment Architect Guide

This guide is intended for Deployment Architects. A Deployment Architect is an author of a data management application or a consumer of collected data. This person might create forms and edit Javascript on their computer to deploy to the Android device. Or they might download data from the server and use Excel to perform analysis. Examples include technical staff and data analytics staff.

Other perspective definitions can be found *here*.

- *Prerequisites*
- *Migrating / Setting-up an ODK 2 application*
  - *Setting up the ODK Aggregate server*
  - *Resetting the Application on the Server*
- *Configuring your Device with an Application*
  - *Setting up ODK Application Designer*
  - *Deploying to the Device*
  - *Resetting the Server*
- *Modifying an ODK 2 application*
  - *Modifying the data entry form*
  - *Updating the Initialization Files*
  - *Updating the Preloaded Data*
  - *Updating the HTML files*
- *Next Steps*

4.3.1 Prerequisites

This guide continues the tour were *Getting Started User Guide* left off. If you haven't yet completed that tour, do it first. When you have concluded the tour of the *Geotagger* example application's screens, return to this guide and we will turn to setting up our own ODK Aggregate server, and setting the application up to run on that server.

4.3.2 Migrating / Setting-up an ODK 2 application

Now that we have seen how a device can join an already-configured application, and synchronize its view of the data with the ODK Aggregate server hosting the application, it is time to set up our own ODK Aggregate server.

The starting point for this is to have a fully configured application on your device. Only after you have your device configured as you want it to appear, would you proceed with the following steps. In this case, we already have the device configured with the *Geotagger* demo, so let's proceed to create an ODK Aggregate server and configure it to serve that demo to your devices.

Setting up the ODK Aggregate server

Follow the instructions for Installing Aggregate. You must install the **ODK Aggregate v1.4.15** release. This is because we are transitioning away from Aggregate and towards *ODK Sync Endpoint*, but v1.4.15 will suit the purposes of this demo fine.

Once you have installed ODK Aggregate, log in with your super-user account. That process is also covered in Installing Aggregate.

Once logged in, enable the *ODK Aggregate Tables Extension*. You should grant the user account on your device the *Administer Tables* permissions.

Resetting the Application on the Server

Resetting the application on the ODK Aggregate server will push the application configuration on your device up to your server, replacing the configuration that is already on your server. Once the configuration is updated, data tables on the server and device will be synced. This process does not destroy data on the server, but instead merges changes on the client with any existing data tables on the server (this enables you to update your configuration without worrying about damaging or destroying the data already captured on the server).

Return to your device, start ODK Tables:

1. Click the diminishing-lines icon to leave the custom home screen.
2. Click the three vertical dots and select *Sync* to launch ODK Services onto the sync screen.
3. Choose *Settings* → *Server Settings*.
4. Edit the *Server URL* to be the URL for this newly configured ODK Aggregate server (https://myodk-test.appspot.com).
5. Click on *Server Sign-on Credential* and choose *Username*.
6. Choose *Username* and enter the superuser username for your ODK Aggregate server
7. Choose *Server Password* and enter the ODK Aggregate server password for that superuser username.
8. Click the back button until you have returned to the sync screen.
9. Click on *Reset App Server* to push your device configuration up to your ODK Aggregate server

After this has completed, you have created your own server that replicates the configuration and contents of the https://opendatakit-simpledemo.appspot.com site. Congratulations!

Note: Any device with a user account with *Administer Tables* permissions can reset the app server. If you configure a device with a user account (or Anonymous user) with only the *Synchronize Tables* permissions, they will not be able to reset the app server and will only be able to sync and join into the existing ODK 2 application on this ODK Aggregate server.

4.3.3 Configuring your Device with an Application

Next, we will work through the steps to configure your device with an ODK 2 application (rather than downloading an existing application from a server).

This task begins with setting up the *ODK Application Designer* on your computer.

For the purposes of this tutorial, we have created a copy of the Application Designer that only contains the files for this *Geotagger* example (it is otherwise identical).

Setting up ODK Application Designer

Read the *Intro* and *Overview* sections to get a sense of the features and functionality of the ODK 2 Application Designer environment (we will install it below). Follow this guide to *Setting Up ODK Application Designer*.

Finally, follow this guide to *Launching the Application Designer*.

If successful, the **`cmd`** window (on Windows) should display some status messages. Below is a screen-shot of my **`cmd`** window beginning with a **`dir`** of the contents of the directory, and running **`grunt`** in that directory:

And a **Chrome** browser window should open to display:

If a **Chrome** browser does not open, try manually launching it and opening http://localhost:8000/index.html.

You can further verify that the Application Designer works by clicking on the *Geotagger* button, then clicking on *Follow link*. This opens the *Geotagger* form on your computer, and simulates all the features available to you on your device.

You can also try other things, like choosing different device dimensions to see how the form renders on different screen geometries.

We will return to this design environment later.

Deploying to the Device

Now that we have the design environment installed and functioning, and because that environment has a copy of the fully-configured *Geotagger* application that is running on https://opendatakit-simpledemo.appspot.com (minus any data that users have submitted to the server), we can work through the steps of deploying that application to your device, and then resetting your server to push that configuration up to your server.

First, confirm that your device has *USB debugging* enabled inside your device's *Settings*. This checkbox is in different places on different devices and may be hidden by default on some. See this guide to USB debuggin on Android for instructions.

Return to the **cmd** window on your computer. `Control-C` to stop the **grunt** command that popped-open the **Chrome** browser. On Windows, you will be asked to confirm this `Terminate batch job (Y/N)?`. Enter `Y` to confirm.

Connect your device to your computer via USB. Wait for the storage connection to be established (on Windows, this will generally pop up a file browser or an options box that enables you to select a file browser).

At the command prompt, type:

```
$ grunt adbpush
```

> **Warning:** This command will force-close ODK Services, Survey, and Tables, and it will clear all ODK 2 data from the device. The data you are pushing will overwrite any exiting application or collected data you might have. Be sure to make backups and be sure you are ready before running this command.

This pushes the configured ODK 2 application within this ODK 2 Application Designer directory to your device. Because this is a stripped-down version of the Application Designer that only contains the simple demo files, this will copy only those files to the device. When you issue this command, the **cmd** window will display a long series of commands and conclude with a display of overall progress and timings:

Now, on your device, launch ODK Tables.

This will initiate the configuration of ODK Tables and conclude with a *Configuration Summary* pop-up reporting that everything was imported successfully. Click *OK*.

Everything should now appear as it did with the application you first joined on https://opendatakit-simpledemo.appspot.com, except you will only have the data rows configured by the ODK 2 Application Designer zip, and not any added or modified since that time.

Resetting the Server

Once you have the application running on the device, you will typically need to reset the contents of the application server. While the *Reset App Server* button on the device can shuffle the various supporting files between the device and the server, it will not destroy data tables that already exist on the server. This is intentional – we want to minimize the potential for accidental loss of data.

Note: Whenever you are developing an application, and have found a need to add a new column to an existing table, you will need to manually delete the data tables from the server before using the *Reset App Server* button from the device.

Open a browser window to the server, log in with a user that has *Administer Tables* or *Site*

Admin privileges.

Navigate to the *ODK Tables / Current Tables* sub-tab.

Delete each of the tables here. In this case, there will be only one, *Geotagger*. The server will now have a set of App-Level files but no data tables, forms for those tables, or data files. Except for the app-level files, it is clean.

Note: If your table has a large number of configuration files or data rows, the server may time out during the deletion process. In this case, the next time you try to create the table on the server, it will resume the deletion process, and potentially time out again until such time as it is able to finish the deletion. Only then will it re-create the table.

Now, from your device, launch ODK Tables, click on the sync icon (two curved arrows) to launch ODK Services, make sure you are logging in as a user with *Administer Tables* or *Site Admin* privileges, and choose *Reset App Server*.

The synchronization process will create the tables and push your content up to this server. Note that the server now only contains the data rows present on the device – it no longer has any of the additional data records from the demo site.

You have now successfully set up the Application Designer, used it to deploy an application to a device, and, from that device, configured an ODK Aggregate server to supply that application to other devices you join to that server.

4.3.4 Modifying an ODK 2 application

The final task is to modify the *Geotagger* example by adding a new data field to it.

The overall development process is:

1. *Revise the data entry form*
2. *Update the initialization files needed by ODK Tables*
3. *Update the preloaded data values as needed*
4. *Update the HTML to include the new field*

And then follow the steps in the preceding section to deploy the modified application to the device and push the application up to an ODK Aggregate server.

Modifying the data entry form

Return to your **`cmd`** window and once again launch the ODK 2 Application Designer environment (and a **`Chrome`** browser) by typing:

```
$ grunt
```

Now, open a file browser and navigate to the directory where you downloaded the Application Designer. Then navigate within that directory to `app/config/tables/geotagger`. Rename the `properties.csv` and `definition.csv` files in this directory to `orig.properties.csv` and `orig.definition.csv`. These were the initialization files needed by ODK Tables and they will need to be regenerated because we are altering the data table to incorporate an additional question.

Navigate within that directory to `app/config/tables/geotagger/forms/geotagger`. Open the `geotagger.xlsx` file in **`Excel`** (or **`OpenOffice`**). This is the form definition used by ODK Survey.

We will be adding a question to ask the user what direction they were facing when they took the photo. For this example, we will be collecting a text response. A more realistic modification might restrict the user to a set of choices (North, Northwest, West, Southwest, South, and so on).

On the survey worksheet, after the image-capture prompt, add a row that looks like the following.

Table 4.2: New Survey Row

type	name	display.text	display.hint
string	Direction	Image Direction	Enter the direction in which the photo was taken (North, South, East, West, and so on)

Save your changes and go back to the Application Designer. Click on the tab that says *XLSX Converter*. Choose this XLSX file or use your file browser to drag and drop the `geotagger.xlsx` file onto this screen (dragging and dropping is not supported on all operating systems).

You should now see some JSON in the output window. Hit the *Save to File System* button. This will display three pop-up notifications announcing that the Application Designer is

1. Writing the updated ODK Survey form definition into the `formDef.json` file in the same location as the `geotagger.xlsx` file.
2. Updating the `definition.csv` file.
3. Updating the `properties.csv` file.

Note: The `definition.csv` and `properties.csv` files are updated because the *form_id* is the same as the *table_id*.

Go back to the **`Chrome`** Browser and click on the *Preview* tab. Click on *Purge Database*. This will delete the earlier *Geotagger* data table – a necessary step because we are adding a

Direction column to that data table. Select *Geotagger* if you do not already have that form open.

Create a new instance of *Geotagger* and advance through it (this will create the data table with the new Direction column). Confirm that the new question is displayed. Note that the date and description are required fields and will generate error pop-ups if you attempt to advance through those prompts without supplying a value.

You have now successfully modified the form.

Updating the Initialization Files

Fortunately, because the geotagger *formId* matches the *tableId*, by using the *Save to File System* button on the CSV, the tool will automatically regenerate the `definition.csv` and `properties.csv` files for this form. Furthermore, the configuration that ODK Tables uses to specify what HTML files to use for the list, detail, and map views are all specified within the XLSX file on the properties sheet. No manual actions are required!

Now, deploy your updated application to your device. Launch ODK Tables to initialize and load your application. Confirm that when you edit a data row that you are now asked for the direction in which the photo was taken.

Updating the Preloaded Data

At this point, we have added the new field to the data table, but have not yet updated the initial set of *Geotagger* locations with values for that field.

Return to your `Application Designer` directory. Recall that when an ODK Tables application first starts up, it reads the `assets/tables.init` file. That file identifies CSV files within `config/assets/csv` that should be imported into the data tables upon first start-up. Read more about importing data into a table from a CSV in the *ODK Tables guide*.

In this example application, the file being imported is `config/assets/csv/geotagger.updated.csv`. If we wanted to, we could edit this file, add a column for the new data field (Direction), and supply values for this field for all of the data rows that form the initial set of *Geotagger* locations.

Alternatively, we can return to the device and use the CSV export functionality within ODK Tables to export the CSV file (into `/sdcard/opendatakit/default/output/csv`). Then pull it off the device and overwrite the CSV file under the Application Designer at `app/config/assets/csv/geotagger.updated.csv`. Finally, open that file and fill in values for the Direction column.

Warning: Some CSV editors, like `Office` or `OpenOffice`, may convert or alter the content inappropriately when you save changes. If your edits cause the device to fail to initialize the data fields, you may need to make this edit manually using a less-sophisticated tool or choose different options when saving your changes.

Updating the HTML files

There are two areas where image information is displayed, one is in the list view, where you can expand or collapse an item, and the other is in the detail view, which is opened when you click or tap on an expanded item in the list view. We will only modify this detail view to report the image direction. A more comprehensive edit would likely also update the expanded item within the list view.

To determine all the HTML files, we can begin with the files referenced in the `properties.csv` file we recently finished editing. Looking again at that file, we see three files referenced:

- `tables/geotagger/html/geo_list.html`
- `tables/geotagger/html/geo_list_thumbnail.html`
- `tables/geotagger/html/geo_detail.html`

Each of these files, or the JavaScript within them, might open or reference other files that might need to be updated. The above files are simply the ones we know are reachable. In general, files for displaying table-specific data are under the `config/tables/tableid` directory. In this example, we will modify the last of these files and its associated JavaScript file.

Open a file browser and navigate to the directory where you downloaded the Application Designer. Then navigate within that directory to `app/config/tables/geotagger/html`. Open `geo_detail.html` in a text editor. Insert a line that defines a *DIR* element above the *Latitude* line in the HTML body region. This will be where we will display the value of the *Direction* field. For example:

```
<h1><span id="TITLE"></span></h1>
<p>Image Direction: <span id="DIR"></span></p>
<p>Latitude: <span id="FIELD_1"></span></p>
```

Save the file. Now, navigate to `app/config/tables/geotagger/js`. Open `geo_detail.js` in a text editor. Navigate down to the bottom of the `display()` JavaScript function (to line 44). And add before the closing bracket:

```
var dir = geoDetailResultSet.get("Direction");
document.getElementById("DIR").innerHTML = dir;
```

Save the file. Once again, push the application to the device. Confirm that when you expand a item in the map list window, and then tap on that expanded item, that it now shows *Image Direction:*. (See example below.)

Congratulations, you have successfully modified this ODK 2 application to add a new data field and display it as a field in the HTML detail-view page.

You could now log onto your server, delete the geotagger table, reset your server, and start collecting geopoints with the new image direction field.

4.3.5 Next Steps

Survey and Tables each have a basic sample application that walks through their features:

- *ODK Survey: Sample Application*
- *ODK Tables: Sample Application*

To get started building applications, first set up the *ODK Application Designer*. After you have familiarized yourself with that tool, you can try building and deploying an application:

- *Building an Application*

A more complete guide to using ODK XLSX Converter is provided in the *ODK XLSX Converter* documentation. More details about Tables web views are available in *ODK Tables Web Pages* and *Injected JavaScript Interfaces*.

For examples of real world applications and details about they are implemented, try out the: *Reference Applications*.

We also provide guides geared towards Deployment Architects for each of the Android and Desktop tools.

- *Managing ODK Survey*
- *Managing ODK Tables*
- *Managing ODK Services*
- *Managing ODK Scan*

However the user guides for these tools are also useful for everyone.

Finally, to expand your knowledge of the more advanced features of the platform, such as data permission filters, read the *Deployment Architect Advanced Topics*.

4.4 ODK Survey

ODK Survey is an Android application for performing data collection in the ODK 2 framework. It operates similarly to ODK Collect, but is based on HTML, CSS, and Javascript rather than native Android, and is more flexible in its presentation and execution.

Note: ODK Survey only works on Android 4.2 and newer devices.

Note: ODK Survey cannot read or display the forms created for ODK Collect (that is, those created via ODK Build, XLSForm, or other form design tools). ODK Survey operates with ODK 2 Data Management Applications*.

4.4.1 User Guide

Installing ODK Survey

Prerequisites

Required

Before installing ODK Survey, you will need the following ODK Tools:

- *ODK Services*

As well as the following third party apps:

- OI File Manager

Recommended

We also recommend installing:

- *ODK Tables*

ODK Tables is not required, but Tables and Survey are most versatile when used together. Tables offers a way to visualize, process, and modify data collected by Survey, all on the device.

Installing the ODK Survey App

1. From your device's *Settings*, choose *Security*.

- Make sure *Unknown Sources* is checked.
- (On older versions of Android, this setting is in *Applications* rather than *Security*)

2. Open a web browser on your phone.
3. Navigate to https://github.com/opendatakit/survey/releases/latest and download the latest ODK Survey APK.
4. In the download window, you will see ODK_Survey.N.N.apk. - Select it to download the file.

- On older devices, the APK will automatically install after you approve the security settings.
- On newer devices, you must go to the download list, rename the file to restore the .apk extension (the extension will have been renamed to .man during the download process), then click on it to install it.

Note: You can also download the ODK Survey APK to your computer and load it on your device via adb or another tool like AirDroid.

Tip: You can also install ODK Survey on an Android emulator. However, this can be slow and is only recommended for developers actively working on Survey.

ODK Survey: Sample Application

In this guide we will be demonstrating how to use ODK Survey via a guided tour of a sample application. This guide demonstrates both the general workflow of Survey and some of the features that differentiate it from ODK Collect.

> **Warning:** ODK Survey performs a similar role to ODK Collect in the ODK 2 Tool Suite. However, it is more complex and not every organization will need its features. The choice of whether to use ODK Collect or ODK Survey and the ODK 2 tools should be made carefully based on your organization's needs. A brief comparison can be found in the guide for *Selecting the Appropriate Tool Suite*.

Prerequisites

Install ODK Survey and its prerequisites from the guide: *Installing ODK Survey*.

Sample Application Overview

We have provided a sample application to help you acquaint yourself with the various features of ODK Survey. This sample app contains six sample forms within it:

- **Example Form** – a form with many examples of data entry widgets.
- **Grid Screen Form** – a form used to demonstrate a new screen layout that allows fully customized prompt placement.
- **Household Survey** – a form used to gather information about a household. To operate correctly, this requires the *Household Member Survey* sub-form and the *Education* sub-form (you should not open those sub-forms directly – they are launched from within Household Survey).
- **Select Examples** – a form with several examples of select widgets, including widgets that access data on Yahoo servers, and others that access CSV files for their choice lists. It also demonstrates the use of custom CSS styles to change the look of the form.
- **Household Member Survey** – a form used to gather information about household members. This is a sub-form of the Household Survey form (you should not open it directly – it is launched from within Household Survey). ODK Survey eliminates the repeat group concept and replaces it with sub-forms. From within the Household Survey you navigate into this sub-form by entering information about individuals in a household.
- **Education** – a form used to gather education information about household members. This is another sub-form of the Household Survey form (you should not open it directly

– it is launched from within Household Survey). This sub-form saves information to the same underlying data table (household_members) as the Household Member Survey form, but it asks different questions. This demonstrates the use of multiple forms to revise different sets of values within a data table. From within the Household Survey you navigate into this form when you enter education information about individuals in a household.

Note: Since the *Education* and *Household Member Survey* operate on the same table, you will only see five tables in ODK Tables and in the Cloud Endpoint even though there are six forms.

Learn More

For instructions on creating your own Survey applications, view the *ODK Survey: Designing a Form* guide.

Installing the Sample Application

Unlike ODK Collect, the ODK 2 tools are application-focused. An application is identified by the name of the directory under the `/sdcard/opendatakit/` folder. The sample application is named *default*, as is the sample applications provided for *ODK Tables*. This means that you can only deploy one of these sample application at a time onto a device. We also provide *instructions* to rename one of these so that two or more applications can co-exist and not interfere with each other on this same device.

To access the sample application and its six sample forms:

1. Launch ODK Survey. Press the *Settings* button that resembles a gear. This will launch the ODK Services tool to the settings page.

2. Follow the *Server Configuration* instructions to set up your server.

- Set your *Server URL* to `https://opendatakit-2.appspot.com`.

Note: The server URL starts with `https://` not `http://`. Don't forget to include the *s*.

- Leave your authentication as *None (anonymous access)*.

3. Back out until you return to Survey.
4. Follow the *Syncing* instructions (see *launching from Survey*).

- Again, leave your user as *None (anonymous access)*.
- Leave the file attachment setting to *Fully Sync Attachments*

After synchronization is complete, your device's configuration will exactly match that of the server. This includes both collected data and application level files (such as form definitions and HTML files). If you had nothing on your device before, your device will be populated with this data and these application files. If you already had files on this device in this application namespace they will be updated to match the server version. Any local configuration files for data tables or forms that are not present on the server will be removed from your device. Everything under the `/sdcard/opendatakit/default/config` directory will be revised to exactly match the content on the server.

Once the configuration and data on the device is an exact match to that of the server, the file attachments associated with those data are synchronized. If you have a slow connection, it may take two or three tries before the sync is successful. This will not overwrite or hurt anything to do multiple synchronizations in a row.

When complete, click *OK* on the *Sync Outcome* dialog and back out of the Services, returning to Survey.

If the sync was successful, ODK Survey will scan through the downloaded configuration, updating its list of available forms.

When that is completed you should now be presented with the list of those six sample forms.

Learn More

For instructions on installing your own Survey application to a device, view the *Moving Files To The Device* guide.

Opening a Form

Open Survey. If you have successfully installed the sample application, you should be presented with a list of the six sample forms.

Note: The *Household Members* and *Education* forms are not intended to be called directly, but are launched from within the *Household* form.

To open a form, tap on it in this list. For this tutorial, open the *Example Form*.

This screen shows the name and version of the form you are viewing. If you scroll down you will see a list of previously created instances of this form.

Tip: Form instances can always be edited, even after they have been finalized.

To fill in a new instance, tap the *Create new instance* button.

Learn More

For more detailed instructions on opening and modifying Survey form instances, view the *Opening a Form* guide.

Navigating a Form

Forms in Survey are defined in HTML, CSS, and JavaScript. A default look-and-feel, along with an extensive selection of prompt widgets, is provided by the ODK 2 framework, but this can be customized by your organization.

To navigate forms using the default look-and-feel:

- Tap on the name of the survey in the top left to access a pop-up menu of options.

- Tap the *Back* or *Next* buttons in the top right of the form to navigate through the form.

Let's fill out the instance of the *Example Form* that we opened in the previous section. After tapping the *Create new instance* button you should see the following screen:

Use the *Next* button in the top right to progress to the first question.

Initial Value

This prompt asks you to give the form an initial rating. Its purpose in this example is to show how Survey can use previously collected data to populate and calculate later fields. Enter any number you like and it will be used later.

Press the *Next* button in the top right to progress to the next question.

Prompt Selection

This prompt allows you to choose which sections of the form to complete. Survey form navigation can be completely customized, even at runtime, to include or exclude sections, repeat portions, jump directly to different prompts based on entered values, and more. For this example we will complete the *label features*, *computed assignment of initial values*, and *custom template* sections. However, feel free to enter any combination you like and explore.

Press the *Next* button in the top right to progress to the next question. Note that we skip the *intent launching* section and progress directly to *label features*.

Label Features

This prompt shows that the label and hint fields of the prompt can be customized by editing the HTML and CSS. This allows your organization to modify the look-and-feel of the prompts to suit their needs.

Press *Next* to see a more complex example:

This prompt shows a label that has been edited to include media files including an image and an audio clip. Press play on the audio clip to hear a bird call. However, media can also be added via spreadsheet columns, which is generally easier.

Press *Next* to advance to the next section.

Reading Previous Values

This prompt is requesting a value that will be used to render the next question. Enter any name you like and press *Next*.

This prompt shows that a prompt can use a previously collected value in the rendering of a prompt. For example, a subject's name and gender could be used to properly address them throughout a survey.

Press *Next* to see another example of data reuse.

This prompt is requesting a value that will be used to render the next question. Enter any value you like and press *Next*.

This prompt will prepopulate the entered data with the value from the previous prompt. In general, you can prepopulate the a prompt with any previously collected value. In another example you might record a subject's address and then prepopulate that address on their household members address prompts.

Press *Next* to advance to the next section.

Custom Template

This prompt is requesting data that will be used in the next prompt to render a custom template. We will also use this to demonstrate constraints. Enter an age that is greater than 20 and press *Next*.

Survey will not allow you to progress until you've entered a valid value. This validation can be done dynamically as well. For example, you could have a running average of crop heights you have measured, and disallow crop heights that differ by more than three standard deviations.

Enter a valid age, weight, and height, and press *Next*.

This prompt will show the data point you entered in the previous prompt, rendered on a plot of average weights. This is a custom prompt defined in JavaScript for this example, it is not a default display option provided by the ODK 2 framework. It demonstrates that Survey can be customized to whatever level your organization requires without the effort of rewriting and recompiling the Android tools.

Press *Next* to advance to the next section.

Update Value

This prompt is prepopulated from the initial value we entered in the first prompt. Whatever you entered for that field will be filled in here. Updating this field will update the value in the database.

This was the final prompt for this example. Press *Next* to advance to the final screen of the form.

Complete Form Instance

This screen tells you that you have reached the end of the form. This **does not** mean that you have entered data for every field. In this example we skipped the majority of the questions. From here you can navigate backwards and update any of your previous answers. You can also use the button in the upper left to navigate to previous questions or leave the form instance.

> **Warning:** Updating answers may cause later prompts to render differently or be invalidated.

To save the form instance, either press *Finalize* or *Incomplete*.

- *Finalize* will mark the form as *Finalized* and indicate that this instance is completed.
- *Incomplete* will mark the form as *Incomplete* and indicate that this form should be revisited and completed in the future. Use this option to save your progress if you have to pause while filling out a form.

After pressing one of the above options you will be returned to the Survey home-screen. If you select *Example Form* again you will see this form instance at the top of the list of previously saved instances, with the date you saved it and the state you chose.

Learn More

For more detailed instructions on navigating Survey forms, view the *Navigating a Form* guide.

Explore the Sample Application

This concludes the guided tour of the sample application for Survey. However, this is far from a complete reference. Please continue to explore the different forms and prompts to learn more about the tool's capabilities.

You can find a more detailed user guide for Survey here: *Using ODK Survey*. And you can find a more detailed guide to managing Survey for Deployment Architects here: *Managing ODK Survey*. You can also find the sample forms shown in this tutorial in the Github repository for App Designer.

Using ODK Survey

Warning: Survey forms are defined in HTML, CSS, and JavaScript that can be edited by your organization. If the interfaces displayed in this guide do not match the form you have on your device, contact an administrator in your organization for further guidance.

- *Opening a Form*
  - *Opening a Subform*
- *Saving a Form Instance*
- *Viewing Saved Form Instances*
  - *Editing Saved Form Instances*
  - *Deleting Saved Form Instances*
- *Navigating a Form*
- *Syncing Forms and Data*

Opening a Form

The home screen of Survey displays a list of all the forms available on the device.

Note: Not all forms listed on this home screen are intended to be launched directly. Subforms will be listed on this page but are generally intended to be opened from within a parent form.

To open a form tap its name on the list. This will launch the home screen of that particular form. Below we have opened the *Example Form*.

This screen shows the form name and version. It also shows a full list of saved instances of the form. For more information on viewing and editing these form instances see the section on *Viewing Saved Form Instances*.

To start a new instance and begin filling in a form, tap the *Create new instance* button.

Opening a Subform

Unlike their parents, subforms are generally not intended to be opened from the Survey home screen's form list. Instead, subforms are integrated into their parent forms and launched directly as prompts. They integrate seamlessly into their parent forms and do not need to be manually opened. They might be indicated by a *Create new instance* button within a form, or the form may directly launch the subform.

For example, the *Household Member Survey* is a subform of the *Household Survey* in the *Survey sample app*.

This screen within the *Household Survey* shows the launcher for the *Household Member Survey* subform. Clicking *Create new instance* will launch the subform.

This is the first page of the *Household Member Survey* subform. It displays the name of the household, **Sample House**, which was collected in its parent *Household Survey* form. After this subform has been filled in, the flow will return to the parent form.

Completing the *Household Member Survey* subform returns us to the same screen we launched from in the *Household Survey*. The instance created by the subform is now displayed. If you tap the *Create new instance* button again, you can create multiple instances.

Saving a Form Instance

When saving a form instance, you can either mark it as **Finalized** or **Incomplete**.

- **Finalized** forms indicate that they are completed and that the data should be used and aggregated.
- **Incomplete** forms indicate that the form has been saved but it is not yet complete. This is useful if you need to stop filling out a form and return to it later, but want to keep your previously collected values.

Note: Marking a form as **Finalized** does not prevent you or another user from modifying it later.

There are three ways to save a form:

1. Navigate to the end of the form. This screen will show the buttons to save the form as *Finalize* or *Incomplete*, as described above. After choosing one of these options, Survey will return to its home screen.

2. Tap the button with the name of the form in the upper left from any screen in the form. This will open a menu that provides navigation and exit options.
   - To save the form as **Incomplete** choose *Save Change + Exit*
   - To save the form as **Finalized** choose *Finalize Changes + Exit*

3. Press the Android back button. This is not the *Back* button provided by ODK in the upper right. This is the button to back out of apps. This will launch a menu with the option to *Save Changes* which will save the form as **Incomplete**.

Note: This menu does not have an option to save a form as **Finalized**.

Viewing Saved Form Instances

A list of previously saved form instances is viewable on the home screen of each form. Open the desired form (instructions in the *Opening a Form* guide) to see this list.

Warning: This list of saved form instances is not limited to those collected on your device. After synchronization this will include all form instances from all devices that have synced with the server. Take care not to edit form instances that you should not be editing.

To protect against unauthorized edits, see *Data Permission Filters*.

This list of instances is ordered reverse chronologically by the last save date, with the most recently edited form instance on top and the oldest form instance at the bottom. These instances are marked as either **Finalized** or **Incomplete** (see *Saving a Form Instance* for definitions).

Editing Saved Form Instances

To edit a form instance, tap the pencil icon next to the instance in the instance list on the form home screen.

This will launch that instance with all collected values prepopulated. When you save this form as either **Finalized** or **Incomplete**, that state will overwrite the previous state of **Finalized** or **Incomplete**. The updated form instance will now be the most recently edited form and appear at the top of the list.

Deleting Saved Form Instances

To delete a form instance, tap the *X* icon next to the instance in the instance list on the form home screen.

Navigating a Form

Forms in Survey are defined in HTML, CSS, and JavaScript. A default look-and-feel, along with an extensive selection of prompt widgets, is provided by the ODK 2 framework, but this can be customized by your organization. This guide assumes you are using the default look-and-feel.

- To advance to the next prompt in a form, press the *Next* button in the upper right.

- To go backward to the previous prompt, press the *Back* button in the upper right.

- To navigate to a specific prompt, press the button on the upper left with the form name to show the menu. Tap the button labeled *Contents*.

This will bring up a menu with a full list of fields and their recorded values. Tap the desired field to navigate to it in the form.

Every change you make to the data in the form is written immediately to the database as a **checkpoint** save.

Syncing Forms and Data

See the instructions in the *ODK Services user guide*.

> **Warning:** If a data table has any checkpoint saves (for example, caused by form crashes), the data table will not be synchronized. Checkpoints must be resolved before sync can proceed. The user must open a form on the problem table and either delete the checkpoint or edit the checkpoint. If editing, after that is complete they must save is as either incomplete or finalized. Once the checkpoints are eliminated, the user can initiate another synchronization, and the data in this table will then be synchronized with the information on the server.

4.4.2 Deployment Architect Guide

Managing ODK Survey

- *Prerequisites*
  - *Required*
  - *Recommended*
- *Setting up a Form Development Environment*
- *Designing a Form*
  - *Full XLSX Reference*
- *Launching With a Different AppName*
  - *Android 4.x Devices*
  - *Android 5.x and Higher Devices:*
  - *Trying the New Launcher*
  - *Making a New AppName*

Prerequisites

Required

To create an Data Management Application that uses ODK Survey, you will need the ODK tools:

- *ODK Services*
- *ODK Application Designer*
- *ODK Cloud Endpoints*

As well as the third party apps:

- OI File Manager

If you have not installed Survey already, follow our guide for *Installing ODK Survey*

Recommended

We also recommend:

- *ODK Tables*

ODK Tables is not required, but Tables and Survey are built to seamlessly integrate and support more robust Data Management Applications.

Setting up a Form Development Environment

To get started creating your own Data Management Applications, go to the *ODK Application Designer* documentation.

Designing a Form

Basic instructions for designing Survey forms are provided in the *ODK Survey: Designing a Form*.

Survey forms are created in **`Excel`** and saved as `.xlsx` files. These are converted into form definitions using the *ODK XLSX Converter*. The linked guide should help you create and modify the files to create your own forms.

Full XLSX Reference

Use the *ODK XLSX Converter Reference* to find all the features you can use in your Survey forms.

Launching With a Different AppName

The ODK 2 tools are designed to support multiple independent Data Management Applications running on the Android device. Each of our tools has the ability to run in the context of either a default application name, or a specified application name.

By default, ODK Survey runs under the *default* application name (as does ODK Tables and the other ODK 2 tools). Application names correspond to the name of the directory under `/sdcard/opendatakit` where the configuration and data files for that application are stored.

> **Warning:** Though the Android tools support multiple AppNames on the device, each *ODK Cloud Endpoints* only supports one AppName at a time. For each application you have running on the device, you will need a new Cloud Endpoint that is configured with that AppName.
>
> Each Data Management Application will store its own server configuration. Therefore after an initial setup that points each application at its proper server, the user will not need to remember which server hosts which app.

Here we describe how to launch the ODK 2 tools into an application name of your choice with the use of widget shortcuts.

First, you must create an alternative application. As a contrived example, we will make an exact copy of the *default* application on the device with a new name. To do so, first load an application to the device (such as the *sample application*). Then open **`OI File Manager`**, navigate to the `/sdcard/opendatakit` directory, and copy the *default* directory, renaming it *myapp*. You have now created the *myapp* application! It is isolated from and operates independently of the default application.

To launch and use that application:

Android 4.x Devices

1. Choose to view the installed applications.
2. Select the *Widgets* tab at the top of that screen.
3. Navigate through the available widgets, and select and hold the *ODK Survey Form* widget. Drag and drop it onto one of your Android launcher (home) screens.
4. A list of available applications and forms will appear, in the form of application name for applications, and *application name* → *form name* for each form within an application. Pick the *myapp* application that you created via **`OI File Manager`**.

Android 5.x and Higher Devices:

1. Long press an open area of the device home screen
2. Select the *Widgets* tab at the bottom of resulting screen.
3. Navigate through the available widgets, and select and hold the *ODK Survey Form* widget. Drag and drop it onto one of your Android launcher (home) screens.
4. A list of available applications and forms will appear, in the form of application name for applications, and *application name* → *form name* for each form within an application. Pick the *myapp* application that you created via **`OI File Manager`**.

Trying the New Launcher

Now, play around with launching ODK Survey using this application shortcut and *Finalizing* a new filled-in form. Exit ODK Survey, and launch it from the applications list (so that it launches as the default application). Verify that you do not see that newly filled-in form. You can also create a new filled-in form in this default application and confirm that it is not visible in the myapp application.

This highlights the isolation of Data Management Applications in the ODK 2 tools. This is even more powerful with applications that use ODK Tables because you can create entirely

isolated applications, such as a forestry app and a health clinic app, and have the forms and data entirely independent of each other.

This eliminates the need for different groups to fork the ODK code base.

Making a New AppName

1. Download a new copy of *ODK Application Designer*. Clear out the `config` directory as you normally would.
2. Open `app-designer/gruntfile.js`.
3. In the `modle.exports` function, find the variable `tablesConfig`.
4. Modify the value of `appName` in variable `tablesConfig`. This value starts as *default*. Set it to the desired new AppName.

   Note: The new AppName cannot be the same as another AppName that already exists on the device.

5. Save `Gruntfile.js`
6. Develop your Data Management Application and push it to the device the normal way (instructions in the *guide*).

Using the above technique will keep your apps cleanly separated. You can also maintain multiple Data Management Applications in the same Application Designer instance by making alternative `app-designer/app` directories and creating new **`Grunt`** tasks to push them to the device.

4.5 ODK Tables

ODK Tables is a program that allows you to visualize and update existing data. Using Tables as your entry-point to data collection, you will be able to gather data using ODK Survey, sync it to a server using ODK Services, and have other users download and edit this same data on their own devices.

Tables also enables web developers to build powerful *data management applications* to handle their complex workflows. While Survey follows a traditional data collection workflow, similar to that of Collect, Tables gives you the flexibility to implement your own arbitrary complex workflow. For example you might collect data via a customized mapping interface: Tables allows you to build an application using web technologies to achieve that.

Note: ODK Tables only works on Android 4.2 and newer devices.

We have included a sample application built on top of Tables along with a handful of data tables that showcase some of its features.

4.5.1 User Guide

Installing ODK Tables

Prerequisites

Required

Before installing ODK Tables, you will need the following ODK Tools:

- *ODK Services*

As well as the following third party apps:

- OI File Manager

Recommended

We also recommend installing:

- *ODK Survey*

ODK Survey is not required, but Tables and Survey are most versatile when used together. Survey offers a simple, form based data collected workflow similar to ODK Collect that can be seamlessly integrated with Tables to create and modify records.

Installing the ODK Tables App

1. From your device's *Settings*, choose *Security*.
   - Make sure *Unknown Sources* is checked.
   - (On older versions of Android, this setting is in *Applications* rather than *Security*)
2. Open a web browser on your phone.
3. Navigate to https://github.com/opendatakit/tables/releases/latest and download the latest ODK Tables APK.

4. In the download window, you will see ODK_Tables.N.N.apk. - Select it to download the file.

- On older devices, the APK will automatically install after you approve the security settings.
- On newer devices, you must go to the download list, rename the file to restore the .apk extension (the extension will have been renamed to .man during the download process), then click on it to install it.

Note: You can also download the ODK Tables APK to your computer and load it on your device via adb or another tool like AirDroid.

Tip: You can also install ODK Tables on an Android emulator. However, this can be slow and is only recommended for developers actively working on Tables.

ODK Tables: Sample Application

In this guide we will be demonstrating how to use ODK Tables via a guided tour of a sample application.

Prerequisites

Install ODK Tables and its prerequisites from the guide *Installing ODK Tables*.

Sample Application Overview

We have provided a sample application to help you acquaint yourself with the various features. This sample app contains five demo apps within it.

- **Tea Houses** - a fictional Benin Teahouse directory. It provides a broad overall view of the different view types that Tables offers.
- **Hope Study** - a simplified subset of a perinatal follow-up application that was piloted on ODK Tables and ODK Collect (now converted to use ODK Survey). It demonstrates how Tables and Survey can be integrated.
- **Plot** - a fictional agricultural field pest- and yield- assessment application. It demonstrates how data can be visualized and actively updated as it is collected on the device.
- **Geotagger** - a simple geotagging application. It demonstrates a basic customization of the user interface with HTML, CSS, and JavaScript.

- **JGI** - an app used to track the daily behavior of chimpanzees. It uses a complex, fully customized and domain specific user interface. It also demonstrates Tables collecting data for multiple data tables and rows simultaneously (as opposed to the single row editing of Survey).

Install the Sample Application

Unlike ODK Collect, the ODK 2 tools are application-focused. An application is identified by the name of the directory under the `/sdcard/opendatakit/` folder. The sample application is named *default*, as is the sample applications provided for *ODK Survey*. This means that you can only deploy one of these sample application at a time onto a device. We also provide *instructions* to rename one of these so that two or more applications can co-exist and not interfere with each other on this same device.

We will use the ODK 2 synchronization mechanism to install this app. It is about 26 MB in size and takes a few minutes to download from the web.

1. Launch ODK Tables. Press the Action Button ( ) and press *Preferences* from the menu.

2. Follow the *Server Configuration* instructions to set up your server.

- Set your *Server URL* to `https://opendatakit-tablesdemo.appspot.com`.

> **Note:** The server URL starts with `https://` not `http://`. Don't forget to include the *s*.

- Leave your authentication as *None (anonymous access)*.

3. Back out until you return to Tables.
4. Follow the *Syncing* instructions (see *launching from Tables*).

- Again, leave your user as *None (anonymous access)*.
- Leave the file attachment setting to *Fully Sync Attachments*

After synchronization is complete, your device's configuration will exactly match that of the server. This includes both collected data and application level files (such as form definitions and HTML files). If you had nothing on your device before, your device will be populated with this data and these application files. If you already had files on this device in this application namespace they will be updated to match the server version. Any local configuration files for data tables or forms that are not present on the server will be removed from your device. Everything under the `/sdcard/opendatakit/default/config` directory will be revised to exactly match the content on the server.

Once the configuration and data on the device is an exact match to that of the server, the file attachments associated with those data are synchronized. If you have a slow connection, it may take two or three tries before the sync is successful. This will not overwrite or hurt anything to do multiple synchronizations in a row.

When complete, click *OK* on the *Sync Outcome* dialog and back out of the Services, returning to Tables.

If the sync was successful, ODK Tables will scan through the downloaded configuration, updating its list of available forms.

After this configuration is set up, ODK Tables should now present a custom home screen with five tabs, one for each of the demos. If it does not, back out of ODK Tables and re-launch it.

Learn More

For instructions on installing your own Tables application to a device, view the *Moving Files To The Device* guide.

Custom Home Screen

Open Tables. If you have successfully installed the sample application, you should be presented with a custom home screen showing the five demo apps.

Tables allows your organization to customize the home screen of your Data Management Application. By default Tables will only show a list of the data tables defined on the device (called the *Table Manager*). But with a custom home screen your organization can implement their own complex workflow and look-and-feel with HTML, CSS, and JavaScript. An example of this is what is displayed after downloading the sample application.

Note: All of these screens and web pages are served directly from the device – there is no network access. These are fully able to function in Airplane mode – without a WiFi or internet connection.

When you design your applications, you can either have them operate without any network access, or you can write them to access data on the internet. This becomes your design choice.

Each tab on this screen is the home page for one of the five demo applications listed above.

Note: For this example we have included all five applications under the same AppName. However, typically you would give them each their own AppName to provide a clean separation of data.

Learn More

For more information about custom home screens, view the *Custom Home Screen* guide.

Tea Houses Demo

For this portion of the tutorial we will explore the *Tea Houses* demo. Select the tab labeled *Tea* and press *Launch Demo*.

The *Tea Houses* demo is a fictional collection of Tea Houses in Benin and the teas they offer.

Custom View

The first screen you will see after launching the *Tea Houses* demo is a custom view.

As with the custom home screen, this custom view is rendered entirely in HTML, CSS, and JavaScript defined within the *Tea Houses* demo. It does not collect or present data, it acts as a navigation screen to allow the user to choose which of the three data set to interact with.

Custom views are not limited to navigation and workflow interfaces. They can also be used to view data, create data visualizations, and modify data in the database. The *Plot Demo* and *JGI Demo* explore this more fully.

Press the button labeled *View Teas* to launch the *List View* of the available teas.

Learn More

For more information about custom views, view the *Custom View* guide.

List View

This screen shows a list of all teas available in the *Tea Inventory* data table. This view is customized with HTML, CSS, and JavaScript. It provides a simple way to view and navigate collected data. As new teas are added to the inventory, this list view will grow.

To see the raw data, we will switch to *Spreadsheet View*. Tap on the lined paper icon at the top of the screen. Here you'll see all the possible view types. Select *Spreadsheet*.

Learn More

For more information about *List Views*, view the *List View* guide.

Spreadsheet View

This view renders a full data table from the database, including all rows and columns. Unlike the views we have seen so far, this view is NOT customized via HTML, CSS, and JavaScript. This view is provided by the ODK 2 platform for convenience in viewing and editing your data directly. It is meant to be a familiar view as if you were looking at it on a spreadsheet program, such as **`Excel`**. Each row here represents a tea, and each was a row in the *List View*.

Return to the *List View* by using the lined paper icon as before and selecting *List*. Tap the *Stonehouse* tea to launch a *Detail View* for that tea.

Learn More

For more information about *Spreadsheet View*, view the *Spreadsheet View* guide.

Detail View

This screen shows all the details of the *Stonehouse* tea entry in the *Tea Inventory* table. The *Tea Inventory* table's *Detail View* displays information about the tea, including whether it is available hot, iced, in bags, or loose leaf. Note that the tea type is being pulled from the *Tea Types* table, but the JavaScript is getting the information from that table to construct our view. Like the other views, we programmed this using rudimentary HTML and JavaScript, but it could be customized to look fancier or display additional information.

Next we will see a combination of the detail and list view options. Back out until you hit the custom view with the three buttons. .. _tables-sample-app-detail-view-learn-more:

Learn More

For more information about *Detail Views*, view the *Detail View* guide.

Detail With Sublist View

From the custom view with the three buttons, select *View Tea Houses*. This will launch another *List View*, this time showing the list of tea houses.

The *Tea Houses* table has been configured to use a *Detail With Sublist View* rather than a *Detail View*. Tap the *Tea for All* tea house to see this.

This screen contains a *Detail View* webpage and a subordinate *List View*. In this case, the *Detail View* displays information on the tea house, and the *List View* displays the teas that the tea house serves. Within the *Detail View*, you can scroll down to see the information we decided to display. It is also written in HTML, CSS, and JavaScript to render these table entries. The look-and-feel is similar to the *Tea Inventory* only because that is how we coded it. Like the *List View*, we programmed this using very rudimentary HTML and JavaScript, but it could be customized to look fancier or display additional information.

Scroll to the bottom of the *Detail View* portion of the screen and you'll see a link as a number of teas. This is using the information in the table called *Tea Inventory* to tell you how many teas this tea house offers, and has also been defined in the JavaScript.

The bottom half of the screen renders the subordinate *List View*, which shows the list of teas available at the *Tea for All* teahouse. It is a separate page that is controlled by the top half.

Note: This is a simple example that has a static list. However, you could dynamically change the list that is rendered with controls in the JavaScript for the top half of the screen. For example, you could have a household detail on top, and list all family members on the bottom. You could then provide a button to change the list to only show adult family members in the list below.

Next we will see the *Map View*. Back out of the *Detail With Sublist View* to see the list of tea houses. Press the lined paper icon and choose *Map* from the menu.

Learn More

For more information about *Detail With Sublist Views*, view the *Detail With Sublist View* guide.

Map View

All the fictional tea houses in Benin appear on the map. Pinch and squeeze or widen to zoom out and in, respectively. The tea house location is plotted based on what appeared in the *Location_latitude* and *Location_longitude* columns in the database. These can be viewed with the *Spreadsheet View*. When you click on a map marker, the *List View* will redraw with that marker's information at the top of the *List View*.

The *List View* at the top portion of the screen is rendered in custom HTML, CSS, and JavaScript, but the map portion is provided by the ODK 2 platform and rendered using `Google Maps`.

Learn More

For more information about *Map Views*, view the *Map View* guide.

Edit Row With Survey

The final portion of the *Tea Houses* demo will be to edit data with Survey. Return to the *List View* by using the lined paper icon as before and selecting *List*. Tap the *Tea for All* tea

house to launch a *Detail With Sublist View* for that tea. Tap the pencil icon in the upper right.

This will launch Survey to edit the *Tea for All* row in the *Tea Houses* data table.

This Survey form allows you to edit any and all of the data fields in the *Tea for All* entry. Navigate to the question that reads:

> *Which tea is the house specialty*

Change the specialty to be Herbal. Complete the form and finalize the changes. When you return to the *Tea for All* detail page you will see the house specialty has been updated to Herbal.

Similarly, this action can be taken from a *List View* by using the + button in the upper right.

Tables and Survey are built to integrate seamlessly. Data can be visualized in Tables and edited in Survey, with your organizations complex workflow moving between as needed. A more complex example of this will be shown later in this tutorial with the *Hope Demo*.

Note: Survey is often the easiest way to edit data. However, Tables offers JavaScript APIs to directly edit data through your own custom user interfaces.

This concludes the *Tea Houses* demo. Next we will open the *Geotagger* Demo.

Learn More

For more information about launching Survey from Tables, view the *Editing With Survey* guide.

Geotagger Demo

For this portion of the tutorial we will explore the *Geotagger* demo. Select the tab labeled *Geo* and press *Launch Demo*.

The *Geotagger* demo is a mapping of sites around the city of Seattle (and anywhere else anyone has recorded and uploaded to the server).

Navigate View

After launching the *Geotagger* demo app, you will see a *Map View* of the points in Seattle, or possibly a larger space. To switch to *Navigate View*, tap the lined paper icon in the upper right and choose *Navigate*.

You will see the same map on the bottom portion of the screen, but the top will be replaced by a compass and heading readouts.

As you turn the compass should update. If you select a point, the compass will add an arrow pointing towards the selected point from your current orientation. The *Distance* and *Heading* values should fill in as well, and update as you move around.

The *Navigate View* can be useful if you have loaded geopoints into your database (either preloaded or collected in the field) and you need to find your way to these points. It can be integrated into other workflows to navigate a worker to a point and then launch them into another data collection activity.

Tip: The *Geotagger* demo also has a more complex *Map View* example. If you select *Map View* and tap on a map marker, that location will be highlighted in the *List View* on the top of the screen and it will expand to give you more information about it. This more sophisticated behavior is all performed in the JavaScript and HTML files.

Next launch the *Plot* Demo.

Learn More

For more information about *Navigate View*, view the *Navigate View* guide.

Plot Demo

For this portion of the tutorial we will explore the *Plot* demo. Select the tab labeled *Plot* and press *Launch Demo*.

The *Plot* demo is a fictional collection of crop data and graphs of that data.

Graph View

After launching the *Plot* demo app, you will see a custom view that lets you select which crop data you want to see. Choose *View Plots*.

The next screen is a *Map View* of the different sites in the records.

Each site is meant to represent an area where crop growth and health is being tracked. This provides a convenient view of the locations of the sample sites, and would be a good use for the *Navigation View* if a user had trouble finding one of the sites. Choose the *Ungoni* site.

The screen shows a *Graph View* of the crop height data collected for the *Ungoni* site. The bar graph shows corn crop heights across three different visits to this farm.

Tip: The graph was rendered using the `D3` JavaScript library. That library can render scatter plots, line graphs, graphs with error bars, and many other visualizations.

Updated Graph View

The graph was rendered on the device based on collected data. If new data is collected this graph will be updated. To demonstrate that, let's perform a new visit. Scroll down the page and press the *New Visit* button.

This will launch Survey to a form that the *Plot* application specified. Advance through the form. Notice that some of the fields are prepopulated, such as the plot being observed. Be sure to leave that set to *Ungoni*.

When you reach the prompt asking for crop height, enter: *130*.

Advance through the rest of the form, entering any data you like. Finalize the changes. When you return to the *Graph View* notice that a new visit has been added to the graph.

Tour the rest of the *Plot* demo to see a variety of other *Graph Views*. These are all rendered in custom JavaScript, and could be customized to your organization's unique needs.

Next launch the *JGI* Demo to see a demo of data collection directly through Tables.

Learn More

For more information about *Graph View*, view the *Graph View* guide.

JGI Demo

For this portion of the tutorial we will explore the *JGI* demo. Select the tab labeled *JGI* and press *Launch Demo*.

The *JGI* demo is a prototype of an application used by the *Jane Goodall Institute* to collect information about Chimpanzee behavior in the field.

Non-Form-Based Data Collection

After launching the *JGI* demo app, you will see a custom view where you can choose to continue or start a new *Follow*. Choose *New Follow*.

The next screen will prompt you to enter data about the *Follow* you are about to perform.

Note that we haven't launched Survey, this data is being collected by custom fields written in HTML, CSS, and JavaScript and rendered directly in the Tables view. Additionally, this data is not being used to create a single row in a single data table, it is going to be used by the following screen's JavaScript code to write to multiple rows in multiple data tables.

When you have filled in these data fields, press *Begin*. This will show start the *Follow* workflow.

This screen is hard not intuitive for a new user to understand. It is highly customized to the specifications of the *Jane Goodall Institute's* workflow. They originally used large paper notebooks with grids. They would check boxes on the grid based on observed chimpanzee behavior according to their own data collection protocols. This screen renders that same grid digitally and gives a worker access to dozens of fields simultaneously. Survey, Collect, or other form based data entry models would be too scripted and confining for this type of dynamic interaction record. Furthermore, this screen will advance to a new data point every 15 minutes. This is another workflow necessity that is only possible because of customized JavaScript.

Finally launch the *Hope* Demo.

Learn More

For more information about customized forms of data entry, view the *Editing Directly in Tables: Custom Views* guide.

Hope Demo

For this portion of the tutorial we will explore the *Hope* demo. Select the tab labeled *Hope* and press *Launch Demo*.

The *Hope* demo is a complex, longitudinal medical survey involving mothers with HIV.

Integrate With Survey

After launching the *Hope* demo app, you will see a custom view that lets you choose whether you are visiting with a new client or following up with an existing client. This study involves multiple visits from the same patient that occur over a period of months during and after the mother's pregnancy. Let's imagine that a client with ID number *44176* has come in for a 6 week follow up visit.

Select *Follow Up with Existing Client*.

This will open a *List View* that shows all the registered clients in the system (registered using the *Screen Female Client* option from the previous screen).

On the top of the screen is a search field that is custom written in HTML, CSS, and JavaScript. Use this to enter the client ID of the patient we imagine to be interviewing: *44176*. After pressing *Search* the desired client should be visible.

Select client *44716* to see a *Detail View* of that patient.

This page is a *Detail View*, but most of the collected data about this patient is not shown. Instead, links to the follow up Survey forms are provided to make follow up visits run smoothly. If you needed to update the patient's information, you could tap the pencil icon in the top right to launch the Survey form containing all of that patient's data.

Tap *Client Forms* and choose *Six Week Follow-Up*. This will launch the Survey to the specific form containing the six week follow-up questionnaire.

This demo imitates a single visit. Next you can try to emulate the full length of the study for a single patient from the initial screening through all the follow up visits. Notice that the *Graph Views* will update with this new information as well.

Learn More

For more information about integrating Survey and Tables, view the *Creating and Editing Data* guide.

Explore the Sample Application

This concludes the guided tour of the sample application for Tables. However, this is far from a complete reference. Please continue to explore the demo applications to learn more about the tool's capabilities.

You can find a more detailed user guide for Tables here: *Using ODK Tables*. And you can find a more detailed guide to managing Tables for Deployment Architects here: *Managing ODK Tables*. You can also find the source code for the demo applications in this tutorial in the Github repository for App Designer.

Using ODK Tables

- *Custom Home Screen*
  - *Showing/Hiding the Custom Home Screen*
  - *Enabling/Disabling the Custom Home Screen*
- *Table Manager*
- *Viewing Data*
  - *View Types*
    * *Spreadsheet View*
    * *List View*
    * *Detail View*
    * *Detail With Sublist View*
    * *Graph View*
    * *Map View*
    * *Navigate View*
    * *Custom View*
  - *Changing View Types: The Lined Paper Button*
- *Creating and Editing Data*
  - *Editing With Survey*
    * *Creating a Record: The + Button*
    * *Editing a Record: The Pencil Button*
    * *Spreadsheet View*
  - *Editing Directly in Tables: Custom Views*
- *Syncing Data*

Custom Home Screen

The custom home screen is an HTML file written by your organization to customize the look-and-feel of using Tables. If a custom home screen is provided, by default it will be the first screen shown after opening Tables.

Showing/Hiding the Custom Home Screen

To hide the custom home screen and see the *Table Manager* for a list of data tables on the device:

1. Open Tables. On the custom home screen press the button with three lines in the upper right.

2. The *Table Manager* will be visible with a full list of data tables stored on the device.

To return to the custom home screen press the back button in your Android navigation buttons.

> **Warning:** You may need to enable the custom home screen before it will appear.

Enabling/Disabling the Custom Home Screen

To enable or disable the use of the custom home screen, follow the instructions for *Tables Settings*.

Table Manager

The *Table Manager* allows you to modify table settings, delete tables, and import or export data into your tables. See the *Deployment Architect instructions* for details.

Viewing Data

Tables supports viewing collected data in a variety of formats. Survey allows you to review individual form instances, but Tables lets you view full data tables as well as create your

own customized visualizations. This is a significant departure from the form based model of data collection, and allows you to manage data directly on the device.

View Types

Tables offers a number of view options for presenting data. These will have been configured by your organizations Deployment Architect and you may not always have a choice in how you view your data. These view types are:

- *Spreadsheet View*
- *List View*
- *Detail View*
- *Detail with Sublist View*
- *Graph View*
- *Map View*
- *Navigate View*
- *Custom View*

Warning: Many of the view types in Tables are customizable by a Deployment Architect. This guide will provide some basic outlines of how to use these view types, but for more accurate instructions you may need to contact your organization's Deployment Architect.

Spreadsheet View

Spreadsheet View is the only view option that will be the same for all Data Management Applications. It is not customizable. It serves as a reliable way to view all of the data stored in a table on the device.

It is intended to have the familiar view as if you were using a spreadsheet program such as **`Excel`**. Each row represents a data record, which often (but not always) corresponds to a form instance created by Survey. You can scroll up and down to view all of the records, or left and right to see each column.

The thin column on the left is called the *status column*: it will show a different color based on the status of that row.

- **White (clear)** – The row is downloaded from the server and has not been modified.
- **Yellow** – The row is modified.
- **Green** – The row is an entirely new row
- **Black** – The row is deleted. It will show as black until you sync with the server and publish those changes.

Custom color rules can be set in table properties. They change the colors of spreadsheet cells based on the values of those cells. This can be useful in browsing larger data sets for records that meet certain criteria. For example, you might be recording crop heights and mark all cells with heights above a certain height as impossible so that they can be revisited or removed. For details on setting these color rules, see the *color rules guide*

Spreadsheet view can also be used to edit data. See the *Spreadsheet View editing guide* for further instructions.

List View

List View is a customizable view that will change based on your Data Management Application's code. In general, it is used to render a list of records from a data table, displaying only a few key values for each record.

Often the items in a *List View* are clickable to launch a *Detail View*, a *Detail With Sublist View*, or a *Custom View* to display details of that item. Sometimes these views can also be viewed as *Map Views* and *Navigation Views*. See *Changing View Types: The Lined Paper Button* for instructions on how to find if these view options are available.

Detail View

Detail View is a customizable view that will change based on your Data Management Application's code. In general, it is used to render the data from a single record in a logical fashion.

A *Detail View* may include some or all of the values from the record it is presenting, and it may include values drawn from other tables. The interface used to present that data is

completely customized by the organization writing the Data Management Application.

This view is often launched from a *List View* or a *Map View*.

Detail With Sublist View

Detail With Sublist View is a customizable view that will change based on your Data Management Application's code. It is a combination of a *Detail View* on the top half of the screen and a *List View* on the bottom half of the screen.

The *Detail View* on the top half of the screen follows all the same rules as a normal *Detail View*. In addition, it can control the *List View* rendered below it. There may be an interactive element within the *Detail View* that will cause the subordinate *List View* to redraw with different values.

Graph View

Graph View is a customizable view that will change based on your Data Management Application's code. In general, it is a often specialized *List View* that creates a graphical rendering of the data (such as a bar graph or pie chart). It may also be a specialized *Detail View* or *Custom View*.

A *Graph View* uses JavaScript libraries such as `D3` to create visualizations of collected data on the device. These will be rendered on demand using the data available, meaning that

they will update and change as new data is collected.

Map View

Map View is a partially customizable view that will change based on your Data Management Application's code. The top portion of the view is a *List View* representing the records in the data table, and the bottom portion of the screen renders the records as geopoints on a map using **`Google Maps`**.

Points are added to the map based on their recorded latitude and longitude values. The map can be navigated by pinching or widening to zoom in and out, or swipe around to move the window (the same controls as the stand alone **`Google Maps`**).

When a point is selected in a *Map View* it will usually update the *List View* on the top portion of the screen to select the same point, and possibly present more data about that point.

Navigate View

Navigate View is similar to *Map View*, but the top portion is replaced with navigational tools to aid in finding a location on the map in the real world. The bottom portion of the screen still renders the records as geopoints on a map using **`Google Maps`**.

When a point on the map is selected, the navigation controls on the top portion of the screen will update to guide you to the selected point.

- **Compass** shows you cardinal directions in addition to an arrow pointing at the navigation point.
- **Distance** shows the distance between your GPS location and the navigation point.
- **Heading** shows the direction that you are facing.
- **Bearing** shows the angle between your heading and your navigation point.
- **GPS Accuracy Spinner** shows the GPS's current accuracy estimate. It will change color based on how good this accuracy is.

The *Arrive* button will return you to the screen that launched the *Navigation View* with a success code. This may launch a follow up Survey or workflow to be performed at the navigation point.

The *Cancel* button also returns you to the screen that launched the *Navigation View*, but with a failure code. It indicates that the navigation point was not reached and it will not trigger a follow up workflow.

Custom View

Custom View is a completely customized view that is defined by your Data Management Application's code. There is no general pattern for *Custom Views*.

Custom Views are arbitrary user interfaces built on top of web technologies and rendered in Tables. They can be anything your organization needs to implement its custom workflow.

Note: *Custom Views* are not limited to displaying data. They can also be used to collect or modify data. See the guide for *editing data with custom views*.

Changing View Types: The Lined Paper Button

The view types that represent multiple records (*Spreadsheet View*, *List View*, *Map View*, *Navigate View*) can be alternately chosen, depending on what the Deployment Architect has configured in the table's settings.

To change to another view type, tap the lined paper icon from the upper right:

This will bring up a menu that lets you select your desired alternate view type.

Tip: *Graph View* is a special case. You may have the lined paper icon available to you, but it may only have *Spreadsheet View* as its alternative option, and may not have an option to return to the *Graph View*. Usually pressing the back button from *Spreadsheet View* will return you to the *Graph View*.

Graph Views also may not have the lined paper icon available at all if they are instead mapped as a *Detail View* or a *Custom View*.

Note: Not all view types will always be available. For example, if the data set does not contain geographic data, the *Map View* and *Navigate View* options will not be available.

Creating and Editing Data

Tables supports creating new rows and editing existing records and provides a variety of methods to do so. These can be integrated into your Data Management Application's workflow or accessed directly.

Editing With Survey

Most data change options use Survey to create or update the record. These options will launch Survey from the Table in question to directly edit the relevant record, and then return control back to Tables where you left off. Which options are available depends on which view type you are currently using.

Creating a Record: The + Button

The + button is available in any of the multi-record views: *List View*, *Graph View*, *Map View*, and *Navigate View*. This button will launch the configured Survey form to create a new record in the table currently being viewed. The example picture above shows the *Tea Houses List View* from the *ODK Tables: Sample Application*. If the + is pressed it will launch a Survey to create a new tea house in the table.

Editing a Record: The Pencil Button

The pencil button is available in any of the single record views: *Detail View* and *Detail With Sublist View*. *Detail With Sublist View* is considered a single record view as the *Detail View* portion is considered the controlling view, and the *List View* below is subordinate.

If the pencil button is pressed, it will launch the configured Survey form to edit the record currently be viewed. When the record has been updated and control returns to the calling view, the new details should be rendered in that view.

Spreadsheet View

Spreadsheet View also offers methods to launch Survey to create or edit records. If you know exactly the table or record you want to edit, this view may be the more direct option. You can also use *Color Rules* to find records that require your attention and then edit them directly.

- **Creating a Record** follows the same workflow as the other *multirecord views*. Press the + button to create a new row in the data table and see it in the *Spreadsheet View*.
- **Editing a Record** can be performed by long pressing on the desired row. A pop up will open when the long press is released.

This gives you the option to:

- *Delete Row* - This will produce a confirmation dialog make sure you want to delete the record. If affirmed, the row will be marked for deleted (or marked for deletion on the next synchronization).
- *Edit Row* - This will launch the Survey form corresponding to this record, similar to the *pencil button*.

Editing Directly in Tables: Custom Views

Tables supports direct creation and updates to data in the database through JavaScript API calls. These will be completely customized to your organization's Data Management Application and you may need to contact that person to find out how to use your particular design.

For more information on how to edit data with these custom views, see *Creating Customized Web Views*.

Syncing Data

See the instructions in the *ODK Services user guide*.

Warning: If a data table has any checkpoint saves (for example, caused by form crashes), the data table will not be synchronized. Checkpoints must be resolved before sync can proceed. The user must open a form on the problem table and either delete the checkpoint or edit the checkpoint. If editing, after that is complete they must save is as either incomplete or finalized. Once the checkpoints are eliminated, the user can initiate another synchronization, and the data in this table will then be synchronized with the information on the server.

4.5.2 Deployment Architect Guide

Managing ODK Tables

- *Prerequisites*
  - *Required*
  - *Recommended*
- *Table Manager*
  - *Importing Data*
  - *Exporting Data*
  - *Table Properties*
    - *General Settings*
      - *Columns*
    - *Display Settings*
      - *Color Rules*
    - *List View Settings*
    - *Map View Settings*
  - *Deleting Tables*
- *Setting Up a Form Development Environment*
- *Adding Your Own Tables*
  - *Initialize from ODK Application Designer*
- *Creating Customized Web Views*
  - *Custom Home Screen*

- *Configuring an App at Startup*
- *Launching With a Different AppName*

Prerequisites

Required

To create an Data Management Application that uses ODK Tables, you will need the ODK tools:

- *ODK Services*
- *ODK Application Designer*
- *ODK Cloud Endpoints*

As well as the third party apps:

- OI File Manager

If you have not installed Tables already, follow our guide for *Installing ODK Tables*

Recommended

We also recommend:

- *ODK Survey*

ODK Survey is not required, but Tables and Survey are built to seamlessly integrate and support more robust Data Management Applications.

Table Manager

The *Table Manager* provides you with access to administer tables, import/export data, modify their settings, or delete them altogether.

The *Table Manager* is the default home screen that is shown when you launch Tables, unless you have a custom home screen configured. See instructions for *Showing/Hiding the Custom Home Screen*.

This view lists every table on the device. If you tap on the name of a table, a *Spreadsheet View* will launch and you can view or edit the contents of that table.

Importing Data

You can load data from a CSV directly into a table that has been defined on the device. To learn how to add a table, see *Adding Your Own Tables*.

A CSV is a comma-separated values file. It is a common way to transport tabular data between different programs. Microsoft **`Excel`** can save and open CSV files, as can **`Open Office`** and a variety of other programs. Tables expects a certain format of the data in order to import the data correctly: the first line must be the comma-separated list of column names. The remaining lines must be the data for each of the corresponding columns.

For example, assume you wanted to load data into table of people's names, with column (field) names of Name and Age. In addition to those columns, your CSV file must also specify the unique row id (instance id) for each data row (the _id column). You can also specify the creator of the row, the time of creation, and other information. But, at a minimum, the file should look like:

```
_id,Name,Age
myUniqueIdforSam,Sam,27
```

This can be achieved by creating a spreadsheet in a spreadsheet editor and saving it as a CSV, or by copying the above text into a text editor and saving it with a `.csv` extension.

The upload process is as follows:

1. Place the CSV file onto the device and place it in the `config/assets/csv/` directory with a filename of `tableid.csv`. For example, `/sdcard/opendatakit/default/config/assets/csv/people.csv` would be the CSV file for the *people* table.
2. Launch ODK Tables and navigate to the *Table Manager* screen.
3. Press the plus + button at the top of the *Table Manager* screen.

4. Press *Select CSV File to Import*.

> **Warning:** You must have installed OI File Manager from the Play Store.

5. Find your file, select it, and press *Pick file*.
6. Press *Append to an Existing Table*.

The data will be read from the file and appended to your data table.

Warning: Prior to any deployment, you should sync your device to your server and export the data table and copy the exported CSV file back on top of the simple CSV file that you created above.

This ensures that the additional fields required by the ODK tools are properly populated and that a server-managed revision number is added to the data rows so that all devices will have the same internal ids for all of your data rows. This eliminates the possibility of the `tables.init` mechanism introducing duplicate records and speeds the sync process and minimizes the occurrence of conflicts across the devices when these devices first sync to the server.

Warning: Specifying the values for the _id column is important. Otherwise, each device, when it loads the CSV file, would assign different unique ids for each of the rows, causing much duplication and confusion.

Exporting Data

You can export any of your tables to a CSV file and associated supporting files. These files will be written to the `output/csv` directory on the device.

A Tables-exported CSV includes all the metadata needed to allow the table to be imported with exactly the same status settings, file associations and metadata settings on another device. Exporting produces the following files:

- file:*tableid.definition.csv* – this defines the data table's structure. It specifies the columns and their column types and is a copy of the file found under `config/tables/tableId/`
- file:*tableid.properties.csv* – this defines the column heading names, translations, and the HTML files associated with *List Views*, *Detail Views*, *Map Views*, and so on, and is a copy of the file found under `config/tables/tableId/`
- file:*tableid.csv* – this holds the data file that you can import to recreate the contents of your data table
- file:*tableId* – this holds an instances folder that holds folders named after each row id (the row id is cleaned up to remove any invalid filename characters such as slashes and colons). Each of those folders contains the row-level attachments for that row id.

To export a table:

1. Launch ODK Tables and navigate to the *Table Manager* screen.
2. Press the arrow -> icon at the top of the *Table Manager* screen.

3. Select the table you want to export.

4. Optionally specify a qualifier that will be inserted into the filenames of the emitted files before the `.csv` extension.
5. Press *Export*.

For example, if you were to export the *geotagger* table and specified *demo* as a qualifier, the following files would be written:

- `output/csv/geotagger.demo.definition.csv`
- `output/csv/geotagger.demo.properties.csv`
- `output/csv/geotagger.demo.csv/geotagger.demo.csv`
- `output/csv/geotagger/instances/1f9e.../137...jpg`
- `output/csv/geotagger/instances/...`

Table Properties

Table properties define a table and its behavior on the device. This includes basic necessities such as the table's ID and columns, references to sister files such as the forms to use when adding new rows or the html file to use when rendering a *List View*, and display settings such as map pin color rules and spreadsheet column width. Some of these properties are defined in the *Properties* worksheet in the XLSX file.

To modify the properties of a table:

1. Launch the *Table Manager*. Tap the gear icon next to the desired table:

2. This will launch the *Table actions* pop up. Select *Edit Table Properties*

3. This will launch the table properties screen.

The table properties can also be accessed by tapping that same gear icon in the *Spreadsheet View* of the desired table.

General Settings

The general settings define a table and are mostly not editable on the device. They include:

- **Display Name**: The string to display to as the name of the table, such as in the *Table Manager* view.
- **Table ID**: The ID of the table, which is used when performing database queries.
- **Columns**: The full list of data columns in the database table.

Columns

Tapping the *Columns* item will launch a list of all the columns in the table.

Note: The columns list excludes the status and metadata columns that the ODK 2 platform automatically adds. It only shows the columns holding data defined by the Deployment Architect.

If one of the columns is then selected, properties for that column can be set.

These include database definitions (which cannot be changed on the device):

- **Display Name**: The string to display as the name of the column in Tables.
- **Element Key**: The database key name for the value.
- **Element Name**: The name of the value in the form.
- **Column Type**: The data type of the value in the database.

Additionally, there are two editable properties:

- **Column Width**: The width of the column when it is displayed in **Spreadsheet View**
  - To change this value, tap the item labeled *Column Width*. A popup will appear in which you can enter a new width value.

– The next time you open *Spreadsheet View* for this table, the column width will be updated.

- **Edit Color Rules**: This lets you set the color rules. See the *color rules guide*.

Display Settings

Display settings change how the table is presented to the user. They include:

- **Change Default View Type**: Allows you to change the default view presented when a user selects a table. If selected this will display a pop with all available view types to choose from. This is typically *List View* or *Map View*.
- **Default Form**: This is the form ID to launch in Survey when adding a new row.
- **Edit Table Color Rules**: This lets you set the color rules. See the *color rules guide* below.
- **Show Status Column Color Rules**: If this is tapped it launches a screen that details the status column colors and their meanings.

Color Rules

Color rules allow you to modify the appearance of cells in *Spreadsheet View* based on the values of the data in those cells. You can have a collection of color rules set for a table to make visually scanning the spreadsheet much quicker and more informative.

To add a color rule:

1. Launch *Table Properties* and scroll down to select the *Edit Table Color Rules* item.

2. This will launch the color rules page. Tap the + button in the upper right to add a new color rule.

3. Choose the *Element Key* or column that will be affected by this color rule.
4. Choose a *Comparison Type* and *Value*. Combined, these two fields determine the equation to use when checking the color rule. For example, you might have chosen an *Element Key* of Visits that tracks the number of visits to a tea house. You might then choose a *Comparison Type* of $<$ and a *Value* of 1000. This would apply the color rule to all tea houses with a visit value that is less than 1000.
5. Choose the *Text Color* and *Background Color* to apply when this color rule evaluates to true. In our above example, we might set the *Backgroung Color* to red to highlight all the least popular tea houses.
6. Press *Save*.

To clear out the existing color rules, tap the trash can icon in the upper right.

List View Settings

The List View Settings determine which HTML files to use when this table is opened in a *List View* or a *Detail View*. These are typically set in the XLSX file, but can be updated here, or swapped between multiple options.

If this is not specified, the table will not be able to be opened in a *List View* or *Detail View*.

Map View Settings

The Map View Settings determine which HTML file to use when this table is opened in a *Map View*. This is used to render the *List View* at the top portion of the screen. This is typically set in the XLSX file, but can be updated here.

If this is not specified, the table will not be able to be opened in a *Map View*.

These settings also contain the *Color Rule For Map* option. This lets you choose between:

- **None**: Uses the default blue color for map markers, and green for a selected map marker.
- **Table Color Rules**: Uses color rules set up in the *Color Rules* screen to determine the map marker (the same color as the *Spreadsheet View* cells.
- **Status Column Color Rules**: Uses the color of the status column as the color for the map marker. This is useful to show which map items have had changes since the last sync.

Deleting Tables

The *Table Manager* allows you to delete a table off a device. However, this is generally discouraged and should rarely be performed.

To delete the table:

1. Launch the *Table Manager*. Tap the gear icon next the desired table:

2. This will open the *Table actions* pop up. Select *Delete this Table*.

3. You will then be shown a confirmation dialog. If you are sure, confirm, and the table will be deleted and marked for the next synchronization.

Setting Up a Form Development Environment

To get started creating your own Data Management Applications, go to the *ODK Application Designer* documentation.

Adding Your Own Tables

The creation of data tables is handled within the *ODK Application Designer*. ODK Tables can display and present data, but cannot create Tables on the fly. This enables the ODK Services application to enforce that the configuration of the device (its tables, HTML files, form definitions, and so on) are identical to those on the server.

Initialize from ODK Application Designer

See the documentation for *Building an Application* and *Creating Web Files* for more details on adding your own tables and defining their properties.

Creating Customized Web Views

Instructions for creating your own custom web views for presenting and modifying data, and implementing your custom workflow, go to the *web view design guide*.

For the convenience of Data Management Application developers, the ODK 2 platform provides a number of basic view types, such as *List Views* and *Detail Views*. These can be used and extended in your applications, or you can create something completely unique to your requirements with a custom view. Some of these views can be configured as defaults in *Table Properties*, and you can also launch directly into them with JavaScript calls from `/system/tables/js/odkTables.js`. Examples include:

- `openDetailView` to launch a *Detail View*, providing a query to select the desired record.
- `openListView` to launch a *List View*, providing a query to select the desired list of records.
- `openTableToMapView` to launch a *Map View* with a similar query to `openListView`
- `openDetailWithListView` to launch a *Detail With Sublist View*. The JavaScript file for the corresponding *Detail View* should then call `setSubListView` to fill in the bottom portion of the *Detail With Sublist View*.
- And more for different view and query types

The above APIs generally take a query as a parameter, run it in the background, and have the results available when the JavaScript file loads. These query results are retrieved with the `getViewData` API available in `/system/js/odkData.js`. There are more APIs available for reading, creating, updating, and deleting records in the `odkData.js` API. Some examples include:

- `query` to read data from the database
- `updateRow` to modify a row in a table
- `deleteRow` to delete a row from the table
- `addRow` to create a new row to a table
- `getAllTableIds` to get a list of all defined tables
- `getUsers` to get a list of user accounts
- And more

Third party libraries, such as *Math.js* or *Snap.js*, can also be included.

Example code to explore these APIs and how they can be used (including the *ODK Tables: Sample Application*) are available in the App Designer Github Repository.

Custom Home Screen

ODK Tables allows you to customize the app home screen. If you supply a custom home screen (`config/assets/index.html`), you will have the option of using this as the home screen of the app. For an example, see the *sample application*.

Configuring an App at Startup

If you are installing Tables on a new device and don't have a server set up from which to pull the data (see the *section about syncing*, you can alternatively configure Tables to import data at startup. This is useful during forms development, as you can push the form definitions, HTML, and JavaScript for your application data down to the phone from your computer and launch ODK Tables, and it will load data from CSV files into your data tables.

The configuration file must be titled `tables.init` and placed in the `/sdcard/odk/tables/config/assets` directory. Below is the complete contents of the `tables.init` file distributed with the sample application:

```
table_keys=teaHouses, teaTypes, teaInventory, teaHousesEditable, geotagger, plot,
↪ plotVisits, femaleClients, maleClients, geopoints, follow
teaHouses.filename=config/assets/csv/Tea_houses.updated.csv
teaTypes.filename=config/assets/csv/Tea_types.updated.csv
teaInventory.filename=config/assets/csv/Tea_inventory.updated.csv
teaHousesEditable.filename=config/assets/csv/Tea_houses_editable.updated.csv
geotagger.filename=config/assets/csv/geotagger.updated.csv
plotVisits.filename=config/assets/csv/visit.example.csv
plot.filename=config/assets/csv/plot.example.csv
femaleClients.filename=config/assets/csv/femaleClients.allfields.csv
maleClients.filename=config/assets/csv/maleClients.allfields.csv
geopoints.filename=config/assets/csv/geopoints.allfields.csv
follow.filename=config/assets/csv/follow.updated.csv
```

The table_keys key contains a comma and space separated list of table keys. Each table key can then have a `.filename` that indicate the filename of the CSV data that should be imported. This file should be under the `config/assets/csv` directory and the name should begin with the **tableId**, followed by an optional qualifier (for example, allfields), and end with `.csv`. If there are row-level file attachments for the table, they would be placed in a **tableId** file within the `csv` directory. Each row-level file attachment filename is relative to the folder for that row's id. If the rows _id column was *myUniqueIdForSam*, then the filenames in the data table for row-level attachments for that row would be relative to `/sdcard/opendatakit/default/config/assets/csv/tableId/instances/myUniqueIdForSam/`.

Note: Any table ids appearing in this file must already have their table definitions and metadata values defined in the definition.csv and properties.csv files within their correspond-

ing `config/tables/tableId` directory.

Tip: Only one attempt is made to read and import data at start-up. If that attempt fails, some or all tables may not be initialized or may be partially initialized. You can trigger a re-processing of this file by going to *Settings* and clicking *Reset configuration* then exiting the ODK tool and re-opening it.

As mentioned earlier, this file is never uploaded to the server. After you have created your user application and loaded data onto your device using this mechanism, resetting the app server will push all the configuration files and all of data (the data rows loaded by the `tables.init` script) up to the server (except for this `tables.init` file). Other devices that synchronize with the server will retrieve all of those data rows during the data-row synchronization phase. There is no need for the devices that synchronize with the server to have a copy of the `tables.init` file and independently perform these actions.

Launching With a Different AppName

The ODK 2 tools are designed to support multiple independent Data Management Applications running on the Android device. Each of our tools has the ability to run in the context of either a default application name, or a specified application name.

For further details on how to launch multiple AppNames and create your own new AppNames, see Survey's guide to *Launching With a Different AppName*.

ODK Tables: Internal Details

Layout of Application Files

The layout of a Data Management Application is as follows:

- `/sdcard/opendatakit`– directory containing all ODK 2 applications. Each application is a sub-directory within this directory.
- `/sdcard/opendatakit/default` – default application name (directory) for the ODK 2 tools

Within the application folder (`/sdcard/opendatakit/default`), the following directories are present:

- `config` – contains read-only configuration files that define the user's application (for example, the 5demos example application you just synced from https://opendatakit-tablesdemo.appspot.com). Within this folder are:

  - `assets` – contains files that initialize your data tables (in the csv sub-folder) and define the custom home screen and provides CSS files for overall appearance of your app, and JavaScript libraries and files for common behaviors in your app.
  - `tables` – contains directories that are named with table ids. Within these sub-directories, the ODK Survey forms and table-specific HTML, JavaScript, and CSS files are found. For example, the HTML file describing the list view for the tea houses table is found in `config/tables/Tea_houses/html/Tea_houses_list.html`.

- `data` – contains the database and row-level attachments (files).
- `output` – contains files that are generated (such as detailed logging files) or exported (such as CSV files) by the ODK tools on the device.
- `system` – an area maintained by the tools themselves (ODK Survey, ODK Tables, ODK Scan, and so on). These files are extracted and placed here by the APKs. You should not modify files in this folder; when first started, the ODK tools sweep this directory to verify that these files match their internal copy. Any deviant file is replaced with a fresh internal copy.

The automatic configuring and loading of data into ODK Tables is governed by the `config/assets/tables.init` file. It provides a list of table ids and the CSV files (located in the `config/assets/csv` folder) that should be imported to populate them. This is discussed in more detail in the *Tables User Guide*.

Note: This file is the only configuration file that is not synced to the server. This is to optimize start-up of your application on other devices; once this initial data has been loaded into your data tables and synced to the server, the other devices will obtain the data through an ordinary sync action.

Note: This file is scanned once. If the import(s) fail, it could leave some tables partially initialized. The file will be re-processed and data rows re-loaded by clicking on *Reset Configuration* on the *Settings* screen then exiting the ODK Tools and re-launching them. Upon being re-launched, the file will be scanned and processed.

Most of the app-level settings that are configured through the *Settings* page are stored in the `config/assets/app.properties` file. Excluded from this file are the *Server Sign-On Credential type*, and the values for that credential (such as username and password). This allows the application designer to specify and enforce most of the app-level settings (such as the server used when syncing) via the sync mechanism.

4.6 ODK Services

ODK Services is a program that handles database access, file access, and data synchronization services between all the ODK 2 applications. Mostly this happens behind the scenes, but you will need to install ODK Services as a prerequisite to using the other ODK 2 tools.

It also allows you to sync data collected by the ODK 2 tools with an ODK Cloud Endpoint. The Services application can be used to reset the Cloud Endpoint with the data that is on a tablet or to sync the data on the tablet with what is currently on the Cloud Endpoint.

4.6.1 User Guide

Installing ODK Services

Prerequisites

Before installing ODK Services, you will need the following third party apps:

- OI File Manager

There are no other ODK Android tools that are prerequisites to installing ODK Services. However, this tool is a prerequisite for all the other ODK 2 Android tools.

Installing Services

1. From your device's *Settings*, choose *Security*.

- Make sure *Unknown Sources* is checked.
- (On older versions of Android, this setting is in *Applications* rather than *Security*)

2. Open a web browser on your phone.
3. Navigate to https://github.com/opendatakit/services/releases/latest and download the latest ODK Services APK.
4. In the download window, you will see ODK_Services_vN.N.N.apk. - Select it to download the file.

- On older devices, the APK will automatically install after you approve the security settings.
- On newer devices, you must go to the download list, rename the file to restore the .apk extension (the extension will have been renamed to .man during the download process), then click on it to install it.

Note: You can also download the ODK Services APK to your computer and load it on your device via adb or another tool like AirDroid.

Tip: You can also install ODK Services on an Android emulator. However, this can be slow and is only recommended for developers actively working on Services.

Using ODK Services

Initial Server Configuration

Before you are able to synchronize your data or application files, you will need to configure your server settings. Instructions are provided in the *deployment architect guide*.

Authenticating Users

To log in or change the authenticated user, launch Services and open the user authentication screen. There are two ways to do this:

- **Launching From the Home-Screen**: Press the Action Button ( ) and select *Change User/Logout*

- **Launching From the Sync Screen**: From within the Sync screen, press the *Change User* button.

You will see the user authentication screen.

Within this page you can enter a new *username* and *password* and click the *Authenticate New User* button to contact the Cloud Endpoint and log in as this new user.

Note: To authenticate a new user, you must have a network connection and have the Server URL set appropriately. See the *deployment architect guide* for instructions on how to set this.

If you want to log out of your current user without logging into a new user, click the *Log Out* button. This does not require a network connection.

Syncing

Use this option to submit your data and download the latest updates from the server. When this process is finished, the data on your device and the server will match. You will also receive any updates to your application that your Deployment Architect might have made.

There are two ways to launch the Sync screen.

- **Launching From Services**: Launch Services. Click the *Sync* icon that looks like two arrows circling each other.

- **Launching From Another Tool**: From within Survey or Tables click the *Sync* icon (same as above). This will launch Services to the Sync screen. Below this is shown in ODK Survey.

You will then see the Sync screen.

Before syncing, you should verify all options are set correctly.

1. The username can be be changed by pressing the *Change User* button. Instructions are provided in the *Authenticating Users* section.

> **Warning:** If you authenticate as a different user after modifying data in the database, you could lose changes. Each user can have their own set of permissions to read, write, and delete different portions of the database. If you switch from one set of permissions to another, changes to areas that the new user is not allowed to modify may be lost.
>
> To prevent this be sure to synchronize all changes before authenticating new users.

2. The sync interaction has four options for managing file attachments. These are offered if bandwidth or storage is a concern:

- *Fully Sync Attachments - Default* - Synchronize all file attachments with the server.
- *Upload Attachments Only* - Only upload attachments from the device to the server.
- *Download Attachments Only* - Only download attachments from the server to the device.
- *Do Not Sync Attachments* - Do not sync any attachments.

Note: All four of the attachment options will fully synchronize your database. This includes all completed forms and collected data.

When you are ready to sync you data click on *Sync Now*.

Services will contact the Cloud Endpoint and synchronize your data. A progress dialog will be displayed and, alternatively, the status of sync can be obtained by looking at the notifications generated by Services in the notification area.

Note: The sync will proceed whether or not you remain on this page and you can use the back button to back out of it and return to your work.

Warning: Should you begin modifying data rows while syncing, the changes to those rows will not be synced until you save them as incomplete or finalize the row, and the act of editing will generally mark the sync as having ended with conflicts. This means that you must complete your edits and re-issue the sync to ensure that your changes are propagated up to the server.

Resolving Sync Conflicts

When you return from ODK Services and next access data, the ODK 2 tools will scan all tables looking for conflicts arising from the synchronization process. If any conflicts are found, you are required to resolve the conflict before proceeding to your activity. The options for resolving conflicts are as follows.

- *Take Local Version* - Use the version on the device, deleting the server version.
- *Take Server Version* - Use the server version, deleting the version that is on the device.

- *Merge Changes* - Will be enabled once all conflicts in the row's data fields have been decided.

Choose the desired option. Once the changes are reconciled, you can then proceed to the activity you were accessing and, when you next sync, the resolved conflicts and any new changes will be pushed up to the server. Then, other users will receive those changes when they sync to the server.

Device Settings

The device settings allow you to change configuration on your individual device. These settings will not be synchronized with the server.

1. Open Services. Press the Action Button ( )

2. Select *Settings* → *Device Settings*

- *Default Locale* specifies your preferred localization. By default this is set to US English. If you provide translations for your Data Management Appli-

cation, this is where to enable them.

- *Text Font Size* customizes the text size across the ODK 2 tools
- *Show Splash Screen* chooses whether to show a splash screen while each app launches.
- *Selected Splash Image* holds the image that will be displayed in the splash screen. By default this is an ODK logo, but can be set to your organization's own logo or another image.

Tables Settings

The tables specific settings modify the behavior of the ODK Tables tool. These settings will not be synchronized with the server.

1. Open Services. Press the Action Button ( )

2. Select *Settings* → *Tables Settings*

- *Use Custom Home Screen* chooses whether to display the `index.html` file of your Data Management Application or the list of tables when ODK Tables

is launched.

Troubleshooting

- If you are not seeing your forms in ODK Survey or your data tables in ODK Tables, try *Resetting Configuration*
- If you are seeing a list of data tables instead of your Data Management Application home screen when you launch ODK Tables, enable the *Use custom home screen* option in *Tables Settings*.
- If you are having trouble syncing, check your *Server Configuration*.

4.6.2 Deployment Architect Guide

Managing ODK Services

- *Prerequisites*
  - *Compatible Servers*
- *Server Configuration*
- *Reseting the App Server*
- *Administrator Settings*
  - *Setting an Administrator Password*
  - *Accessing Administrator Settings*
  - *Managing Server Settings*
  - *Managing Tables Settings*
  - *Managing Device Settings*
  - *Locking Administrator Settings*
- *Resetting Configuration*

Prerequisites

ODK Services is a prerequisite to all Data Management Application. You will also need the ODK tools:

- *ODK Application Designer*
- *ODK Cloud Endpoints*

You will also need at least one of the following ODK tools:

- *ODK Survey*
- *ODK Tables*

Survey and Tables can work independently and do not require you to use both.

Finally, you will need the third party apps:

- OI File Manager

If you have not installed Services already, follow our guide for *Installing ODK Services*

Compatible Servers

It is important to match your version of ODK Services with an appropriate version of a *ODK Cloud Endpoints*. To do this, find the release from the Services Releases page and match it to a release from the Sync Endpoint Releases page. Or, if you want to use ODK Aggregate, check the release notes for the appropriate version.

Server Configuration

Before you are able to synchronize data or application files to a device, you will need to configure your server settings within Services. This tells the device which server to contact and what user to authenticate. After these settings are configured, you can optionally lock them with an administrator password. See *Administrator Settings*.

1. Open Services. Press the Action Button ( )

2. Select *Settings* → *Server Settings*

3. Choose *Server URL* and specify your server URL

Note: If you are using SSL, be sure to specify `https://...`

---

4. Authenticate user credentials

Note: If your server is configured to allow anonymous access this step is optional.

---

   1. Change the *Server Sign-on Credential* to *Username* and enter the appropriate credentials in the *Username* and *Server Password* fields.
   2. Exit out of the *Server Settings* page, and then the *Settings* page, by using the back button.
   3. You will then be asked to *Authenticate Credentials*. Select the *Authenticate New User* option.

> **Warning:** If you decline (by choosing to *Log Out*), or if your credential is rejected by the server, then your credential will be reset to the anonymous (unprivileged) user.

4. On the next screen select *Verify User Permissions*.

5. After the verification succeeds, you will see a *Verification Successful* popup, select *OK*.

Reseting the App Server

Resetting your app server pushes the configuration and data on your tablet up to the server. After pushing files from *ODK Application Designer* to the device, this is how to push those files to the server to initialize your Data Management Application. All other devices synchronizing with your server will receive these configuration and data files.

Note: This option should only be used to initialize or update your Cloud Endpoint.

Warning: If a data table on the server does not exist on the device, that table, all of its data, and all associated files (such as forms) will be deleted from the server.

If a data table on the server is identical to one on the device, the data in that table will be synced and the files on the server will be updated to be exactly those present on the device (deleting any files associated with this table that existed only on the server).

Before resetting:

1. It is critical that you first ensure that your device contains all the tables, files, and data you want to preserve in your application. See *instructions*.
2. Authenticate as a user who has administrator privileges. See *instructions*.

To reset the server you must launch the Sync screen. Launch Services. Click the *Sync* icon.

You will then see the Sync screen.

Before resetting, you should verify all options are set correctly.

1. The username can be be changed by pressing the *Change User* button. If you do not see the *Reset App Server* button then you need to change users to an administrator. Instructions are provided in the *Authenticating Users* section.

> **Warning:** If you authenticate as a different user after modifying data in the database, you could lose changes. Each user can have their own set of permissions to read, write, and delete different portions of the database. If you switch from one set of permissions to another, changes to areas that the new user is not allowed to modify may be lost.
>
> To prevent this be sure to synchronize all changes before authenticating new users.

2. The sync interaction has four options for managing file attachments. These are offered if bandwidth or storage is a concern:

- *Fully Sync Attachments - Default* - Synchronize all file attachments with the server.
- *Upload Attachments Only* - Only upload attachments from the device to the

server.

- *Download Attachments Only* - Only download attachments from the server to the device.
- *Do Not Sync Attachments* - Do not sync any attachments.

Note: All four of the attachment options will fully synchronize your database. This includes all completed forms and collected data.

Click on *Reset App Server*. A confirmation dialog will popup asking you to confirm resetting the App Server. Again, this can delete all data on this Cloud Endpoint! If you are sure you want to continue, click *Reset*.

Services will contact the ODK Cloud Endpoint and attempt to push all configuration and data currently on the tablet up to the specified Cloud Endpoint. A progress dialog will be displayed and, alternatively, the status of resetting the app server can be obtained by looking at the notifications generated by Services in the notification area.

Note: The sync will proceed whether or not you remain on this page and you can use the back button to back out of it and return to your work.

Warning: Should you begin modifying data rows while syncing, the changes to those rows will not be synced until you save them as incomplete or finalize the row, and the act of editing will generally mark the sync as having ended with conflicts. This means that you must complete your edits and re-issue the sync to ensure that your changes are propagated up to the server.

Administrator Settings

Administer settings allow you to lock in certain settings so that they cannot be changed without the administrator password.

Tip: To modify a setting locked behind administrator privileges, enter the administrator password and then access that setting.

Setting an Administrator Password

1. Open Services. Press the Action Button ( )

2. Select *Settings* → *Enable user restrictions*

3. Select *Admin Password*. A prompt will appear where you can enter a new admin password.

4. After creating an admin password, the screen show show that it is enabled.

5. Back out to the Settings screen

Accessing Administrator Settings

After the administrator password is set, you can enter it to access the administrator settings.

1. From the Settings screen, select *Admin Access to Settings*

2. You will be prompted to enter the admin password.

3. After entering the correct password, you will see the full list of administrator settings available to you.

Managing Server Settings

- *Server URL* - if checked the Server URL will be locked.
- *Server Sign-on Credential* - if checked the means of authenticating will be locked.
- *Username and/or Password* - if checked the username and password fields will be locked.
- *Allow unsafe/unsecure Authentication* - if checked Services will allow synchronization with servers not using SSL encryption.

> **Warning:** This option should only be used for testing. When deployed to the field you should always enable SSL encryption.

Managing Tables Settings

- *Use custom home screen* - if checked the custom home screen option will be locked.

Managing Device Settings

- *Text Font Size* - if checked the text font size will be locked.
- *Change Splash Screen settings* - if checked the splash screen image and enable/disable flag will be locked.

Locking Administrator Settings

When you have finished configuring the administrator settings, back out of the menu. You will then see the normal settings menu, but with all appropriate settings locked. To modify these locked settings, follow the instructions for *Accessing Administrator Settings* and repeat the process.

Resetting Configuration

This option will clear the ODK 2 cache of table and form definitions and scan the file system to refill that cache. This is automatically run after each successful sync operation to ensure that Survey and Tables display the correct information. If you have manually modified files inside of the `/sdcard/opendatakit/` folder via **`grunt`** commands, with **`OI File Manager`**,

or by some other means, you may need to use this option to refresh the cache. If you are not seeing forms or tables that you expect, this option may fix that problem.

Note: This option does NOT delete any data or files. It also does not reset your server URL setting. But it will log you out of your currently authenticated user and clear your device and tables settings.

After pressing this option, you will be prompted to confirm this is what you want to do.

Press *OK* to clear the config. Back out of the *Settings* menu. The next time you run Tables or Survey they will rerun their initialization logic, which may take a few moments.

ODK Services: Internal Details

Sync Details

Syncing has two phases. In the first phase, data tables are created on the device that correspond to the data tables on the server, and the form definitions and other files on your device are made to exactly match those available on the server (updating them as needed).

> **Warning:** If a data table on the device does not exist on the server, the configuration files and all associated forms for that table will be removed from the device. To prevent data loss, the table itself will not be deleted, but, by removing all of the configuration files for that table, the data will generally be unusable.

In the second phase, it synchronizes the contents of the local data tables with the contents on the server, including any row-level file attachments associated with individual records in the data table. Row-level file attachments are bundled and synced one row at a time.

Unlike ODK Collect, where individual forms can be added and removed at will, ODK Services and the ODK 2 tools are organized Data Management Applications consisting of a set of interrelated data tables and forms. All the forms and tables on the server collectively define the Data Management Application* and ODK Services ensures that the device conforms to that Data Management Application definition. You can operate multiple independent Data Management Applications on a single device by placing their files and forms under different application folders within the `/sdcard/opendatakit/` folder. Each such application will publish to a different ODK Cloud Endpoint. This is a significant and powerful change from the ODK 1 mindset.

Database Details

ODK 2 data is stored in a SQLite Database running on the Android device. After a device synchronizes with the server, this database will fully match the schema and contents of the database running in the *ODK Cloud Endpoints*.

Each Survey form instance will write to a row in the database. However, this mapping is not one-to-one: the form may not fill the entire row's columns and another form might fill other fields in the same row. Furthermore, sub-forms allow you to launch forms that write to other database rows. See *ODK Application Designer* and *ODK Survey* for more details.

Data tables and schema can also be created manually and used as a back-end for your Data Management Application using *ODK Tables*.

At any point you can copy the local database on the Android device onto your desktop computer and inspect its contents and schema. If your application name is *default* then the database is stored in:

```
/sdcard/opendatakit/default/data/webDb
```

To inspect the database, use the `adb pull` command (Google documentation is available here). Then use a program such as DB Browser for SQLite to view the database. Further instructions are available in the *Pushing and Pulling Files* guide.

4.7 ODK Scan

Warning: ODK Scan is not yet fully released! It is not guaranteed to work at the same quality level as ODK Survey, Tables, Services, or the rest of the release ODK 2 tools.

It is currently at the **Beta** stage, which mean it does not have all features, but is not likely to have significant reductions or alterations in functionality. Beta releases are provided to gather user feedback on the usability and capabilities of the application, as well as bug reports (to make the application more robust). Updates may result in loss of data or incompatible changes in form designs.

ODK Scan is an Android application that uses the device's camera and specialized code to automatically digitize written data from paper forms. Using the app, users take pictures of paper forms and ODK Scan detects and collects the fill-in bubble, checkbox, and written number data. It also saves image snippets of handwritten text and displays them on the screen for easy data entry. The digitized data can then be validated, exported, saved into a database, and used for custom data reports. This workflow from paper form to digital database occurs in five processes and is supported through the use of ODK suite tool

4.7.1 User Guide

Installing ODK Scan

Prerequisites

Before installing ODK Scan, you will need the following ODK Tools:

- *ODK Services*
- *ODK Survey*
- *ODK Tables*

As well as the following third party apps:

- OI File Manager

Installing Scan

Warning: ODK Scan is only compatible with Android versions 4.4 or newer.

To install the apk:

1. From your device's *Settings*, choose *Security*.

- Make sure *Unknown Sources* is checked.
- (On older versions of Android, this setting is in *Applications* rather than *Security*)

2. Open a web browser on your phone.
3. Navigate to https://github.com/opendatakit/scan/releases/latest and download the ODK Scan APK.
4. In the download window, you will see ODK_Scan.N.N.apk. - Select it to download the file.

- On older devices, the APK will automatically install after you approve the security settings.
- On newer devices, you must go to the download list, rename the file to restore the .apk extension (the extension will have been renamed to .man during the download process), then click on it to install it.

Note: You can also download the ODK Scan APK to your computer and load it on your device via adb or another tool like AirDroid.

Note: To synchronize your data with the cloud you will also need *ODK Cloud Endpoints*.

Note: Before scanning you'll first need to create printable form template using the *ODK Scan Form Designer*.

Using ODK Scan

- *Scanning a Form*
  - *Prior to scanning*
  - *Scanning the form*
- *Survey: View, Verify, & Edit Data*
  - *Reviewing Your Data*
  - *To verify and edit any of the data*
  - *Saving and Finalizing Changes*
- *Your Data in Tables*

Scanning a Form

Prior to scanning

Have a printed form ready. For more information on printing the form created in Form Designer see the *printing instructions*.

Open the Scan app, and be sure that the template you want to use this session is selected in the settings. Go to *Settings* → *Templates to Use*, make sure the correct form is selected, and click *OK*.

Scanning the form

1. When you are ready to begin scanning, click *Scan New Form* from the main page in Scan. This will bring up a camera window.
2. Adjust your positioning until there is a good view of the form in the viewfinder. When you are ready to take the picture, **tap the camera icon**.
   - The form should take up 80% of the photo area.
   - Make sure that the form is lying as flat as possible so that there will be no curvature in the form.
   - Tap anywhere in the viewfinder to focus the camera.

3. If the preview of the photo looks good, tap the checkbox icon to move onto the next step. To retake the photo tap the *Back* button and to exit the camera tap the *X*.

4. Once you select the check mark to begin photo processing, a small message will pop up saying *Processing photo in background.*

5. When the photo has been successfully (or unsuccessfully) processed, you will see a notification at the top of the screen in the Android toolbar. Pull the top toolbar down and tap the ODK Scan notification. This will open Scan and pull up the photo of the selected scan.

   - The successfully processed photo will show an overlay of colored boxes that indicate the fields that Scan has detected. Any bubbles or checkboxes recognized as filled will show an overlay of the value that was assigned to them in the form designer. Number fields will show an overlay of the number that the app recognized for each digit.

   - If the photo was unsuccessfully processed you will be prompted to retake the photo.

6. From this screen, you can choose to either begin reviewing the data from this scan, or save it to review later. Press *Transcribe* to be taken into ODK Survey where you will be able to view and edit data.

- Or press *Save*. This scan is now accessible by tapping the drop down options (at the top right of the screen), then *Main Menu* → *View Scanned Forms*). From the drop down options, you can select *Scan New Form* to continue scanning and saving forms.

Tip: To increase accuracy of Scan's results, you can consider building a stand with a clear plastic surface to place your phone or tablet on top off while you take the each photo. The stability can help improve the alignment and reduce blur in photos. Below is an example of a stand built with PVC piping and Plexiglass.

Survey: View, Verify, & Edit Data

Reviewing Your Data

You'll be taken to Survey after pressing *Transcribe* on a scan. There you'll see a clickable list of all of the fields pulled from your form template, your *Table of Contents*. You can return to this screen when transcribing data by pressing the button on the top, left (with your form template's name, the example image below being *scan_TB03_Register1*).

To verify and edit any of the data

Select the field you want to view, and you'll be taken to a screen where you'll find an image of the field and the data, as interpreted by Scan, and an editable box below. Type in any changes if there are discrepancies between the data digitized by Scan and the ground truth data.

Navigate to the next section to validate and edit either by:

- Pressing the *Next* or *Back* buttons at the top of the screen,
- Or go to the button with your form name and select *Contents* to return to the main screen of captured data.

Note: The order that these fields are presented can be set when originally creating the form template in Form Designer. With a data field selected, in *Form Properties* enter a numbered order (for example: 1, 2, 3, and so on) in *Order of Fields*.

Note: Text boxes and text fields cannot be digitized. However, Scan will capture an image of text boxes (not text fields: text fields are to be used primarily as labels on your form), and when verifying data in Survey you can type in the data directly into the app.

Saving and Finalizing Changes

You have the option of saving changes you've made to the data and returning to it later to further review. Go to the *Form Name* → *Save Changes + Exit*. You can access this scan's data again from *Scan>* → *View Scanned Forms*. They will be arrange in the chronological order they were originally scanned.

If you've made changes you don't want to keep, *Form Name → Ignore Changes + Exit*.

Once you've verified all the fields, select *Form Name → Finalize Changes + Exit*. You will also have the option to *Finalize Changes* if you are navigating through the data fields by using the next button and reach the end of the data contents. Once you are finished here you will return to Scan, where you can scan a new form or transcribe a saved scan. Both options accessible through navigating to Scan's Main Menu.

Your Data in Tables

With each verified and finalized scan, a new line of data will be entered into Tables. To view (on your device) the verified data collected in this instance: open the Tables app and select the line with your form's name listed. This will open up a spreadsheet of your data. If you need to need to edit the data in a record from here:

1. Double tap on the cell you want to edit.
2. You'll be given the option to either *Edit* or *Delete* that row. Choosing *Edit* will launch the form in Survey.
3. You can change the *View Type*, *Color Settings*, and more by pressing the settings wheel and making any changes you need.

4.7.2 Deployment Architect Guide

Managing ODK Scan

- *Prerequisites*
- *Transferring a Form Template to the App*

Prerequisites

To create an Data Management Application that uses ODK Scan, you will need the ODK tools:

- *ODK Services*
- *ODK Survey*
- *ODK Tables*
- *ODK Application Designer*
- *ODK Cloud Endpoints*

As well as the third party apps:

- OI File Manager

If you have not installed Scan already, follow our guide for *Installing ODK Scan*

Transferring a Form Template to the App

ODK Scan works with machine readable forms created using the *ODK Scan Form Designer*. Refer to the *Using ODK Scan Form Designer* for instructions on how to create these forms.

After creating a form with Form Designer, you'll have generated the machine readable files. To push them to your device, you will use the same mechanism that is used to push Survey and Tables files to the device.

1. Create a form using the ODK Scan Form Designer. Save that form with the *Save to File System* option.
2. Follow the instructions in *Moving Files To The Device* to push updates to the device. These describe pushing Survey files, but they will push Scan files to the device too with the same procedure.
3. To confirm that the *[your_form]* template has been successfully been transferred, open the ODK Scan app on your device and go to *Settings* (the wheel icon) and select *Templates to Use*. The folder name should appear in the list of templates.

Managing ODK Scan's Data

ODK Scan shares a database with the rest of the ODK tools, and the data can be accessed using the normal means through the *ODK Cloud Endpoints* and *ODK Suitcase*. However,

Scan adds extra columns to store snippets of each data field's original image, the image file type, and the data value predicted by Scan.

Suitcase Formatting

ODK Suitcase is the mechanism for downloading and exporting data from the ODK 2 data tables into local `.csv` files. Suitcase has a specific option to format Scan's `.csv` files more to be more readable. The image below shows this option underlined in red.

4.8 ODK Application Designer

ODK Application Designer is a tool to help you design *data management applications* on top of the ODK 2 framework. It works in conjunction with **`Excel`** or **`OpenOffice`** for form design, the **`Chrome`** browser for rendering, and your favorite editor for template design.

In the context of the ODK 2 tools, application design consists of:

- designing the forms used in data collection (by ODK Survey)
- designing the HTML landing pages and screens used for navigating, curating, and visualizing that data on your Android device (within ODK Tables).
- customizing the look-and-feel of both of these via customized images, logos, and CSS rules.
- designing mark-sense forms for paper-based data entry (by ODK Scan)

Tip: The tools operate independently – you are not required to use all the tools, or even install them on your device. If you are only interested in data collection, you may only want ODK Survey. Or if you are only interested in data dissemination and visualization, you might only want ODK Tables.

Simply select the combination or individual tool that fits your needs. However, all of these tools require ODK Services to access the database, sync to a server, and vend HTML files.

The major goals of the ODK Application Designer are:

1. Simplify the form-design process by providing a preview of your form with the same screen geometry as your target Android device. You no longer have to copy each iteration of your form onto your device to see how it will look or fit on a smaller screen.
2. Simplify the design and testing of customized list- and detail- views in ODK Tables, and in the design of graphical representations of data within ODK Tables
3. Simplify the customization of the look-and-feel of your forms through a simple visual theme editor / generator where your modifications can be immediately viewed and the resulting CSS styles or theme can be saved to a file for later incorporation into your application deployment.
4. Simplify the conversion of the XLSX file into a form definition by providing a drag-and-drop conversion app running locally on your desktop.
5. Enable the creation of mark-sense forms (ODK Scan Form Designer) that can be scanned by your Android device for data input. The resulting data is available to the other ODK 2 tools without need to communicate with a remote server.

4.8.1 Deployment Architect Guide

Setting Up ODK Application Designer

Application Designer Prerequisites

You must install the following software on your computer in order to use Application Designer:

- - Java is required by the Android SDK
- - Google's **`Chrome`** browser.
- - a framework for easily building fast, scalable applications. Download Version 6.2.2 or higher and install it from NodeJS
- - a task-based scripting environment (installation is described below).

- - the software development kit for Android devices (installation is described below).

> **Warning:** It is tricky to foresee all the issues that can crop up on many different machines and setups. If something in this process does not go as expected, please check the ODK Forum.

Java

Make sure Java 7 or higher is installed on the computer you plan to use. If it is not, download and install it. If you are using MacOSX, it may require special care and attention. See MacOSX Java install and MacOSX Java FAQ.

NodeJS

You must use Version 6.2.2 or higher. To avoid directory path problems on Windows, we require **npm** version 3, and that is only available on this node release (or higher). Follow the instructions to install NodeJS.

For Windows

After installing NodeJS, ensure the location of the `npm` folder is added to the *PATH* variable of your system. If it is not, subsequent calls to access grunt will fail. For example: `C:\Users\[username]\AppData\Roaming\npm`

For Mac/Unix

After installing NodeJS, open a **terminal** (which you can do by clicking the spotlight in the top right corner of the screen, typing **terminal**, and clicking the program named **Terminal**) and type:

```
$ npm --version
```

> **Warning:** If a number is not displayed, but you instead receive a message that the command **npm** cannot be found, you will have to perform some additional configuration.
>
> As of this writing, by default NodeJS installs its commands into `/usr/local/bin/`. In the **terminal**, type:
>
> ```
> $ ls /usr/local/bin/npm
> ```

If this command outputs something like `/usr/local/bin/npm`, but you are still unable to run:

```
$ npm --version
```

try running:

```
$ /usr/local/bin/npm --version
```

If this is successful, then **npm** is successfully installed, and you will just have to add `/usr/local/bin/` to your system *PATH* variable (see below).

If the command:

```
$ ls /usr/local/bin/npm
```

outputs a message telling you permission is denied, then you will have to change the ownership of the `/usr/local/` and `/usr/local/bin/` directories. On Mac, follow the instructions to take ownership of these directories, or to at least give yourself read permission. On other Unix systems, use the **chown** command or the user-interface appropriate to your distribution to do so.

Grunt

After installing NodeJS, install **grunt** by doing the following:

Note: These installation steps are copied from the Grunt Getting Started guide.

On Windows, open a **cmd** window (go to Start Menu, search for **cmd** and open it); on MacOSX, open a **terminal** window. Within this window, type:

```
$ npm install -g grunt-cli
```

If the above command is unsuccessful, some machines may need to append **sudo** at the beginning of the command. If **grunt** is successfully installed, the following command:

```
$ grunt --version
```

Should display the installed version of **grunt**. For example the version might be `grunt-cli v1.2.0`

Warning: If **grunt** is not found, you may need to add it to the *PATH* variable of your system.

Android SDK

To install the Android SDK:

1. Browse to the Android SDK download page.
2. Scroll down on this page to the section labeled: *Get just the command line tools*.

Note: You can alternatively install the full Android Studio if you so wish, in which case you should follow Google's instructions and then skip to step 6 of this guide.

---

3. Within that section, download the appropriate file(s) based on your operating system.
4. Accept the license agreement
5. Wait for the install of the SDK Tools to complete. Windows will need to manually run the `.exe` file previously downloaded to start installation.
6. Run the SDK Manager

- On Windows, it is available in the *Start Menu* under Android SDK Tools

> **Warning:** If the packages fail to install, you may need to run the Android SDK as an Administrator.

- On Mac/Unix, open the SDK folder you downloaded above. In the `bin/` or `tools/` directories (on some versions it is in both places–it doesn't matter which you use), double click the file called `android`.

7. Select the latest versions of the following packages (by checking their checkboxes):

- Android SDK Tools
- Android Platform-tools
- Android Build-tools

8. If extra packages are selected, you may unselect them before installation.
9. Click *Install 3 packages* in the lower right corner of the screen.
10. A licensing pop-up dialog will appear. Accept the license agreement(s) by selecting the *Accept License* option. If there are multiple licenses, you may need to select each license in the *Packages* window on this dialog and check

this *Accept License* option for each of them before the *Install* button will become enabled.

11. Click the *Install* button on that dialog to begin the installation process.

Among many other things, this will install the Android Debug Bridge software on your computer. This tool enables the scripted pushing of files and APKs down to your Android device. See adb (Android Debug Bridge) for a listing of its capabilities.

Next, on Windows open a **cmd** window (open the Start menu, type **cmd** in the search box, select and open it), and on Mac/Unix open a **terminal** window. Type:

```
$ adb version
```

If this displays a version string, then your installation is complete; you are done with this section and can move on to *Installing Application Designer*.

Warning: If there is an error complaining about Java not being installed, you will need to close this **cmd** or **terminal** window and download and install Java. After installing Java, open a new **cmd** or **terminal** window and type this command again.

Warning: If **adb** is not found, then you need to add it to the *PATH* variable of your system.

Add `adb` to your *PATH*

For Windows

1. Open a Windows File Explorer and navigate to the location of your Android SDK. This will typically be at one of: `C:\Users\your_username\android-sdks` or `C:\Program Files\Android\android-sdk` or `C:\Program Files (x86)\Android\android-sdk`.
2. Navigate into the `platform-tools` folder.
3. Click in the file path at the top of the File Explorer window. The path will become a selected text string. Copy it into your copy-and-paste buffer.
4. Open the Start menu.
5. Right-click on *Computer*.
6. Choose *Properties*. The System Control Panel screen opens.
7. Select *Advanced system setting* on the left sidebar. The System Properties dialog opens.

8. Click on the *Environment Variables…* button at the bottom of the screen. The Environment Variables dialog opens.
9. Select the *Path* variable in the bottom System variables scroll window.
10. Click *Edit…*
11. Click into the *Variable value* text box.
12. Press your *End* key to move to the very end of this extremely long string.
13. Enter ';' and paste the `platform-tools` directory path after that semicolon.
14. Click on *OK* and exit all of the windows.
15. Verify that you have made the change by closing all **cmd** windows and open a new one (so it picks up the change), and type

```
$ adb version
```

You should now see the version of the **adb** tool. For example: `Android Debug Bridge version 1.0.31`. You can now move on to *Installing Application Designer*.

For Mac/Unix

The *PATH* variable is nothing more than a default list of places the system looks for commands. Open a **terminal**. Type:

```
$ echo $PATH
```

You will see a colon-separated list of folders on your computer. (echo means just print whatever comes next, and the `${ }` means that the system will treat *PATH* as a variable, not a program. You don't need to know this to follow these instructions, but knowledge is power.) For example, you might see something like this:

```
$ echo $PATH
  /usr/local/bin:/usr/local/sbin:/usr/bin:/bin
```

This means that when you type:

```
$ adb --version
```

the system will look for the command called **adb** in the directories `/usr/local/bin/`, `/usr/local/sbin/`, `/usr/bin/`, and `/bin/`.

Note the location where you downloaded the Android SDK. It should contain a folder called `platform-tools`, which itself contains the program **adb**. If this was in the folder `/Users/someuser/Desktop/android-sdk/` you should be able to run:

```
$ /Users/someuser/Desktop/android-sdk/platform-tools/adb --version
```

This works because we're telling the computer exactly where the program **adb** exists. By putting the `platform-tools` directory on the system's *PATH* variable, we will be able to just type **adb** and have the system find it in the `/Users/someuser/Desktop/android-sdk/platform-tools/` directory.

This process is more involved on Mac/Unix than on Windows. Use a text editor (not **Word**, but something like **TextEdit**), select the option to open a file, and browse to your home directory. You can find your home directory by typing:

```
$ echo ~
```

in a **terminal**. ('~' is a shortcut for the home directory.) Macs use a hidden file called `.bash_profile` in the home directory to set variables like *PATH*. Other Unix systems use files like `.bashrc`. You might have to check the specifics for your distribution to know which you should use. Open the appropriate file. If the file does not already exist, create a new file that will be saved with the appropriate name in your home directory.

We want to add the location of the **adb** tool to your *PATH* while preserving the existing *PATH* information. Assuming that your **adb** program is in the `/Users/someuser/Desktop/android-sdk/platform-tools/` directory, you would add the following command to the end of the `.bash_profile` file:

```
$ export PATH=${PATH}:/Users/someuser/Desktop/android-sdk/platform-tools
```

Save the file, close the **terminal** window, open a new **terminal** window, and type:

```
$ echo $PATH
```

You should see your old path with the new directory you added above, and you should now be able to run:

```
$ adb --version
```

Tip: If you are going to be heavily customizing the look-and-feel of the application with a lot of external JavaScript libraries, you might also choose to install **bower**.

You can now move on to *Installing Application Designer*.

Installing Application Designer

Download the zip file.

Unzip the file you downloaded and move the resulting folder to somewhere other than your `Downloads` directory; such as your `Documents` folder.

Windows Users Tip

You will be opening a **`cmd`** window and changing your current directory (using the **`cd`** command) into this directory every time you use this tool. It is therefore useful to create a shortcut that opens a **`cmd`** window directly into this directory:

1. Open a file browser and navigate to the unzipped directory containing a number of files and directories, including a `Gruntfile.js`.
2. Click into the top location bar that displays the nested list of folders to this folder.
3. Copy this path to the cut-and-paste buffer.
4. Now, move down to the list of files, right-click.
5. Select *New...*, *Shortcut*.
6. Type **`cmd`** for the location of the item.
7. Click *Next*, and then *Finish*.
8. Select this newly-created **`cmd.exe`** shortcut and right-click.
9. Select *Properties*.
10. Click on the *Start in* text box, delete its contents, and paste the path to this folder.
11. Click *OK* to accept the change.
12. Double-click the **`cmd.exe`** shortcut to open a **`cmd`** window.
13. Confirm that it opens in the intended directory (you should see the full path to that directory displayed to the left of the blinking cursor).

MacOSX Users Tip

`Terminal` will open a new **`terminal`** window if you drag a folder (or pathname) onto the **`Terminal`** application icon, and you can also drag a folder to the tab bar of an existing window to create a new tab in that folder.

You have now completed the installation of the ODK Application Designer software.

Using ODK Application Designer

ODK Application Designer Overview

This section presents a brief overview of the features of the ODK Application Designer.

The ODK Application Designer is accessed through a **Chrome** browser. Once launched, it opens **Chrome** to display:

This screen has 6 tabs:

- *Preview* (shown above) - used to preview ODK Survey forms and ODK Tables list-, detail- and graph- views (future). Displays these within a user-selected device geometry.
- *Customize* - a visual style and visual theme editor. This editor immediately shows the effects of changes to specific settings in your CSS file. This functionality is undergoing changes and not recommended.
- *XLSX Converter* - converts the XLSX description of a form into a `formDef.json` representation used by ODK Survey.
- *File Browser* - enables browsing of the directory structure that will exist on your Android device so that you can access or view other files (currently necessary for accessing the ODK Tables list- and detail- views).

- *Scan Form Designer* - drag-and-drop mark-sense form designer tool.

Preview

The *Preview* tab (shown above) has several controls:

- *Launch Page* – what webpage to launch in the window below.
- *Purge Database* – during development, if you are adding new fields or changing their data types, you will need to purge the database so that the database structure can be re-generated with the proper fields and data types.
- *Device Dimensions* – what dimensions to make the window below.
- *Back Button* – this is generally only useful when viewing ODK Tables web pages.

Warning: The *Back Button* may cause unpredictable results when used while displaying ODK Survey forms.

The Launch Page opens the ODK Survey Framework Page. This is the `formDef.json` in the Android device's application frameworks folder (`/sdcard/opendatakit/default/config/assets/framework/forms/framework`). The contents of this form are defined by the `framework.xlsx` file in that same directory.

For example, if you click on the household test form, and click the *Follow Link* button on the next screen, the *Household Survey* form is launched, yielding this screen:

You can navigate through forms, enter and exit sub-forms, and save results just like on your Android device.

Note: The development environment does not allow you to submit data to a server. ODK Deploy (currently under development, not yet released) will provide this functionality.

Customize

The *Customize* tab contains the CSS style and theme generator:

Using this tool, you can change background colors, fonts, and other settings affecting the appearance of a form. The changes are reflected immediately in the form shown to the left of the toolbar.

This functionality is under active development and not currently recommended.

XLSX Converter

The *XLSX Converter* tab contains the conversion tool that transforms XLSX files produced by Excel or OpenOffice into the `formDef.json` file used by ODK Survey:

See *ODK XLSX Converter* documentation for more information about this tool.

File Browser

The *File Browser* tab provides a view into what will become the application's directory on the phone.

ODK Scan Form Designer

The *Scan Form Designer* tab presents a drag-and-drop editor for mark-sense form creation.

See *ODK Scan Form Designer* documentation for more information about this tool.

Launching the Application Designer

The ODK Application Designer is both

1. a workspace containing the cohesive interacting set of forms and files you have created and
2. a set of tools used for that development.

Tip: We recommend unzipping and creating a new `Application Designer` directory for each new set of ODK Survey forms, ODK Scan forms, and ODK Tables files that are not intended to be deployed as a cohesive unit. If you need to have several of these sets of forms and files co-resident on the same Android device, you would create different application names for each set. The standard set up uses the default application name (appropriately entitled `default`). To create a new application name, create a folder with that name next to your default app and make sure it is stored on the device in the opendatakit folder. The underlying ODK 2 tools will then keep each of these sets of forms and files isolated from each other.

To launch the application designer, open the **cmd** shortcut or **Terminal** window onto the directory containing `Gruntfile.js` (the unzipped `ODK Application Designer` directory) and type:

```
$ grunt
```

This should automatically open **Chrome** and display the **Preview** tab. When you need to stop the server, return to the **cmd** or **terminal** window where you typed the **grunt** command and press `Ctrl+c`. This will stop the process.

Warning: If you have Parallels or other virtualization software running, it might try to open **Chrome** in this system by default. If so, you should still be able to navigate to http://localhost:8000/index.html.

Warning: If the **Chrome** browser does not open, try opening it yourself and browsing to http://localhost:8000/index.html.

Warning: If the page never times-out, but never loads (it remains blank or constantly spinning), then stop **grunt** and try this command instead:

```
$ grunt --verbose connect:livereload:keepalive
```

This will start **grunt**, but disable the file-change detection mechanisms that automatically reload an HTML page when it or any JavaScript file it uses has been modified. Others have reported that uninstalling **npm** and **node**, and then re-installing them may correct the issue.

Application Designer Directory Structure

There are many folders and files within the `Application Designer` directory. Fortunately, the only ones that are of interest for a non-software-developer are:

- `app/` - folder containing everything that will be pushed to the Android device.
- `Gruntfile.js` - contains the definitions of tasks that push files to the Android device, launch the **Chrome** browser, and pull data and log files off the Android device.

Initially, you will only be concerned with the contents of your `app/` directory – the set of files that are placed on the Android device. As your sophistication grows, you may want to define your own **grunt** tasks to automate repetitive steps in your deployment and device management processes. Adding or modifying tasks is beyond the scope of this document;

please refer to the **grunt** website (see *Getting Started Deployment Architect Guide* for the link to that site).

For completeness, here is the full list of the files and sub-folder in this directory. Again, you generally do not need to be concerned with the contents or specifics of any of these:

- `app/` - folder containing everything that will be pushed to the Android device.
- `devEnv/` - contains the HTML for the 6 tabs of the Application Designer.
- `grunttemplates/` - contains template files used by **grunt** tasks.
- `node_modules/` - contains additional software installed by **npm**, such as external tools used by **grunt**.
- `scanFormDesigner/` - contains the Scan Form Designer tool.
- `test/` - contains tests of the computer-based simulated device environment.
- `themeGenerator/` - contains the HTML and JavaScript for the ODK ThemeGenerator CSS style and theme customization tool (accessed via the **Customize** tab).
- `xlsxconverter/` - contains the HTML and JavaScript for the ODK XLSX Converter tool that converts XLSX form definitions into formDef.json files (accessed via the **XLSX Converter** tab).
- `.bowerrc` - JSON configuration for the **bower** tool.
- `.editorconfig` - when your text editors are configured to use it, enables consistent formatting to files across all contributors to your application design. See EditorConfig.
- `.hgignore` - source code management configuration.
- `.hgtags` - source code management configuration.
- `.jshintrc` - configuration for JSHint - a program that flags suspicious usage in programs written in JavaScript.
- `bower.json` - used to control library management through **bower**. By default, the `.bowerrc` file has been configured to install these libraries in `app/framework/libs/` so that you have access to them when your app is pushed to the phone.
- `deleteDefAndProp.sh` - MacOSX shell script to traverse the relevant parts of the `app/` directory and delete the `definition.csv` and `properties.csv` files.
- `Gruntfile.js` - contains the definitions of tasks that push files to the Android device, launch the **Chrome** browser, and pull data and log files off the Android device.
- `index.html` - the main HTML for the ODK Application Designer web page.
- `macGenConverter.js` - MacOSX command-line wrapper for the XLSX Converter tool (converts a single XLSX file piped into **stdin** into a `formDef.json` on **stdOut**).
- `macGenFormDef.sh` - MacOSX shell script to traverse relevant parts of the `app/` directory and generate `formDef.json` files from XLSX files.

- `package.json` - configuration information for **npm**.
- `README` - description linking back to this document.

The `app/` Folder

Everything in this folder mimics what is on the Android device. The directory looks as follows:

- `config` - user-defined configuration for your application.
- `data` - file attachments, and, on the device, the database.
- `output` - on the device, logging files and exported CSV and media files.
- `system` - files managed by the ODK tools (do not modify).

The `app/config/` Folder

This folder is synced to the device. It contains all of the form and table configuration files and initialization scripts. This is the sub-folder in which you will be primarily working.

This folder contains:

- `assets`
- `tables`

The `app/config/assets/` Folder

- `css/` - contains the common CSS files for ODK Tables detail, list and home screens, and for app forms in ODK Survey (`odk_survey.css`).
- `csv/` - contains the data files to be initially read and loaded into the ODK Survey and Tables databases.
- `fonts/` - contains the fonts used throughout the application.
- `framework/` - contains the `framework.xlsx` and other relevant framework files.
- `img/` - contains the images used throughout the application.
- `js/` - contains JavaScript used by the ODK Tables custom home screen and/or the ODK Survey custom forms list
- `libs/` - contains the various libraries used throughout the application like jQuery and D3.

- `tables.init` - contains the initialization directives for which data (CSV) files should be loaded at initial start-up of the ODK tools.
- `index.html` - the HTML for the ODK Tables custom home screen, if it is enabled in the ODK Tables configuration settings.

The `app/config/tables/` Folder

This folder has a predefined directory structure, but the content is entirely dependent upon the needs of your application.

The zip file for the ODK Application Designer populates this with all the subfolders used by each of the ODK Tables and the ODK Survey demonstration zip files. Ultimately, when you have completed your application design, this folder will contain none of these original folders but would instead contain only the folders which you have created.

Note: Unlike ODK Collect, which stores each submission in a separate file, ODK Survey and ODK Tables store their combined collected submission data in data tables (one row per submission).

ODK Tables can display the contents of a table through one or more custom list views; it can display individual submissions through one or more custom detail views. Graphical views are simply list views in which the data is presented graphically using a library such as D3. All of these custom views are defined here.

ODK Survey, unlike ODK Collect, has the additional flexibility of supporting multiple forms to create, access and update data within a single common data table. This enables creating multi-stage workflows such as initial screenings and follow-ups, or registrations and status-updates (submission data can be editable, or not, based upon the form used at that workflow stage).

To accommodate these various capabilities, the `tables` directory is structured such that individual data tables each have their own directory within the `tables` directory. The table's *table_id* is the name of this sub-directory. When defining a new data table, begin with a form whose form id is the table id.

The `app/config/tables/table_id/` Folder

A canonical `table_id` sub-folder contains:

- `definition.csv` - defines the data columns in this table. Generated when the *form_id* XLSX file underneath this `table_id` is processed by the XLSX Converter.
- `properties.csv` - defines the appearance properties for this table. Example properties are the detail view HTML file name, the list view HTML file name, the default

view type of the table, etc. Generated when the *form_id* XLSX file underneath this `table_id` is processed by the XLSX Converter.

- `forms/` - contains directories for each ODK Survey form that manipulates this table. The names of these sub-directories are the *form_id* values of those forms. Within each sub-directory, there is a `form_id.xlsx` file defining the ODK Survey form and the `formDef.json` generated by the XLSX Converter when it processed that form definition file. If the form has form-specific images or media files, custom CSS, layouts, or prompt types, those files should reside within the form's sub-directory (nested sub-folders are permitted).
- `html/` the custom HTML files for the ODK Tables list and details views of the table's contents.
- `css/` - contains CSS files specific to this table.
- `js/` the JavaScript files needed for the custom ODK Tables HTML list and detail views (found in the `html/` directory).

ODK Scan is currently split in where it stores its configuration for mark-sense forms. The current location for the ODK Scan templates is under `app/config/scan/form_templates` directory. This will likely change and lead to additional sub-directories here.

The `app/data/` Folder

The ODK Application Designer stores user data in this directory. The database itself is in the `webDb` directory. Any data files associated with a row in the database are stored within this folder under the `tables/<table-id>/instances` directory.

The `app/output/` Folder

The ODK Application Designer provides various **`grunt`** tasks to pull files off the Android device. These files include JSON objects for debugging, exported CSVs, and the database itself. The **`grunt`** tasks store these files here. There is also a logging directory which contains logs that are useful for debugging issues.

The `app/system/` Folder

This folder contains the files that the ODK tools depend upon and which are expected to be changed only when different versions of the ODK APKs are released.

> **Warning:** Files in this folder are managed by the ODK tools. If you change any of these files, the tools may detect the change and restore the file when they next start. The

goal is that only the ODK core team should have to modify things in this folder. If you feel you need to modify anything in this directory, please contact us.

The general structure is:

- `js/` - contains JavaScript for the Java to JavaScript interfaces common to both ODK Table and ODK Survey.
- `libs/` - contains 3rd party JavaScript libraries used by ODK Tables and ODK Survey.
- `survey/` - contains JavaScript used by ODK Survey to render forms.
- `tables/` - contains JavaScript used by ODK Tables to render the custom home screen, list, detail, and graphical views created by the Application Designer.
- `tables.deleting` - information related to data deletion
- `tables.pending` - information related to pending data changes
- `index.html` - the generic HTML for all ODK Survey forms.

ODK XLSX Converter

ODK XLSX Converter is a tool, similar to XLSForm, that converts XLSX files (created with **`Excel`**) into ODK Survey definition files that are used by ODK Survey.

Warning: Forms created with XLSX Converter are not compatible with ODK Collect. They only work with ODK Survey.

For example, a spreadsheet like this:

Table 4.3: Example Spreadsheet

type	name	display.prompt.text
begin screen		
text	name	Enter name:
integer	age	Enter age:
end screen		

Will result in a survey like this:

Tip: See `exampleForm.xlsx` (located at the path `app/config/tables/exampleForm/forms/exampleForm/` in the Application Designer) for a more extensive list of examples.

Deployment Architect Guide

Using ODK XLSX Converter

ODK Survey offers a rich set of features that can be seamlessly integrated into a custom form. A lot of the functionality can be implemented solely within an Excel workbook. This guide is designed to help you take advantage of this via a guided tour of example tasks.

- *Creating and Loading a Form into ODK Survey*
- *Creating a Simple Survey Form*
- *Adding Multiple Choice Questions*
- *Using Custom Section Worksheets*
- *Using Calculations*

- *Using Queries*
  - *Linked Tables*
- *Internationalization*
- *More Advanced Branching*
- *Creating a Custom Initial Worksheet*
- *Using Validate*
- *Customizing Prompts*
- *Other Features*

Tip: For a full reference to all the functionality available, see the *ODK XLSX Converter Reference*.

Creating and Loading a Form into ODK Survey

Below are the steps to create a new form from the *exampleForm*:

1. Within the Application Designer's folder, create the following directory structure `app/config/tables/your_table_id/forms/your_table_id/`
2. Copy the `exampleForm.xlsx` from `app/config/tables/exampleForm/forms/exampleForm/` into this new directory.
3. Rename the XLSX file to `your_table_id.xlsx`
4. Edit the XLSX file and on the **settings** worksheet, change the value for *table_id* to *your_table_id*. Then update the display title for the survey and the form version. Save the changes.
5. If you have not already, run **`grunt`** to launch the **`Chrome`** browser and open the Application Designer home page.
6. Navigate to the *XLSX Converter* tab, choose this file to convert it. Once converted, choose *Save to File System* and click *OK* on the 3 pop-ups that alert you to the saving of 3 files to the file system. The three files that are saved are:

- `app/config/tables/your_table_id/definition.csv` – defines the user-defined columns in your table
- `app/config/tables/your_table_id/properties.csv` – defines the appearance and available detail and list view HTML files for the table

- `app/config/tables/your_table_id/forms/your_table_id/formDef.json` – defines the ODK Survey form defined by the XLSX file

7. The first two files are written only if the form id matches the table id. That form and the XLSX file define the data table.
8. Repeat the edit, conversion, and save steps to update the columns in your table and your survey form.
9. Connect your device to your computer with a USB cable.
10. In a separate **command** window, navigate to the Application Designer directory and type:

```
$ grunt adbpush
```

to push the contents of the `app/config` directory to your device.

11. Start ODK Survey. The form should now be available in ODK Survey.

Creating a Simple Survey Form

Typing the following in the **survey** worksheet of a workbook with an appropriate **settings** worksheet will result in a simple survey.

Table 4.4: Creating a Simple Survey Example Form

clause	Condition	type	name	display.prompt.text
		integer	person_age	How old are you?
if	data('person_age') >= 18			
begin screen				
		text	pizza_type	What is your favorite kind of pizza?
		integer	num_slices	How many slices would you like?
end screen				
else				
		note		You are too young to be eating pizza
end if				

The first row contains an empty clause and an empty condition column. Therefore, the display.prompt.text will be shown on the screen, and the resulting integer answer will be stored in the variable person_age.

On the next line there is an if in the clause column and data('person_age') >= 18 in the condition column. If the answer stored in the variable person_age is greater than or equal to 18, the following commands should be done until either an else or an end if tag is reached. Notice the other three columns are left blank.

In the next row, there is a begin screen tag in the clause column. The remaining four columns are left blank. Until an end screen tag is reached in the clause column, all the following questions will be displayed on one screen. In this case, the user will be asked to input their favorite type of pizza and how many slices they would like on the same page, assuming they are 18 or older.

In the next row, there is an else tag. Until end if is reached, anyone who did not satisfy the requirement for the if tag will be asked the following questions. In this case, a note to the user that they are too young to be eating pizza will be displayed.

Note: An important thing to remember when using the clause column is when to open and close new tags. The general rule is that the most recently opened grouping is the first to be closed.

Adding Multiple Choice Questions

There are three types of multiple choice questions supported by ODK Survey:

- select_one
- select_one_with_other
- select_multiple

Multiple choice questions use the values_list column in the **survey** worksheet. The values_list column is what links a multiple choice question to its answer set contained on the **choices** worksheet.

The pizza survey example used earlier can be improved upon with multiple choice options.The resulting **survey** worksheet would look like this:

Table 4.5: Adding Multiple Choice Questions Example Survey Worksheet

clause	Condition	type	values_list	name	display.prompt.text
		select_one	yes_no	person_age	Are you 18 or older?
if	selected(data('person_age'), 'yes')				
begin screen					
		select_multiple	topping_list	pizza_type	What are your favorite kind of pizza toppings (select up to 3)?
		integer		num_slices	How many slices would you like?
end screen					
else					
		note	You are too young to be eating pizza		
end if					

and the corresponding **choices** worksheet would look like this:

Table 4.6: Adding Multiple Choice Questions Example Choices Worksheet

choice_list_name	data_value	display.title.text
yes_no	yes	Yes
yes_no	no	No
topping_list	pepperoni	Pepperoni
topping_list	olives	Black Olives
topping_list	onions	Onions
topping_list	mushroom	Mushrooms
topping_list	pepper	Green Peppers
topping_list	bacon	Canadian Bacon
topping_list	pineapple	Pineapple

Now, instead of typing their age, the user simply selects whether they are older than 18 or not. Furthermore, instead of entering the type of pizza they like, they can select from a list of toppings.

Tip: Because you determine whether a question is select_one or select_multiple from the **survey** worksheet, the same choice set on the **choices** worksheet can be used for both select_one and select_multiple questions.

Using Custom Section Worksheets

Custom section worksheets can be added to a workbook to make the control flow of a survey more readable. We could move all the previous questions about pizza to a new worksheet and name it **Pizza**. Our **survey** worksheet would then look like this:

Table 4.7: Custom Section Worksheets Example

clause	condition	type	values_list	name	display.prompt.text
do	section Pizza				

Tip: When splitting a survey into different sections, it is wise to put a note before each section call with display.prompt.text set to read *Section <name_of_section>*. This is because a do section <name_of_section> call is transparent to the user. Unless the form designer explicitly adds a note, the user will not realize that they entered a section.

Also, after leaving a section, if the user swipes back, the survey will go to the row before the do section call. If the user then swipes forward at this point, the survey will go to the beginning of the section they just completed. It is often beneficial to the user to put a note before entering a section and before leaving a section.

Using Calculations

The **calculates** worksheet is an optional worksheet. It consists of two columns:

- calculation_name: Each row of the **calculates** page represents a function that can be used elsewhere in the workbook by referencing the individual calculation_name.
- calculation: The calculation to be performed.

Note: The calculation column can store any valid JavaScript expression.

Tip: There are also some built in functions for ODK Survey that can be used anywhere in the workbook. See the *Forumla Functions* for more details.

In general, calculations are referenced in the condition column of **survey** worksheets. For example, suppose that on the **survey** page under the variable name *birthday* the user entered their birthday for a question of type date. The **calculates** worksheet might look like this:

Table 4.8: Calculates Worksheet Example

calculation_name	calculation
daysOld	(now().getTime()-new Date(data('birthday')).getTime())/1000/60/60/24
isBirthdayToday	calculates.daysOld()%365 == (now().getTime()/1000/60/60/24)%365

and one of the **survey** worksheets may look like this:

Table 4.9: Calculation Survey Worksheet Example

clause, condition"	type	name	display.prompt.text	
if	calculates.isBirthdayToday()			
		note	happyBirthday	Happy Birthday!
end if				

Notice that the <calculation_name>s do not contain parentheses () at the end of them. However, when referencing them it is always in the format of **`calculates.<calculation_name>()`**.

Tip: Variable names have scope for the entire workbook.

The **calculates** worksheet is handy because it adds readability to a workbook. Instead of having long, complicated JavaScript calculations in the **survey** worksheets, they can be consolidated in one, easy to reference location that allows for reusability. Also notice the consistent use of camelCase for variable naming across the different worksheets.

Using Queries

The **queries** worksheet is an optional worksheet.

For queries that get their data from external sources, the following columns should be used:

- query_name
- query_type

- uri
- callback

For linked_table queries, these columns should be used:

- query_name
- query_type
- linked_table_id
- linked_form_id
- selection
- selectionArgs
- orderBy
- auxillaryHash

Each row of the queries page represents a choice set that can be used by select prompt types in the workbook. In general, query_name is referenced in the values_list column of **survey** worksheets. For example, suppose that on the **survey** page under the variable name region the user is asked to select the region they are from. Then the user is asked to select which country they are from. The choices for the list of countries can be filtered based on the region the user selected. The **queries** worksheet might look like this:

Table 4.10: Queries Worksheet Example

query_name	query_type	uri	callback
regions_csv	csv	"regions.csv"	_.chain(context).pluck('region').uni return {data_value:region, display:{title: {text: region} } }; }).value()
countries.csv	csv	"regions.csv"	_.map(context, function(place){place.data_value = place.country; place.display = {title: {text:place.country} }; return place; })

The data for the queries is coming from the `regions.csv` file that is located in the same directory as the `formDef.json` and specified in the uri column. Thus, the query_type for both queries is csv. A snippet of the `regions.csv` file looks like the following:

Table 4.11: regions.csv

region	country
Africa	Algeria
Africa	Angola
Africa	Benin

Knowing the structure of the `regions.csv` helps in understanding the callback function provided in the callback column. The callback function maps the results from the `regions.csv` file to the data_value and the display.prompt.text fields using JavaScript. The **survey** worksheets may look like this:

Table 4.12: Queries Survey Worksheet Example

clause	condition	type	values_list	name	display.prompt.text	choice_filter
begin screen						
		select_one_dropdown	regions_csv	region	Please select your region:	
		select_one_dropdown	countries_csv	country	Please select your country:	choice_item.region === data('region')
end screen						

The choice_filter in this example ensures that the options for the country question will only be the countries from the previously selected region. Notice that choice_item.region specifies that any country with a corresponding region equal to the answer stored by the region question will be displayed.

The **queries** worksheet is powerful because it allows more flexibility in terms of where data for the survey can reside.

Linked Tables

linked_table is the other use for the **queries** worksheet. linked_table allows you to launch a subform that can edit a different data table. For example, if a survey is dealing with information about households, the user may want to ask questions about the general household but also questions about specific users. linked_table can be used to launch subforms that ask questions about the specific household members. The **survey** worksheet may look like this:

Table 4.13: Linked Table Survey Worksheet Example

clause	condition	type	values_list	name	display.prompt.text	choice_filter
		text		house_id	Input the unique household id:	
		integer		num_members	How many people live in this house?	
		linked_table	members		Add and enter information for the different household members	
		select_one	members	household_head	Who is the household head?	

The **queries** worksheet would look like this:

Table 4.14: Linked Table Query Worksheet Example

query_name	query_type	linked_form_id	linked_table_id	selection	selectionArgs	newRowInitialElementKeyToValueMap
members	linked_table	members_info	house_members	house_id = ?	[opendatakit.getCurrentInstanceId()]	{ house_id: opendatakit.getCurrentInstanceId() }

First the user enters a house id for the house and answers an arbitrary question about its residents. This information is stored in the data table for general household information (specified on the **settings** worksheet under table_id). Then the user reaches a linked_table prompt that uses the values_list members. This is connected to the members query on the **queries** worksheet. It links to a different survey called members_info that edits a different data table. The selection criteria is that the house_id in the house_members data table matches the instanceID of this current household.

Initially this list will be empty since no members have been added. The user can click on the *Create Instance* button to add new people for this household. The house_id will be set automatically for this new member via the newRowInitialElementKeyToValueMap content, which specifies that the house_id field in the linked table should be initialized with the instanceID of the current household.

Note: The selection criteria and its type (in this case, house_id and text) must be added to the model subset of the subform (members_info) in order for selection criteria to be persisted to the database and for the subform to be found by its parent form; the selection criteria cannot filter on session variables since those values are never persisted.

When the user finishes the subform, the screen will return to the same linked_table prompt. At this point, the user can continue adding more users, edit an existing member's info, or go to a different screen.

The values_list for the select_one question prompt in the example above also uses the members query. Instead of being able to launch subforms to edit information about different members, the selection criteria is used to populate a multiple choice question. The answer to the multiple choice question is saved to the general household data table, not the members data table.

Internationalization

Survey offers the ability to display text in different languages. This requires usage of the **settings** worksheet to determine which language to use. However, for any language other than the default language, extra display columns need to be added. For example, if one of the non-default language options was Spanish (2-letter language code "es"), every worksheet

with a display.prompt.text column would also need a display.prompt.text.es column. This is true for all columns that need an alternate language option.

Table 4.15: Internationalization framework_translations Worksheet Example

type	name	display.prompt.text	display.prompt.text.es
text	user_name	What is your name?	¿Cuál es su nombre?
integer	user_age	How old are you?	¿Cuántos años tienes?

The labels used in the buttons and prompts supplied by ODK Survey are defined in the **framework_translations** sheet of the `framework.xlsx` file under `config/assets/framework/forms/framework.xlsx` Simply add your language code and translations to this sheet of this XLSX file and run *XLSXConverter* on it to enable support of your language across all of the built-in buttons and prompts within ODK Survey.

More Advanced Branching

ODK Survey supports situations where the user needs to be in control of which survey or section of a survey they are working on. To do this, the branch_label column is used, as well as the **choices** worksheet. It also utilizes a new question type: user_branch. The following example combines aforementioned surveys and allows the user to decide whether they want to fill out the survey about pizza, or the survey about birthdays.

A choice set needs to be added to the **choices** worksheet with the applicable branching options. The resulting **choices** worksheet would look like this:

Table 4.16: Branching Choices Worksheet Example

choice_list_name	data_value	display.title.text
which_form	pizza_form	Order pizza?
which_form	birthday_form	Is it your birthday?

And the **survey** page would look like this:

Table 4.17: Branching Survey Worksheet Example

branch_label	clause	condition	type	values_list	display.prompt.text
			user_branch	which_form	Choose a survey to fill out
pizza_form					
	do section pizza				
birthday_form					
	do section birthday				

The XLSX file would then have corresponding **section** worksheets called *pizza* and *birthday* that contain the survey examples documented earlier.

Creating a Custom Initial Worksheet

When ODK Survey opens, it displays a list of the different forms available on the device. After the user has selected which type of form to work on, Survey launches the initial worksheet for that particular survey. So far the initial worksheet has not been discussed and if one is not explicitly included in the XLSX file, survey uses this default initial worksheet:

Table 4.18: Custom Initial Worksheet Example

clause	Condition	type	display.prompt.text
if // start	(opendatakit.getCurrentInstanceId() != null)		
		opening	Edit form
do section survey			
		finalize	Save form
else // start			
		instances	Saved instances
end if // start			

This checks to see if an instance of the current form has been selected **`(opendatakit.getCurrentInstanceId() != null)`**. If it has, it opens that form. If not, it displays the instances that the user can edit. This utilizes three new types:

- opening
- finalize

- instances

> **Warning:** When creating a custom initial worksheet, it is very important to include a finalize type. After completing a survey, it is the finalize prompt that lets the user formally finish the survey so that the results can be used.

Using Validate

When users start having more control over which questions they are asked, it can lead to problems if they bypass required prompts. The validate feature allows for the form creator to require form validation in custom places. By default, the form performs a validation during the finalize section of the survey. However, this type of operation can be performed at multiple points throughout the survey on specific questions using the prompt type validate and the column validation_tags.

The following example will collect information from a user in *section1* and *section2* and will prevent completion of *section3* if certain questions have invalid answers.

The **survey** page would look like this:

Table 4.19: Validate Survey Worksheet Example

branch_label	Clause	type	values_list	display.prompt.text
welcome_screen				
		user_branch	which_branch	Choose the section to enter
	goto welcome_screen			
branch1				
		note		Selected Section 1
	do section section1			
		note		Returning from Section 1
	goto welcome_screen			
branch2				
		note		Selected Section 2
	do section section2			
		note		Returning from Section 2
	goto welcome_screen			
branch3				
		note		Selected Section 3
	validate user_info			
	do section section3			
		note		Returning from Section 3
	goto welcome_screen			

The **choices** worksheet would look like this:

Table 4.20: Validate Choices Worksheet Example

choice_list_name	data_value	display.title.text
which_branch	branch1	Do Section 1
which_branch	branch2	Do Section 2
which_branch	branch3	Do Section 3

The **section1** worksheet would look like this:

Table 4.21: Validate Section1 Worksheet Example

type	name	display.prompt.text	required	validation_tags
text	user_name	What is your name?	TRUE	user_info, finalize
integer	user_age	What is your age?	TRUE	user_info, finalize
note		Thank you for answering		

The **section2** worksheet would look like this:

Table 4.22: Validate Section2 Worksheet Example

type	name	display.prompt.text	required	validation_tags
text	occupation	What is your current occupation?	TRUE	user_info, finalize
integer	user_age	How long have you worked at your current job (in years)?	TRUE	finalize
note		Thank you for answering		

If the user selects to do *section 3* on the welcome page, survey will jump to the branch3 branch_label. The first row says to validate user_info. Survey then checks that every question with the validation_tags user_info has been answered satisfactorily. If the questions have been answered correctly, it will go on to the next line (do section *section3*). If not, it will force the user to answer the missing, tagged questions.

The use of many different validation_tags can allow users to update information in the survey as it becomes available and to restrict questions that depend on other information. In general, the validation feature can be used to give users more control over their work while still maintaining a level of order and restriction.

Warning: Like the use of sections and gotos, validate has no user interface. In other words, when a user runs into a validate call, they will have no idea unless Survey finds something wrong with the form. Whenever using sections, gotos, or validates, if the form designer wants the user to be aware of what is happening, a note explicitly informing the user must be added.

Customizing Prompts

There are 3 ways to customize prompts:

- Add additional columns to your XLSX Converter form definitions like inputAttributes to tweak existing prompts.

- If that's too limiting, you can make a custom HTML template by setting the templatePath column. Templates can include `<script>` and:code:*<style>* tags. ODK Survey uses **`handlebars`** templates. **`Handlebars`** has a few built-in helpers for creating conditional templates and templates with repeated components: see their documentation.
- Finally, if you need to parse data from a special type of input or retain some kind of state while your widget is active, you will need to delve into the ODK Survey JavaScript. By providing a `customPromptTypes.js` file in your form directory, you can define **`Backbone`** views that extend the base prompts.

Our HTML page rendering uses a custom database object coupled with **`Backbone`** views to define the event handling, validation, data model interactions, and construction of the rendering context object that is passed to **`Handlebars`**. The **`Handlebars`** templates make use of **`Bootstrap`** framework for UI components.

Other Features

Different surveys and forms can also be entered using the external_link type, the url column, and the url.cell_type column. To access a separate survey stored elsewhere, a local url can be specified in the format: `'?' + opendatakit.getHashString('<relative path to survey>', null)`. Converting the example above to this format would leave the **choices** worksheet looking the same. However, the **survey** worksheet would look as follows:

Table 4.23: External Link Survey Worksheet Example

branch_label	label	condition	type	values_list	display.prompt.text	url	url.cell_type
			user_branch_form	which_form	Choose a survey to fill out		
pizza_form							
			external_link		Open Form	'?' + opendatakit.getHashString('../config/tables/pizza/forms/pizza', null)	formula
	exit section						
birthday_form							
			external_link		Open Form	'?' + opendatakit.getHashString('../config/tables/birthdays/forms/birthdays', null)	formula
	exit section						

ODK XLSX Converter Reference

- *Excel Worksheets*
  - *Survey*
    - *Required Columns*
    - *Optional Columns*
    - *Prompt Types*
  - *Settings*
  - *Properties*
  - *Calculates*
  - *Choices*
  - *Model*

  - *Queries*
  - *User Defined Section*
  - *Custom prompt_types*
  - *column_types*
  - *framework_translations*
  - *common_translations*
  - *table_specific_translations*
- *Built-in Functionality*
  - *Formula Functions*

Excel Worksheets

A workbook is composed of one or more worksheets. XLSX Converter expects the worksheets within a workbook to use the following nomenclature.

Table 4.24: Worksheet Reference Table

Worksheet	Required?	Description
survey	Required	Contains the content and control flow of the survey. It contains the full list of questions and determines the order in which they will be asked. This worksheet can be broken into different section worksheets for ease of use.
settings	Required	Includes such details as the form name and id, as well as the default language.
properties	Optional	Defines the key-value properties that can specify the detail, list view, and other properties to use with this table. This sheet should only be specified in forms whose form_id matches their table_id.
calculates	Optional	Contains the JavaScript formulas that can be used in other worksheets
choices	Optional	Contains the sets of choices for multiple choice questions. Each row

Note: Each worksheet has a set of required and optional columns. For the XLSX workbook to be valid, all entries must have legal values in the required columns. Optional columns can be left blank at any point, and omitted entirely if not used.

Survey

All XLSX Converter form definitions require a **survey** sheet. The **survey** worksheet contains the structure and most of the content of the form. It contains the full list of questions and information about how those questions should be presented. Most rows represent a question; the rest of the rows specify control structures such as screen groups. Blank rows are ignored.

Note: In this document, questions and question types will also be referred to as prompts and prompt types.

There are many prompts available for form development. Some ask the user a question and get a response, but other prompts are simply informational and referring to them as questions is not semantically correct.

Required Columns

A list of the required columns for a **survey** worksheet follows.

Table 4.25: Survey Worksheet Required Columns

Column	Description
type	The prompt type that will be used to display information to the user. Prompt types can also be used to get data from a user.
name	The name of the prompt type. This name will be used throughout the workbook to reference the prompt.
display.prompt	A string token identifying the translation entry that can define the text, audio, image and video to display for this prompt. Alternatively, this column can be omitted and the prompt text can be specified directly via the display.prompt.text column.

Optional Columns

A list of the optional columns that can be incorporated into a **survey** worksheet is below.

Table 4.26: Survey Worksheet Optional Columns

Column	Description
branch_label	Used to identify which part of the survey to branch to when used with a goto clause or user_branch prompt.
calculation	When used with the assign prompt type, assigns a value to a prompt type.
choice_filter	Used to filter the choices of a multiple choice or linked_table prompt.
clause	Used in conjunction with the condition column to manage the control flow of the survey. clause and condition control which questions get asked in what order, if at all. The clause column contains control flow options such as if, and the condition column contains a predicate to determine if action will occur. if statements always require a condition statement. For other clause statements, a blank condition column is assumed to be true. Other commands include begin screen, end screen, and do section.
comments	Never displayed to the user. Used for development purposes to leave comments about the form for future

Prompt Types

The following prompt types are available in ODK Survey.

Table 4.27: Survey Prompt Types

Prompt Type	Description
acknowledge	Used to display a message to the user and have them click a checkbox to acknowledge that they have read the message.
assign	Used for internal assignment of a variable.
audio	Used to capture an audio recording.
barcode	Used to capture a barcode.
date	Uses a date picker widget to capture a date.
datetime	Uses a date time picker widget to capture a date and time.
decimal	Used to display a message to the user and have them enter a decimal.
geopoint	Used to capture a GPS location.
image	Used to capture an image.
integer	Used to display a message to the user and have them enter an integer
linked_table	

Settings

Table 4.28: Settings Worksheet Columns

Column	Description
setting_name	The name of the setting within the form
value	The value for the setting
display.title	A string token identifying the translation entry with the text shown to the user when the (survey) title is displayed. Alternatively, this column can be omitted and this text can be specified directly via the display.title.text column.
display.locale	A string token identifying the translation entry with the text shown to the user when the translation locale is displayed. Alternatively, this column can be omitted and this text can be specified directly via the display.locale.text column.

Available setting_name values that can be used:

Table 4.29: setting_name values

Value	Required?	Description
table_id	Required	The unique id of the table that the form data gets stored in.
survey	Required	Specify the title of the form via content of the display.title.text column. That value will appear as the title to the user.
form_id	Optional	A unique identifier for the form. Default value is the unique id that ODK Survey uses to identify the form.
form_version	Optional	A value used for version control of the form. The recommended format is yearmonthday (for example: 20131212 to say the 12th of December 2013).
<section_name>	Optional	Used with display.title.text to set how the section name will appear to the user on the contents screen.
instance_name	Optional	Used to display the name of saved instances of the form. This must be the name of a

A sample **settings** worksheet might look like this:

Table 4.30: Settings Worksheet Example

setting_name	Value	display.title.text	display.locale.text	display.locale.text.hindi
table_id	sample_form			
form_version	20130819			
survey		Sample Form		
default			English	English (as Hindi name)
hindi			Hindi	Hindi (as Hindi name)

Tip: If the survey has been broken up into multiple worksheets, each worksheet can be assigned its own title by adding a row for it and filling in the display.title.text column.

Tip: In the case of multiple languages, the display.locale.text column determines how the different language options are presented to the user.

Properties

This holds the key-value settings for specifying detail and list views, and other parameters. The columns in this sheet are:

Table 4.31: Properties Worksheet Columns

Column	Description
partition	The class of property to set
aspect	
key	The name of the property to set
type	Valid options: object, array, rowpath, configpath, string, integer, number, boolean
value	The value of the property to set

For example, the following configuration specifies that the default view for the table is the list view (HTML). It also defines the detail view, list view, and map view HTML files. And, for the map view, it defines the color rule to apply to the pins in the map view and the latitude and longitude columns to use in displaying those pins.

Table 4.32: Properties Worksheet Example Table : header-rows: 1

partition	aspect	key	type	value
Table	default	defaultViewType	string	LIST
Table	default	detailViewFileName	string	config/tables/Tea_houses/html/Tea_houses_detail.htm
Table	default	listViewFileName	string	config/tables/Tea_houses/html/Tea_houses_list.html
Table	default	mapListViewFileName	string	config/tables/Tea_houses/html/Tea_houses_list.html
TableMapFragment	default	keyColorRuleType	string	None
TableMapFragment	default	keyMapLatCol	string	Location_latitude
TableMapFragment	default	keyMapLongCol	string	Location_longitude

Calculates

The **calculates** worksheet is an optional worksheet.

Table 4.33: Calculates Worksheet Columns

Column	Description
calculation_name	The name used to reference the calculation in other worksheets.
calculation	The JavaScriptf forumla to be evaluated.

Each row of the **calculates** page represents a function that can be used elsewhere in the workbook by referencing the individual calculation_name. The calculation column can store any valid JavaScript expression. In general,

Note: Calculations are referenced in the condition column of **survey** worksheets.

Tip: There are built in functions for ODK Survey that can be used anywhere in the workbook. See the *Forumla Functions* section for more details.

If a complex calculation is required, you can access the full power of Javascript and the **jquery.js** (that is: `$.some_func(...)`) and **underscore.js** (that is: `_.some_func(...)`) libraries. Internally, the calculate column is wrapped and evaluated as a Javascript function:

```
function() {
    return (YOUR_CALCULATE_COLUMN_CONTENT_HERE);
}
```

You can write your own code to perform a join via defining and invoking an anonymous function in your calculate. Here is an example:

```
(function() {
    var result = "";
    _.each(data('valueListField'), function(element) {
        result = result + ", " + element;
    });
    return result.substring(2);
}) ()
```

This defines a function and then invokes it. The available functions within a calculates expression are the following:

Table 4.34: Available Calculates Functions

Function	Description	Usage
`data(fieldName)`	Retrieve the value stored under this fieldName	`data('myField')`
`metadata(instanceMetadataFieldName)`	Retrieve value stored under this name	`metadata('_group_modify')`
`selected(promptValue, qValue)`	Test whether qValue occurs within a select-multiple	`selected(data('mySelectMultipleField 'myChoiceDataValue')`
`countSelected(promptValue)`	Count the number of selections in a select-multiple	`countSelected(data('mySelectMultiple`
`equivalent(promptValue1, promptValue2, ...)`	Test if values are equivalent	`equivalent(data('promptA'), data('promptB'))`
`not(conditional)`	Negate a condition (equivalent to !(conditional))	`not(data('fieldA') === data('fieldB'))`
`now()`	Return the current time	
`isFinalized()`	Return whether or not the current row is finalized	
`assign(fieldName, value)`	Store value in fieldName and return value.	`(8 + assign('myField', 5))*10`

Additionally, the following functions are also available, but are generally not useful in calculates. They are used within template helper functions (…/`system/survey/js/handlebarsHelpers.js`).

Table 4.35: Template Helper Functions

Function	Description	Usage
`getCurrentLocale()`	Return the currently-active locale	
`localize(locale, displayProperty)`	Localize the given display.xxx text	`localize(getCurrentLocale(), display.hint)`
`width(string)`	Determine the rendered width of a string	
`expandFormDirRelativeUrlPath(content)`	Return url for a file within the form directory.	

And, finally, you can also reference the *opendatakit* object (that is: `opendatakit.some_func(...)`) within these functions (`system/survey/js/opendatakit.js`).

Choices

The **choices** sheet allows you to specify the set of choices for multiple choice prompts.

Table 4.36: Choices Worksheet Columns

Column	Description
choice_list_name	The name used to reference the set of choices. This name must be the same as the values_list in the **survey** worksheet.
data_value	The value that gets stored as the user's response.
display.title	A string token identifying the translation entry with the text shown to the user for this choice value. Alternatively, this column can be omitted and this text can be specified directly via the display.title.text column.
display.title.text	The text that the user sees for this choice.
display.title.image	An image that the user will see associated with a particular choice.

Model

The **model** sheet is an optional sheet that allows you to specify the data model for the table_id specified in the **settings** worksheet.

Table 4.37: Model Worksheet Columns

Column	Description
name	The name of the data field to be used in table_id
type	The type of data that can be put into this data_field of the table.
isSessionVariable	Whether or not this field is a session variable (not persisted – defaults to false).

Many more columns can be specified, including a default column or, as shown in the exampleForm, a default[0] column to initialize the first element (index zero) of a select multiple field. Default values cannot be calculates and must be simple literal values (integers, numbers and strings).

Queries

The **queries** worksheet is an optional sheet that allows you to request data from external sources for use in select prompts. These are some of the things you can do with queries:

- Connect to website APIs.
- Get data from external Android Applications via file content providers.
- Get data from a linked table
- Open CSV files included in the survey's directory.
- Pass key-value maps to linked_table forms when creating or opening that form.

Table 4.38: Queries Worksheet Columns

Column	Description
query_name	The name used to reference the information returned by the query.
query_type	Legal value are ajax, csv, and linked_table. Used to specify the provenance of the query data.
uri	Used by ajax and csv queries. The uri to use for an ajax query or the name of the CSV file to use relative to the location of the `formDef.json` file.
callback	Used by ajax and csv queries. The function that will be used to map the query results to the set of choices for a multiple choice prompt.
linked_table_id	Used by linked_table queries. The table_id used to identify the table that the data will come from. This should match the table_id provided in the **settings** worksheet.
linked_form_id	Used by linked_table queries. The id of the form that will be used to get the results for the linked_table. This value should match the

The two columns newRowInitialElementKeyToValueMap and openRowInitialElementKeyToValueMap allow you to pass information from your originating form into the linked form. The element keys in these maps correspond to the element keys in the linked form (not the current form). These can refer to any of the form's fields; commonly, the values you would pass into the openRowInitialElementKeyToValueMap would refer to session variables. You would typically pass the instanceID of the originating form (that is: `opendatakit.getInstanceID()`) into the linked form when creating it so that you can store that id in a field in that linked table, thereby tying the newly-created row in that table back to the originating form's row.

User Defined Section

A custom named section is essentially a subset of the **survey** worksheet. Thus, all of the columns that were described in the *survey* section are applicable in a custom section worksheet. However, the following worksheet names are reserved and cannot be used to name a custom section worksheet:

- settings
- properties
- choices
- queries
- calculates
- column_types
- prompt_types
- model
- framework_translations
- common_translations
- table_specific_translations

Custom prompt_types

Custom prompts can be created within the survey. The **prompt_types** worksheet can be used to specify the custom prompts so that they will be recognized by Survey.

Table 4.39: prompt_types Worksheet Columns

Column	Description
prompt_type_name	The name that will be used to reference the prompt_type
type	The type of object that will be used to store the data received by the user for this prompt type.

column_types

Custom columns can be used within a workbook that are used to store functions, formulas, and path names. The **column_types** worksheet can be used to specify these custom columns.

Table 4.40: prompt_types Worksheet Columns

Column	Description
column_type_name	The name that will be used to reference the column.
type	The type of information that will be stored in the column (for instance, function, formula, app_path_localized).

framework_translations

The **framework_translations** sheet is only present in the `framework.xlsx file`. It defines the translations for all of the standard prompts provided by the ODK 2 framework.

Table 4.41: framework_translations Worksheet Columns

Column	Description
string_token	The name that will be used string to be translated.
text.<locale>	The value of the translated text string. There can be as many of these columns as you want translated languages (such as text.default, text.gr, text.es).
image.<locale>	The value of the image url fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as image.default, image.gr, image.es).
audio.<locale>	The value of the audio url fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as audio.default, audio.gr, audio.es).
video.<locale>	The value of the videourl fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as video.default, video.gr, video.es).

The locale code should generally be the 2-letter language code, or, if necessary, the *language_COUNTRY* naming used by Android can be used to identify a specific language variant. For example: *en_US*, *en_UK* for US English and UK English, respectively.

common_translations

The **common_translations** sheet is optional. It should only be present in the `framework.xlsx` file. It can be used by application designers to define translations used across multiple forms and web pages in an application.

The format for this sheet is the same as that for the **framework_translations** sheet.

Table 4.42: framework_translations Worksheet Columns

Column	Description
string_token	The name that will be used string to be translated.
text.<locale>	The value of the translated text string. There can be as many of these columns as you want translated languages (such as text.default, text.gr, text.es).
image.<locale>	The value of the image url fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as image.default, image.gr, image.es).
audio.<locale>	The value of the audio url fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as audio.default, audio.gr, audio.es).
video.<locale>	The value of the videourl fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as video.default, video.gr, video.es).

The locale code should generally be the 2-letter language code, or, if necessary, the *language_COUNTRY* naming used by Android can be used to identify a specific language variant. For example: *en_US*, *en_UK* for US English and UK English, respectively.

table_specific_translations

The **table_specific_translations** sheet is optional. It should only be present in the XLSX file whose form_id matches the table_id. It defines translations that are available to all forms and web pages specific to that table id.

Table 4.43: framework_translations Worksheet Columns

Column	Description
string_token	The name that will be used string to be translated.
text.<locale>	The value of the translated text string. There can be as many of these columns as you want translated languages (such as text.default, text.gr, text.es).
image.<locale>	The value of the image url fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as image.default, image.gr, image.es).
audio.<locale>	The value of the audio url fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as audio.default, audio.gr, audio.es).
video.<locale>	The value of the videourl fragment relative to the appName directory for this locale. There can be as many of these columns as you want translated languages (such as video.default, video.gr, video.es).

The locale code should generally be the 2-letter language code, or, if necessary, the *language_COUNTRY* naming used by Android can be used to identify a specific language variant. For example: *en_US*, *en_UK* for US English and UK English, respectively.

Built-in Functionality

The **`jquery`** and **`underscore`** libraries are available when defining calculates expressions.

ODK Survey exposes built-in functionality through formula functions to decrease form development time.

Formula Functions

The following formula functions can be used to simplify calculations or expressions.

Table 4.44: Built in formula functions

Name	Description	Example
assign	Assignment operator that will assign the value to the field and return the value	assign('fieldname', value)
countSelected	Returns the number of items selected from a select_multiple prompt	countSelected(data(‘options’))
data	Returns the value of a field or session variable.	data(‘options’)
equivalent	Check to see if two values are equivalent	equivalent(data(‘option1’), data(‘option2’))
isFinalized	Returns true if this submission is finalized	isFinalized()
localize	Localizes the text passed in.	localize(data('options'))
metadata	Returns a metadata field of this row	metadata(‘_group_read_only’)
not	Negates the argument passed in.	not(selected(data('examples'), 'label_features'))
now		now().getDay()
[illegible]	Returns the current date	[illegible]
selected		selected(data('visited_continents'), [illegible]

And, additionally, the *opendatakit* object is also available for use in calculates expressions.

> **Warning:** The *opendatakit* object contains many useful functions but these should be considered internal methods subject to change. When upgrading, be sure to confirm that the methods you use have not disappeared!

ODK Scan Form Designer

ODK Scan uses the *ODK Scan Form Designer* to create machine-readable forms. Scan is only compatible with forms created in the Scan Form Designer and loaded onto the phone. They can then be chosen among the templates within ODK Scan and successfully processed on the device.

Find the Scan Form Designer inside a tab in the *ODK Application Designer* using **`Google Chrome`**.

> **Warning:** You muse use **`Google Chrome`**; other web browsers are not compatible at this time.

A basic overview of the steps to design a form are:

1. Set the page style
2. Add images and data fields
3. Save form (for future editing) and Export form to ODK Scan app
4. Print form for users to complete by hand

You can begin using the Scan app once you've got a form template and forms with hand-written data entered.

Video tutorials for each of these steps –from designing to exporting your form template– are available as a YouTube playlist.

Once you have exported your completed form you can transfer the form template to the ODK Scan app and begin scanning paper forms. See the *ODK Scan* documentation for more on those next steps.

Deployment Architect Guide

Using ODK Scan Form Designer

Getting Started

After *Launching the Application Designer*, find the Scan Form Designer inside a tab (see the *ODK Application Designer Overview* for a tour of all the tabs).

> **Warning:** You must use **`Google Chrome`**; other web browsers are not compatible at this time.

The Scan Form designer presents a default page, the toolbar across the top of the screen, and Form Properties in the gray editing area surrounding the the page.

Tips for Increased Accuracy

- Adding images, especially high resolution images, to your form will provide increased reference points for Scan to help with aligning the form. Adding labels as images, as opposed to text fields, can help improved the accuracy of your scans.

- Fields stretched across the page are more likely to appear curved or warped in the photo taken by Scan, and the misalignment can lead to recognition errors. Ideally, each field should only be a few inches wide.
- Similarly, large sized numbers can also appear stretched or misaligned; small to medium sized numbers are recommended with 2pt spacing between each digit is recommended.
- Fill-in Bubbles can be slightly more accurate than Check Boxes.
- Ink pens are recommended over pencil when users are completing your printed form.
- You don't need to worry about leaving space for a printable border, Scan will automatically create a border around your form template.
- Currently Scan is capable of reading one-page, one-sided forms, so the Form Designer will only allow you to create one-page, one-sided forms.

Set the Page Style

You must select the page style you want before you add any images or field boxes. You cannot change the page style later.

> **Warning:** If you switch the page style later everything will be cleared to the default (blank!).

Choose your page format from the toolbar by selecting *File* → *Set page style*. The options for page style are:

- Letter portrait
- Letter landscape
- A4 portrait
- A4 landscape
- Legal portrait
- 3x5 index card

Form Properties

In the workspace to the left of the form you are creating is a box titled *Form Properties*. This is were you can tailor each field for style and for establishing how the data will be organized and presented after it is scanned and digitized. The key properties to note at this point are:

- *Name*: An identifier for the ODK tools back end. A name is generated automatically but can be customized if desired. No spaces allowed; if blanks are entered (for example: "Date mo 1" it will be saved with underscores (for example, "Date_mo1"). If desired, the name can be the same as the display text.
- *Display Text*: A label for the field that relates the nature of the data input and will be a reference point in Survey when looking at the data answers after collection (for example: "PolioVaccDate"). If desired, this can be the same as the name. The display text can include spaces if desired.
- *Verify field*: Choose whether the field requires validation by the user reviewing the scan when transcribing in Survey.
- *Order of fields*: Enter the order that the fields will be presented to the person verifying each field of data in Survey. Provide order by listing number, for example: 1, 2, 3.
- Select *Update Field* to apply any changes.

Adding Images

Anchor Images

You'll find that the default starting page of the Form Designer has images in each corner. These anchor images act as fiducial markers, or points of reference for the ODK Scan app when the form is eventually photographed with ODK Scan. Points of reference help the app orient the form so it knows which fields on the paper form correspond to the fields in the digital template. Additionally, any typed text fields that you added to the form will be

viewed as images by the app and give the app additional points of reference to orient the form for processing.

Note: Anchor images are essential for accurate Scan readings

You can customize the anchor images with your own images:

- Delete the preloaded anchor images by *Deleting Field* when the image is selected, and follow the instructions below on how to add new images.
- Each corner's anchor image must be unique, and the higher the resolution the better.

Add Images

1. To begin adding images, you must first be working on the image layer. From the toolbar, select *Edit* → *Images*.
2. Choose the image from your computer by clicking *New Image*. The image will appear in the image workspace area to the right of the form you are editing.

3. Use cursor to select the area of the image you want to use; this can be resized later.
4. *Add Selection*
5. Selected image will be placed in the upper left-hand corner of the editing layer workspace. Drag the center of the image to place it where you want on the form, and the corners of the image to resize it.
6. You can keep adding selections from the same image while in *Image Layer* mode.
7. Return to *Edit* → *Field* to add more fields.

8. If you return to *Edit* → *Image* to add more images, you will see the previously uploaded files in the righthand corner of the workspace. Click on a file to quickly load the image for selection.

Uses for Images

In addition to customizing the anchor images on your form and adding additional points of reference to guide the ODK Scan app, you may also want to use images to:

- Add a logo or picture
- Add tables or charts to the form
- You want to add text without typing it out in the *Text* field of the form designer. This is helpful if you are working off an existing form and do not want to retype all of the text from the form. You can grab images of the text instead and upload it to use in the form designer.

Adding Data Fields

To begin adding data fields, you must first be working on the *Fields* layer by selecting from the toolbar *Edit* → *Fields*.

There are seven different field inputs that are supported by the ODK Scan Form Designer. Two of these field do NOT support digitization:

- *Text Box*
- *Text*

And five allow for automatic digitization

- *QR Code*
- *Checkboxes*
- *Fill-in Bubbles*
- *Number*
- *Formatted Number*

To add a field, select *Add* → *(desired field)*. Once you've added a field, the field will appear in the top left section of the form. You can then drag and drop the field to the placement you want on the form, as well as shrink or expand the field by pulling the corner.

Text Box

This will be a blank field where users will write in information. In the scanning process, text boxes capture an image of what has been written in the box, but they do not automatically digitize the letters.

Note: To digitize a text box, a user will manually transcribe the image of the text box into a text prompt in ODK Survey.

Text

This is one way you add typed text to a form. Text fields are not an input field for users and will not be digitized by scan, but act more as labels for fields that will be automatically digitized. Text fields also help ODK Scan orient the photo of the scanned form to the template file by providing additional points of reference.

Tip: Another way to add typed text to a form is as an image.

QR Code

A matrix barcode that can contained encoded numbers, words, or other data.

When a form with a QR code box is scanned, the ODK Scan App will process any QR code data inside that area. This is designed for a process such as placing a unique patient ID code sticker on a printed form and then using the ODK Scan app to automatically link the encoded data with the other data elements on the form. The only stipulation is that the QR code must fit inside the box whose size you specify in the form designer.

To create a custom QR code, you can use an online QR code generator, such as these example: QR Code Generator or QR Stuff.

Once you have a QR code saved as an image, you can add it to your form like any other image file. See Adding Images for more information.

Checkboxes and Fill-in Bubbles

For ODK Scan, Fill-In Bubbles and Checkboxes have the same functionalities and options; they only vary in how they look.

Note: Fill-in bubble option results in slightly more accurate scan results than similar checkboxes.

With checkboxes or fill-in bubbles there are a few additional elements to consider in *Form Properties*.

Bubble Type

The *Bubble type* field allows you to select how to categorize and count user entries.

- *Tally*: Filled bubbles will be read by ODK Scan as one unit each and will be added up to result in a number value. Each filled bubble/checkbox is one tally mark. (for example, one filled bubble for each child vaccinated).
- *Select one*: User chooses only one answer to the prompt. (for example, Male or Female).
- *Select many*: User chooses all applicable answers. (for example, Reasons for extra care: Low birth weight, family history of infant death, twins…).

Grid Values

Grid Values are the values designated to each bubble or box. The default value for each bubble or box filled in by the user is 1, and you can customize the answers ODK Scan attributes to each box or bubble. For example, if in a grid of one row and two columns, row 1, col 1 is given the value of "yes," when that box is marked by a user in Survey and Tables the digitized answer will be "yes."

Number and Formatted Number

The Number field is to add a number input that does not need any special formatting (for example, it's not a date, decimal, or a number split up by a dash). It is what you should use for things like number of polio vials in stock, age of child, and patient ID number.

The Formatted Number field has an option for digits to be split up by delimiters, allowing you to create a date, decimal, and dashed-number input. This is what you should use for things like date of registration and infant weight, and for anything like a serial number or refrigerator product code where the number is broken up by a dash.

Note: How to Write in Numbers

When a person fills out a number field they will be asked to write in the digits by connecting the appropriate dots in each box. The digits will end up looking like the numbers on a digital clock.

Warning: Scan's accuracy for number digitization is not as high as it is for the other fields. Bubbles and checkboxes have been tested at 99% accuracy in the field, but number accuracy can dip into the 80s or worse depending on form design and field conditions.

If you plan to use numbers in your form, be sure to review the *Tips & Recommendations* section and test your form in field conditions.

Group Options

At the far right of the toolbar is *Group Options*, which allows you to create subforms. With subforms you can link several fields together, useful when wanting to move multiples fields around your form at once and keep them together

1. While holding the `Shift` key, select all the fields you want to group together.
2. From the toolbar, select *Group Options* → *Group Fields*.
3. A dialog box will appear asking to confirm that you want to make a subform. After selecting *Yes*, you will need to name this subform.

If you need to ungroup fields, with the subgroup selected, from the toolbar select *Group Options* → *Ungroup Fields*.

Save & Export the Form

Save Incomplete

If you are working on a form and wish to save it for future editing, go to *File* → *Save Incomplete* to save the `.zip` file to your computer.

Load Incomplete

When you return to continue working on a saved form, go to `File > Load Incomplete` and select the `.zip` from your computer. Make sure it is still in the `.zip` format and is not an unzipped folder.

Warning: Always make sure to SAVE your form this way, even if you are also exporting or saving to file system. This is the ONLY way to reload a form if you want to make changes. The exported file will NOT work if you try to load it back into the form designer.

Save to File System

Once your form is complete, you are ready to generate the machine readable files. Go to *File → Save to File System*. Give the file the name you will want to see it called in the app and in Survey and Tables, as you will not be able to change this name later.

This will generate the JSON template file, JPG form photo, and all other files necessary for the Scan app to read and process your forms. It will save them to the application file system, which can be pushed to the device using **`Grunt`** with the typical command for pushing your app to your device (performed inside the `Application Designer` directory:

```
$ grunt adbpush
```

> **Warning:** Saving to the file system does NOT save a version that can be edited later. Please use the *Save Incomplete* function to get an editable file.

Export Completed Form

If you would prefer to export your Scan machine readable files externally from the file system, you can use this option. Go to *File* → *Export Completed Form*. Give the export file the name you will want to see it called in the app and in Survey and Tables, as you will not be able to change this name later.

This will give you a `.zip` file that you can unzip and use to print hard copies of your form and transfer your form `.json` template to the ODK Scan App.

Note: This step is NOT necessary. Most people will use the "Save to File System" option.

Printing the Form

After you have saved and exported your form, print hard copies for your user to complete.

1. From the location you saved it on your computer, unzip the exported file.
2. Within the folder, find and open the file called form `.jpg`. This is the image of the form that you created in the Form Designer is the form you will print to hard copy.

example_form	--	Folder	Today, 12:57 PM
form.jpg	885 KB	JPEG image	Today, 12:57 PM
scan_EHN_r...formDef.json	27 KB	JSON	Today, 12:57 PM
scan_EHN_r...0a_main.xlsx	8 KB	Spreadsheet	Today, 12:57 PM
scan_registe...formDef.json	58 KB	JSON	Today, 12:57 PM
scan_registe...ient_row.xlsx	9 KB	Spreadsheet	Today, 12:57 PM
template.json	111 KB	JSON	Today, 12:57 PM

3. Print the entire image on one page. Black and white is fine even for forms that were created with colored elements.

Tips & Recommendations

General

- Use only **`Google Chrome`** to access the form designer! Other browsers are not compatible and may cause you to lose the form you're working on.
- Make sure your browser zoom is set to 100%. Zooming out can cause the data fields to appear weird on the form.
- **Do not refresh your browser without first saving your form** – the form will be reset to the default blank form.
- The *Copy* function, can be an easy shortcut if you need to create multiples of the same field. This could be useful, for example, if on your form you want to collect the date of birth for each child in the family, or need to create multiple entries for dates of treatment.
- With the field you want to copy selected, go to *Copy* on the toolbar, and the new field will appear in the top left of the form. Edit any of the *Form Properties* as needed.
- Grouping fields together can be a shortcut when needing to move multiple fields around as you're working on your form; instead of moving them one at a time.
- You can both *Delete* and *Undo Delete* for fields and images from the toolbar.

Design Considerations

- Currently Scan is capable of reading one-page, one-sided forms, so the Form Designer will only allow you to create one-page, one-sided forms.
- Numbers left blank will be recognized by Scan as "" (the empty string).
- Therefore, if for instance you have a field that can have a range in the number of digits (for example, like Patient ID Numbers where one patient's ID could be 5 digits long, and another's 7 digits) create a text field to give your user instructions to leave any blank digits at the front of the field, so that those blanks will not not alter the final value interpreted by Scan.
- Since Scan cannot digitize handwriting, and text will have to be manually typed in when verifying the data set, if the form you are basing your template on is text heavy think creatively and strategically about the ways you can use bubbles or checkboxes instead.

- For example, instead of asking users to write in their symptoms, you can provide bubbles for the most common symptoms, and leave a Text Box for anything not listed.
- Repeat formatting for forms with multiple sections to make it as easy as possible for those writing in information to navigate the fields and the form. For example, place labels in the same position for each field, group subsections close together and create borders around them, and so on.
- Think through the order that users will be collecting information and try to best replicate that in the order that fields are presented on the form.
- For example, if the person completing the form will ask about the child's age before asking about the vaccines they have had (or if you want them to ask about age first), place the number field for age earlier in the form's progression than fields for the vaccines.
- Be strategic about when using fill-in bubbles or checkboxes. To not confuse your user, it is best to use just one type on the form. Alternatively, you can use both to signal the different types of responses that can be given; for example, use fill-in bubbles for all of your *select one* questions, and checkboxes for *select many*, to signify to your user that they are being asked a different type of question.
- Fields by default are created with borders. In the *Form Properties* box you can change the thickness of borders, number of borders, as well as the margins surrounding the fields.
- Use the arrow keys on your keyboard to move selected fields more exactly.
- You can align fields relative to each other by holding down `Shift` to select multiple fields at once, and then go to *Align Field* to select the alignment you want for the selected fields.
- Using the *Change Position* function, located on the toolbar, if fields are placed close enough that they overlap, by sending one field forward or backward, you can overlay them to best fit your form.

4.9 ODK Cloud Endpoints

ODK Cloud Endpoints are servers that communicate with the ODK 2 Android applications. They implement the ODK 2 REST Protocol.

There are currently two options for Cloud Endpoints to communicate with ODK 2 tools.

- *ODK Sync Endpoint* - Supports the full ODK 2 REST Protocol
- *ODK Aggregate Tables Extension* - Supports the majority of the ODK 2 REST Protocol; however, is missing group permission filtering support.

4.9.1 System Administrator Guide

ODK Sync Endpoint

ODK Sync Endpoint is an implementation of *ODK Cloud Endpoints*. It runs a server inside a **`Docker`** container that implements the ODK 2 REST Protocol.

It communicates with your ODK 2 Android applications to synchronize your data and application files.

Authentication

ODK Sync Endpoint does not store user information in its own database, instead it integrates with an *LDAP* directory or an *Active Directory*. That directory is then used to authenticate users and obtain user roles.

Note: As a consequence of the integration, Basic Authentication is the only supported authentication method.

ODK Sync Endpoint prerequisites

You must have **`Docker 17.06.1`** or newer, and be running in *Swarm Mode*. Follow these links for detailed instructions on installing **`Docker`** and enabling Swarm Mode.

- Docker
- Swarm Mode

ODK Sync Endpoint Setup

ODK Sync Endpoint requires a database and a *LDAP* directory, you could follow the instructions and deploy all three components together or supply your own database and/or *LDAP* directory.

Note: All of the following command should be run on your server

---

Setup instructions:

1. Choose a directory to store you endpoint in. In that directory, run:

```
$ git clone https://github.com/opendatakit/sync-endpoint-default-setup
```

2. Navigate into the the "sync-endpoint-default-setup" directory
3. Checkout the sync-endpoint code by running:

```
$ git clone https://github.com/opendatakit/sync-endpoint
```

3. Navigate into the sync-endpoint directory. Most likely

```
$ cd sync-endpoint
```

4. Build sync endpoint by running the following: (NOTE: you will need Apache Maven installed >= 3.3.3)

```
$ mvn clean install
```

5. Navigate back to the parent "sync-endpoint-default-setup" directory.
6. In the "sync-endpoint-default-setup" directory run:

```
$ docker build --pull -t odk/sync-web-ui https://github.com/
↪opendatakit/sync-endpoint-web-ui.git
```

7. In the "sync-endpoint-default-setup" cloned repository run:

```
$ docker build --pull -t odk/db-bootstrap db-bootstrap
```

8. In the "sync-endpoint-default-setup" cloned repository run:

```
$ docker build --pull -t odk/openldap openldap
```

9. In the "sync-endpoint-default-setup" cloned repository run:

```
$ docker build --pull -t odk/phpldapadmin phpldapadmin
```

10. Enter your hostname in the `security.server.hostname` field in the `security.properties` file.
11. If you're not using the standard ports (80 for *HTTP* and 443 for *HTTPS*) enter the ports you're using in the `security.server.port` and `security.server.securePort` fields in the `security.properties`. Then edit the **ports** section under the **sync** section in `docker-compose.yml` to be `YOUR_PORT:8080`.

Note: It is important that the right side of the colon stays as 8080. This is the internal port that the web server is looking for.

---

12. If you're using your own *LDAP* directory or database, continue with the instructions:

- *Custom database instructions*
- *Custom LDAP instructions*

13. In the cloned repository:

```
$ docker stack deploy -c docker-compose.yml syncldap
```

14. The server takes about 30s to start, then it will be running at http://127.0.0.1.
15. See the *LDAP section* for instructions on configuring users and groups.

Custom database

1. If you haven't followed the *common instructions*, start with those.
2. Remove the *db* and *db-bootstrap* sections in `docker-compose.yml`.
3. Modify `jdbc.properties` to match your database. Supported database systems are **`PostgreSQL`**, **`MySQL`** and **`Microsoft SQL Server`**. Sample config for each type of database can be found on Github.
4. Modify `sync.env` to match your database
5. In the cloned repository,

```
$ docker stack deploy -c docker-compose.yml syncldap
```

6. The server takes about 30s to start, then it will be running at http://127.0.0.1.

Custom LDAP directory

1. If you haven't followed the *common instructions*, start with those.
2. OPTIONAL: If your LDAP directory uses a certificate that was signed by a self-signed CA,
   1. Make the public key of the CA available to ODK Sync Endpoint with this command.

```
$ docker config create org.opendatakit.sync.ldapcert PATH_TO_
↪CERT
```

2. Uncomment the relevant lines in the *configs* section in `docker-compose.yml` and the *configs* section under the *sync* section in `docker-compose.yml`.

3. Remove the *ldap-service* and *phpldapadmin* sections in `docker-compose.yml`.

4. Modify the relevant sections in `security.properties` to match your LDAP directory. Further instructions are in the file.

Note: The default configuration does not use ldaps or StartTLS because the LDAP directory communicates with the ODK Sync Endpoint over a secure overlay network. You should use ldaps or StartTLS to communicate with your LDAP directory.

---

5. In the cloned repository:

```
$ docker stack deploy -c docker-compose.yml syncldap
```

6. The server takes about 30s to start, then it will be running at http://127.0.0.1.

Stopping ODK Sync Endpoint

1. Run:

```
$ docker stack rm syncldap
```

2. OPTIONAL: If you want to remove the volumes as well,

   - Linux/macOS:

```
$ docker volume rm $(docker volume ls -f "label=com.docker.
↪stack.namespace=syncldap" -q)
```

   - Windows:

```
$ docker volume rm (docker volume ls -f "label=com.docker.
↪stack.namespace=syncldap" -q)
```

LDAP

- The default admin account is *cn=admin,dc=example,dc=org*.

- The default password is *admin* - it can be changed with the *LDAP_ADMIN_PASSWORD* environment variable in `ldap.env`
- The default readonly account is *cn=readonly,dc=example,dc=org*.
- The default password is *readonly* - it can be changed with the *LDAP_READONLY_USER_PASSWORD* environment variable in `ldap.env`. This account is used by the Sync Endpoint to retrieve user information.

The LDAP directory that you deployed with the instructions above is an **OpenLDAP** server. In addition to the directory, a **phpLDAPadmin** server is also deployed to help you configure the directory.

If you'd prefer to use the **OpenLDAP** command line utilities, they're installed in the OpenLDAP container. These tools are accessible with this command:

- Linux/macOS:

```
$ docker exec $(docker ps -f "label=com.docker.swarm.service.
↪name=syncldap_ldap-service" --format '{{.ID}}') LDAPTOOL ARGS
```

- Windows:

```
$ docker exec (docker ps -f "label=com.docker.swarm.service.
↪name=syncldap_ldap-service" --format '{{.ID}}') LDAPTOOL ARGS
```

Note: The phpLDAPadmin server listens on port 40000, it is important that you do not expose this port to the internet.

The following guides assume that you're using **phpLDAPadmin**.

Creating users

1. Click: *login* on the right and login as *admin*.
2. Expand the tree view on the right until you see *ou=people*.
3. Click on *ou=people* and choose *Create a child entry*.
4. Choose the *Generic: User Account* template.
5. Fill out the form and click *Create Object*.
6. Assign users to groups with *these instructions*.

Creating groups

1. Click: *login* on the right and login as *admin*.
2. Expand the tree view on the right until you see *ou=groups*.
3. Click on *ou=default_prefix* and choose *Create a child entry*.
4. Choose the *Generic: Posix Group* template.
5. Fill out the form and click *Create Object*.

Note: The group name must start with the group prefix, in this case the group prefix is *default_prefix* so for example: *default_prefix my-new-group*

---

6. Assign users to groups with *these instructions*.

Assigning users to groups

1. Click: *login* on the right and login as *admin*.
2. Expand the tree view on the right until you see *ou=default_prefix*, then expand *ou=default_prefix*.
3. This list is all the groups under *ou=default_prefix*.
4. Click on the group that you want to assign users to.
5. A few groups are created when the LDAP server is brought up, refer to *Data Permission Filters* for descriptions of these groups.
6. If the *memberUid* section is not present:
   (a) Choose *Add new attribute*.
   (b) Choose *memberUid* from the dropdown, then enter *uid* of the user you want to assign.
   (c) Click *Update Object* at the bottom to update.
7. If the *memberUid* section is present,

1. Navigate to the *memberUid* section.
2. Click modify group members to manage members.

HTTPS

1. Store your certificate public key in a **`Docker`** config with this command:

```
$ docker config create example.com.fullchain.pem PATH_TO_PUBLIC_KEY
```

2. Store your certificate private key in a **Docker** secret with this command:

```
$ docker secret create examepl.com.privkey.pem PATH_TO_PRIVATE_KEY
```

3. Modify the *configs* section and *secrets* section in *docker-compose.yml* to include name of the **Docker** config and **Docker** secret created above.
4. Uncomment the relevant lines in the *nginx* section in *docker-compose.yml*.

ODK Aggregate Tables Extension

ODK Aggregate Tables Extensions enable the ODK 2 tools to share data via bi-directional synchronization with a central ODK Aggregate server.

The ODK 2 REST Protocol is compatible with ODK Aggregate v1.4.15. The sync protocol has been augmented to cache the user's permissions on the device and, for super-users or administrators, to cache the full set of users and all of their permissions (so that the super-user and/or administrator can assign rows to particular individuals).

Server Setup

First you'll have to install ODK Aggregate v1.4.15 to a server (see Installing Aggregate).

1. Install ODK Aggregate v1.4.15 to a server.
2. Log onto your ODK Aggregate v1.4.15 instance.
3. Go to the *Site Admin* → *Preferences* page.
4. Check the checkbox for *ODK Tables Synchronization Functionality*.
5. Go to the *Site Admin* → *Permissions* page.
6. Add ODK Aggregate usernames by typing one or more users' e-mail addresses into the text area and clicking *Add User*.
7. If you have created an ODK Aggregate username, be sure to *Change Password* on that account to set the initial password for the account.
8. Grant these users the *Synchronize Tables* permissions.
9. Select at least one user to be the administrator and grant them *Administer Tables* permissions. This user will have the ability to *Reset App Server* from the Android device and add or remove tables and configuration files on the server. This is the equivalent of the Form Manager permissions in ODK 1.x deployments.
10. Click *Save Changes*. These changes will not take effect until you do!

Changing the AppName

ODK Aggregate is configured by default to use the **default** application name. To change the name, go to the *Site Admin* → *Preferences* screen and click the *Change ODK 2 App Name* button, and enter a new application name. For example, the https://opendatakit-surveydemo.appspot.com server is configured with *survey* as its application name.

Note: The ODK 2 tools are designed to support multiple, independent, ODK 2 applications running on the Android device. Each of the tools has the ability to run in the context of either a default application name, or a specified application name.

By default, all the ODK 2 tools run under the default application name. Application names correspond to the name of the directory under `/sdcard/opendatakit` where the data files for that application are stored.

When you run ODK Services from within ODK Survey, the ODK Survey tool informs ODK Services to run in the context of the application name under which the ODK Survey tool is running. When ODK Services then interacts with ODK Aggregate, it reports that application name to the server. The server must be configured with exactly the same application name or it will reject the requests from ODK Services. This also applies when launching ODK Services from within ODK Tables.

Using Device Synchronization

For more information on syncing, see *ODK Services Syncing*.

4.10 ODK Suitcase

ODK Suitcase is a cross-platform tool that provides access to data on an ODK Cloud Endpoint from a personal computer.

Data downloaded from an *ODK Cloud Endpoints* is stored as spreadsheets in CSV format. This format is compatible with most spreadsheet software, for example **`Excel`** or **`Numbers`**. Once downloaded, the spreadsheets will be available for offline viewing. Similarly, data to be uploaded to an ODK Cloud Endpoint must be stored in a properly formatted csv file.

4.10.1 Deployment Architect Guide

Installing ODK Suitcase

Prerequisites

1. Set up an *ODK Cloud Endpoints*

   Note: Ensure you are using a compatible Cloud Endpoint from the same revision.

2. Make sure `Java` 7 or higher is installed on the computer you plan to use. If it is not, download and install it.

Installing ODK Suitcase

1. Navigate to https://github.com/opendatakit/suitcase/releases/latest and download the latest `ODK Suitcase.jar` file.
2. Double click the file to start. If that fails, try running:

```
$ java -jar path/to/jar
```

3. Alternatively you can use command line operation. For help on the command line interface type:

```
$ java -jar path/to/jar --help
```

Using ODK Suitcase

Graphical Interface

If your ODK Cloud Endpoint allows for anonymous access then you can leave the *username* and *password* fields blank. Otherwise, please specify an ODK Cloud Endpoint username and password with sufficient permissions.

By default ODK Suitcase creates a `Download` directory where the ODK Suitcase jar file is located and saves data in that directory. To specify a different directory for ODK Suitcase to store downloaded data in, click on the … button.

ODK Suitcase provides three options to customize the CSV file.

- Download attachments:

  - If this option is selected, ODK Suitcase will download all attachments from the given table and the CSV generated will contain hyperlinks to the local files.
  - If this option is not selected, the CSV generated will contain hyperlink to the given ODK Cloud Endpoint.
- Apply Scan formatting:
  - When this option is selected, ODK Suitcase will optimize the CSV by replacing certain columns added by ODK Scan.
- Extra metadata columns
  - When this option is selected, two more columns will be included in the CSV, create_user and last_update_user.

Command Line Interface (CLI)

ODK Suitcase also provides a command line interface that can be easily called by scripts and other programs. The CLI has the same features as the graphical user interface. CSV files produced by the two interfaces should also be identical.

For a list of all available options, open command prompt/power shell or terminal. Type:

```
$ java -jar path/to/jar.jar --help
```

Combine the individual commands described in the help to perform the actions needed. Examples are as follows.

- To download CSV of table *table_id* from app *default* with attachments as an anonymous user to the `default` directory.

```
$ java -jar suitcase.jar -download -a -cloudEndpointUrl "https://your-
↪endpoint-server.com" -appId "default" -tableId "table_id"
```

- To download CSV of table *table_id* from app *default* with attachments with username *user* and password *pass* to:file:‘ ~/Desktop‘:

```
$ java -jar suitcase.jar -download -a -cloudEndpointUrl "https://your-
↪endpoint-server.com" -appId "default" -tableId "table_id" -username "user
↪" -password "pass" -path "~/Desktop"
```

To script the CLI, write the commands you would like to execute in a scripting language (for example, Bash, Batch, Python, Ruby) and use a scheduler (such as Cron or Windows Task Scheduler) to schedule the tasks. To skip over ODK Suitcase's prompts to overwrite, pass `-f` as an argument to ODK Suitcase.

4.11 Building an Application

This will walk you through the steps of building a Data Management Application from scratch. The goal is to start with an empty folder and show you the necessary steps to create a working application that runs on you Android device.

Note: This section covers topics useful to Deployment Architects. A Deployment Architect is an author of a data management application or a consumer of collected data. This person might create forms and edit Javascript on their computer to deploy to the Android device. Or they might download data from the server and use Excel to perform analysis. Examples include technical staff and data analytics staff.

Other perspective definitions can be found *here*.

4.11.1 Prerequisites

You will need to install:

- *ODK Application Designer*
- *ODK Services*
- *ODK Survey*
- *ODK Tables*

Before getting started, be sure you have familiarized yourself with the ODK 2 platform. The *Getting Started User Guide* and *Getting Started Deployment Architect Guide* guides are a good place to start. The *ODK Survey: Sample Application* and *ODK Tables: Sample Application* are also good reference points.

4.11.2 Cleaning App Designer

Your freshly installed copy of Application Designer comes with lots of example forms, tables, and configuration. This is useful for learning the tools and as references when building our application, but the files should be cleaned before building your own application.

Enter your Application Designer directory, navigate to `app/config/` and delete everything inside the directory.

4.11.3 ODK Survey: Designing a Form

When creating a new form, the appropriate directory structure must be created. Once this directory structure is in place, an `.xlsx` form can be created. From this `.xlsx` form, a `formDef.json` file will be generated using the XLSX Converter. This `formDef.json`, in the appropriate directory, is what the system will use to create the Survey form.

Creating the Directory Structure

New forms must be placed under the `app/config/tables/` directory as described in the *The app/config/tables/ Folder* section. Given a form with the name *formId*, it will have a *tableId* of the same name unless you explicitly specify otherwise. The directory structure that should be created is `app/config/tables/tableId/forms/formId` (where, under many circumstances, the value for *tableId* will be the same as the value for *formId*). To get started, for Windows open a:program:*cmd* window within your `Application Designer` folder (click the **`cmd`** shortcut you created earlier), and for Mac/Unix open a **`terminal`** window within your `Application Designer` folder. Type:

```
$ grunt addtable:tableId
```

Where tableId is the name of your new form and table. For example, to create a census form, type:

```
$ grunt addtable:census
```

This will create the required directory structure for an individual table, including the forms directory. It also created basic HTML and JavaScript files, which will be covered later.

Navigate into the forms directory (`app/config/tables/census/forms/` in our example), and create a directory with the form ID as its name. For our example, create a `app/config/tables/census/forms/census` directory. Within that directory, ODK Survey expects to find the `formDef.json` that defines the form.

Tip: We recommend placing the `.xlsx` file used to generate that `formDef.json` in this folder as well. Survey will not use this file, but it is a useful reference and provides an easy to remember storage location in case the form needs to be updated in the future.

Any custom screen, prompt templates, or other media related to the form should be also placed in this directory (or in a sub-directory).

Creating an `xlsx` Form

With the proper directory structure in place, you can now create your form. The *ODK XLSX Converter* documentation extensively details the full range of options, settings, and features available when creating a form. For this basic example, follow these instructions:

1. Create a new file `census.xlsx` inside the `app/config/tables/census/forms/census` folder created in the previous section.
2. Create a *settings* worksheet. This sheet holds general settings for the form. Create the following headers:

- setting_name: has defined options, such as form_id.
- value: the value of the named setting.
- display.title.text: the text shown to the user inside Survey.

3. Create the following rows:

Table 4.45: *settings* worksheet

setting_name	value	display.title.text
form_id	census	
form_version	20180101	
table_id	census	
survey		Census Form

4. Create a *survey* worksheet. This sheet defines the questions and flow of your form. Create the following headers:

- type: the prompt type.
- values_list: the name of the list of choices for a multiple choice question.
- name: the variable name.
- display.promp.text: the question the user will see in Survey

5. Create the following rows:

Table 4.46: *survey* worksheet

type	values_list	name	display.prompt.text
text		name	What is your name?
select_one	yesno	isAdult	Are you 18 years or older?

6. Create a *choices* worksheet. This sheet contains the lists of responses you define for your multiple choice questions. Add the following headers:

- choice_list_name: the group name for all the responses in a choice set

- data_value: the data value to be selected
- display.title.text: the text the user will see to select this value

7. Create the following rows:

Table 4.47: *choices* worksheet

choice_list_name	data_value	display.title.text
yesno	y	Yes
yesno	n	No

With this `.xlsx` file you've created a simple Survey form that will ask the user to type in their name and respond whether they are 18 years old or not. This form will be titled *Census* and it will write to a table in the database with table ID *census*.

Creating `framework.xlsx`

The `framework.xlsx` file is central to the structure of the Application Designer. It defines which forms exist. It has no persisted data. In this case, it only presents a list of forms and allows you to open them.

1. Create the following directories: `config/assets/framework/forms/`.
2. Inside that folder, create `framework.xlsx`
3. Create an *initial* worksheet. Add header: clause and value do section survey.

Table 4.48: *initial* worksheet

clause
do section survey

4. Create a *settings* worksheet. Add the same headers: setting_name, value, display.title.text.
5. Fill in the following rows:

Table 4.49: *settings* worksheet

setting_name	value	display.title.text
table_id	framework	
form_version	20180101	
form_id	framework	
survey		Common JavaScript Framework

6. Create a *framework_translations* sheet. This sheet allows you to translate or customize the text displayed in buttons, messages, and other system text. Translations for your form would be specified in its own *translations* sheet

in its `.xlsx` file. For now, copy the string_token and text.default columns from one of the example `framework.xlsx` files provided with the default Application Designer.

7. Create a *choices* sheet. Add the same headers: choice_list_name, data_value, display.title.text.
8. Add the following row:

Table 4.50: *choices* worksheet

choice_list_name	data_value	display.title.text
test_forms	census	Census Form

9. Create a *survey* sheet. Add the headers: branch_label, url, clause, condition, type, values_list, display.prompt.text.
10. Add the following rows. They tell the software what to do if you're previewing in **`Chrome`**.

Note: This is only tested and expected to work in **`Chrome`** and not other browsers like **`Firefox`**, **`Safari`**, or **`Edge`**.

Table 4.51: *survey* worksheet

branch_label	url	clause	condition	type	values_list	display.prompt.text
		if	opendatakit.getPlatformInfo().container == "Chrome"			
				user_branch	test_forms	Choose a test form
		else				
				note		This is the default form.
		end if				
		exit section				
census						
	'?' + odkSurvey.getHashString('census')			external_link		Open form
		exit section				

Updating `framework.xlsx`

To add another new form to an existing `framework.xlsx` file, take the following steps.

Note: These steps are not part of the running example. They are provided here for reference.

Assuming you have created a `testForm.xlsx`, the appropriate directory structures for `testForm.xlsx`, and then properly generated and saved the `formDef.json`, the following lines would need to be added into the `framework.xlsx` *survey* worksheet.

Table 4.52: Example Framework Survey Worksheet

branch_label	label	clause	condition	type	values_list	display.text	display.hint
testForm							
	"?' + opendatakit.getHashString('testForm')			external_link		Open form	
		exit section					

The following changes will also need to be made to the `framework.xlsx` **choices worksheet**

Table 4.53: Example Framework Choices Worksheet

choice_list_name	data_value	display.text
test_forms	testForm	testForm

The changes to the *choices* sheet add the *testForm* form as one of the choices that is shown in the user_branch prompt (a user-directed branching prompt type). The changes on the *survey* sheet add a branch label, testForm, that matches the data_value from the *choices* sheet (this branch label will be jumped to if the user selects the testForm selection on the user_branch screen). The new branch label then renders an external_link prompt type that has the necessary arguments to open the *testForm*.

Generating `formDef.json`

Once you have a saved your `.xlsx` file, you can use the XLSX Converter to create a `formDef.json`. Make sure your Application Designer is running (see *Launching the Application Designer*) and navigate to the *XLSX Converter* tab. Drag the `.xlsx` form or select it with the *Choose File* button and use the *Save to File System* button to save the form definition file back to the file system.

For the ongoing example, convert the `app/config/assets/framework.xlsx` using the instructions above. Then repeat this process with `app/config/tables/census/forms/census/census.xlsx`

> **Warning:** The *Save to File System* button uses the *form_id* and *table_id* within the `.xlsx` file to identify where to write the `formDef.json` file. If you have copied the `.xlsx` file from some other location, and forgot to edit it, it may update back to that older

location! If the *form_id* is equal to the *table_id*, two additional files are written that define the table's user data fields and that define the key-value properties for the table.

Once you have made these changes and used XLSX Converter on the `framework.xlsx` file to update the `app/config/assets/framework/forms/framework/formDef.json` file, you should see your new form show up in the *Preview* tab of the Application Designer. Clicking on that should open your form.

Tip: If you don't see your form in the *Preview*, try refreshing your browser.

Tip: You can also convert your forms with the **`Grunt`** command:

```
grunt xlsx-convert-all
```

Debugging your Survey

The XLSX Converter should report most problems with your survey.

If the form is not being rendered correctly but your survey generates a `formDef.json` without an error, first try purging the database (dropping all the existing data tables) using the *Purge Database* button on the *Preview* tab. You will typically need to purge the database whenever you add or remove fields from your form or change their data type.

If that does not resolve the issue, try stopping the **`grunt`** command (on Windows, `Control-C` should produce a prompt asking to confirm whether to stop or not. On Mac, `Control-C` kill the process with no prompt.), and re-running it. **`Grunt`** can sometimes get overwhelmed with changes and stop working. After restarting, test your form.

If there are other problems, the contents of the JavaScript Console will be helpful to the ODK core team for debugging. Open the JavaScript Console by clicking the icon with the three bars in the top right, select *More Tools*, select *Developer Tools*, and then select the *Console* tab. Select all of the debugging output, then copy it, save it to a file, and post it to the ODK Forum or create a ticket on the Github Issue Tracker.

Moving Files To The Device

Note: You must have USB debugging enabled on your device in order to perform this step. See these instructions for help.

In order to see these changes on an Android device, you must first have ODK Survey installed on your device. Then:

1. Connect the device to your computer via a USB cable
2. Open a **cmd** or **terminal** window within the *Application Designer* directory (the one containing `Gruntfile.js`), as described in the *Application Designer Directory Structure* documentation.
3. Type:

```
$ grunt adbpush
```

Note: If it gives you an error, you may need to run `grunt adbpush -f` to force it.

Note: If you do not see the form, you may need to *reset the configuration*.

This will copy all of the files under config onto your device. You should then be able to launch ODK Survey, and it will display your form in its list of forms. Click the form to open it.

More **grunt** commands can be found in *Pushing and Pulling Files*.

4.11.4 ODK Tables: Designing a Custom View

One of the most powerful aspects of ODK Tables is its ability to run HTML and JavaScript pages as the skin of the app. Through a JavaScript API presented to these files, you can query the database and control the app.

Writing an app using HTML and JavaScript yields a lot of power. However, it can lead to a complicated design cycle.

The HTML and JavaScript files you write rely on the JavaScript API implemented within the ODK Tables APK to retrieve database values for your application. This JavaScript API, since it is implemented in the APK, makes it difficult to debug your custom views off the phone. At present, the only way to test your HTML pages is on the device. Fortunately, on Android 4.4 and higher, **Chrome** can access the browser Console and set breakpoints on the device, providing a clumsy but viable debug environment.

Understanding the Web File

There are several pieces of boilerplate you have to include in your own code in order to debug the files in **Chrome**.

In the default Application Designer, open `app/config/tables/Tea_houses/html/Tea_houses_list.html`. Alternatively, if you are doing the running example, open `app/config/tables/census/html/census_list.html`, which should have been automatically created for you. Notice the following four lines in `<head>`:

```
<script type="text/javascript" src="../../../assets/libs/jquery-3.2.1.js"></
↪script>
<script type="text/javascript" src="../../../../system/js/odkCommon.js"></script>
<script type="text/javascript" src="../../../../system/js/odkData.js"></script>
<script type="text/javascript" src="../../../../system/tables/js/odkTables.js"></
↪script>
```

In the first line you are making the **jQuery** object available to your code. **jQuery** is a powerful, commonly used set of functions for accessing and performing actions within a webpage. In the next three lines you are adding the *odkCommon*, *odkTables*, and *odkData* objects if they are not already provided by the browser environment. When running on the device, the ODK Tables APK will provide these, and the contents of these files will be ignored. When running in Application Designer on your computer, these files provide the approximate functionality of the APK, allowing you to create and debug your scripts. However, at the moment, these implementations make use of RequireJS, which the ODK Tables HTML files do not use (RequireJS is extensively used by ODK Survey). This causes these to break in Application Designer **Previews**.

More detail is provided in *ODK Tables Web Pages*.

Creating Web Files

To write your own file, first decide on the *tableId* for your table and instantiate a directory using the **grunt** command:

```
$ grunt addtable:tableId
```

If you completed the example in *ODK Survey: Designing a Form* you have already done this for the *census* table.

This **grunt** task creates the needed directory structures and also constructs the HTML and JavaScript files with the necessary features for working within the **Chrome** development environment.

Note: These files need content from your data table to display. It is recommended that you first design a Survey form (for example, using *this guide*) which you can use to populate

data. You can also prepopulate data into the database with a `tables.init` file. Further instructions are available in the *Configuring an App at Startup* guide.

Creating a List View

Continuing the ongoing example, open or create the file `app/tables/census/html/census_list.html`. This will display a list of records collected with the form.

Ensure the file looks like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<!--List View-->
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <link href="../../../assets/css/list.css" type="text/css" rel="stylesheet
↪" />
        <script type="text/javascript" src="../../../assets/commonDefinitions.js
↪"></script>
        <script type="text/javascript" src="../tableSpecificDefinitions.js"></
↪script>
        <script type="text/javascript" src="../../../assets/libs/jquery-3.2.1.js
↪"></script>
        <script type="text/javascript" src="../../../../system/js/odkCommon.js">
↪</script>
        <script type="text/javascript" src="../../../../system/js/odkData.js"></
↪script>
        <script type="text/javascript" src="../../../../system/tables/js/
↪odkTables.js"></script>
    </head>
    <body>
        <script type="text/javascript" src="../js/census_list.js"></script>
        <div id="wrapper">
            <div id="list"></div>
        </div>
        <script>
            $(function() { resumeFn(0); });
        </script>
    </body>
</html>
```

This HTML file should be minimal. It links all the source files and provides `<div>` to put the list in. Most of the work happens in the JavaScript file. Open or create `app/tables/census/js/census_list.js`. Ensure its contents look like this:

```
/* global $, odkTables, odkData, odkCommon */
'use strict';

// The first function called on load
var resumeFn = function() {

    // Retrieves the query data from the database
    // Sets displayGroup as the success callback
    // and cbFailure as the fail callback
          odkData.getViewData(displayGroup, cbFailure);
}

// Display the list of census results
var displayGroup = function(censusResultSet) {

    // Set the function to call when a list item is clicked
    $('#list').click(function(e) {

        // Retrieve the row ID from the item_space attribute
                    var jqueryObject = $(e.target);
                    var containingDiv = jqueryObject.closest('.item_space');
                    var rowId = containingDiv.attr('rowId');

        // Retrieve the tableID from the query results
                    var tableId = censusResultSet.getTableId();

                    if (rowId !== null && rowId !== undefined) {

            // Opens the detail view from the file specified in
            // the properties worksheet
                                    odkTables.openDetailView(null, tableId,␣
↪rowId, null);
                            }
                });

    // Iterate through the query results, rendering list items
    for (var i = 0; i < censusResultSet.getCount(); i++) {

        // Creates the item space and stores the row ID in it
        var item = $('<li>');
        item.attr('id', censusResultSet.getRowId(i));
        item.attr('rowId', censusResultSet.getRowId(i));
        item.attr('class', 'item_space');

        // Display the census name
        var name = censusResultSet.getData(i, 'name');
        if (name === null || name === undefined) {
```

```
            name = 'unknown name';
        }
        item.text(name);

        // Creates arrow icon
        var chevron = $('<img>');
        chevron.attr('src', odkCommon.getFileAsUrl('config/assets/img/little_
↪arrow.png'));
        chevron.attr('class', 'chevron');
        item.append(chevron);

        // Add the item to the list
        $('#list').append(item);

        // Don't append the last one to avoid the fencepost problem
        var borderDiv = $('<div>');
        borderDiv.addClass('divider');
        $('#list').append(borderDiv);
      }
      if (i < censusResultSet.getCount()) {
          setTimeout(resumeFn, 0, i);
      }
};

var cbFailure = function(error) {
    console.log('census getViewData CB error : ' + error);
};
```

The HTML and JavaScript files also depend on a few more files. For convenience, the example reuses CSS and image files from the *ODK Tables: Sample Application*. Open up a default Application Designer and copy the following files to this application's directory (using the same directory paths):

- `config/assets/css/list.css`
- `config/assets/img/little_arrow.png`
- `config/assets/libs/jquery-3.2.1.js`

Creating a Detail View

A *Detail View* will display the details of a record. It is commonly used alongside *List View* to provide options to browse through a data set and learn more about a particular record.

Open or create `app/tables/census/html/census_detail.js` Ensure the file looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.
↪org/TR/html4/loose.dtd">
<html>
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <link href="../../../assets/css/detail.css" type="text/css" rel=
↪"stylesheet" />
        <script type="text/javascript" src="../../../assets/commonDefinitions.js
↪"></script>
        <script type="text/javascript" src="../tableSpecificDefinitions.js"></
↪script>
        <script type="text/javascript" src="../../../assets/libs/jquery-3.2.1.js
↪"></script>
        <script type="text/javascript" src="../../../../system/js/odkCommon.js">
↪</script>
        <script type="text/javascript" src="../../../../system/js/odkData.js"></
↪script>
        <script type="text/javascript" src="../../../../system/tables/js/
↪odkTables.js"></script>
    </head>
    <body>
        <script type="text/javascript" src="../js/census_detail.js"></script>

        <h1><span id="TITLE" class="main-text"></span></h1>

        <fieldset>
          Is over 18: <input id="FIELD_1" type="checkbox" name="isAdult" />
        </fieldset>

        <script>
            $(display);  // calls the detail display function when ready
        </script>
    </body>
</html>
```

This HTML file should define the user interface elements that will be populated by database calls in the JavaScript. Open or create `app/tables/census/js/census_detail.js`. Ensure its contents look like this:

```
/* global $, odkTables, odkData */
'use strict';

var censusResultSet = {};
var typeData = {};

// Called when the page loads
var display = function() {
```

```
  // Runs the query that launched this view
  odkData.getViewData(cbSuccess, cbFailure);
};

// Called when the query returns successfully
function cbSuccess(result) {

  censusResultSet = result;
  // and update the document with the values for this record
  updateContent();
}

function cbFailure(error) {

  // a real application would perhaps clear the document fiels if there were an␣
↪error
  console.log('census_detail getViewData CB error : ' + error);
}

/**
 * Assumes censusResultSet has valid content.
 *
 * Updates the document content with the information from the censusResultSet
 */
function updateContent() {

  nullCaseHelper('name', '#TITLE');

  if(censusResultSet.get('isAdult') === 'y') {
    $('#FIELD_1').attr('checked', true);
  }
  $('#FIELD_1').attr('disabled', true);

}

/**
 * Assumes censusResultSet has valid content
 *
 * Updates document field with the value for the elementKey
 */
function nullCaseHelper(elementKey, documentSelector) {
  var temp = censusResultSet.get(elementKey);
  if (temp !== null && temp !== undefined) {
    $(documentSelector).text(temp);
  }
}
```

As with the *List View*, this view requires a separate CSS file. Copy the following file from a default Application Designer, maintaining the directory path in this application's directory:

- `config/assets/css/detail.css`

Defining Default View Files

The `.xlsx` form should be updated to indicate the default view type, and where to find the HTML files for *Detail View* and *List View*. Open `app/config/tables/census/forms/census/census.xlsx` and add a new worksheet titled *properties*. Add the following headers: partition, aspect, key, type, and value.

Add the following rows to set your *List View* and *Detail View* default files:

Table 4.54: *properties* worksheet

partition	aspect	key	type	value
Table	default	defaultViewType	string	LIST
Table	default	detailViewFileName	string	config/tables/census/html/census_detail.html
Table	default	listViewFileName	string	config/tables/census/html/census_list.html

See *Properties* for more details about specifying custom HTML files.

Run `census.xlsx` through the XLSX Converter again (*Generating formDef.json*) to update the configuration.

After that, you can deploy your app to your device. Open Survey and fill in a few census records. Then, open Tables and select the *Census* table. This should automatically launch the *List View* defined above. Tapping an item in the *List View* should launch the detail view.

Debugging Tables Web Files

You can use the **`Chrome`** browser on your computer to inspect for devices and connect to this custom screen on your Android device, and debug from there. Some useful guides include:

- Get Started with Debugging JavaScript in Chrome DevTools
- Get Started with Remote Debugging Android Devices

Warning: The edit-debug cycle is awkward because you must make the HTML or JavaScript change on your computer then push the change to your device, and reload the page (for example, by rotating the screen). When you do rotate the screen, however, it is rendered in a new web page, necessitating connecting to that new page to resume debugging (the prior page sits idle and will eventually be destroyed. If you don't see any activity, it is likely because you are pointing at the wrong web page. Return to inspect devices, and select the newest page).

As with ODK Survey, you can use the JavaScript Console to look for and fix errors in your HTML/JavaScript. If you are having trouble please check on the ODK Forum. Keep in mind that the debug objects only emit a subset of the data in your ODK Tables database.

4.11.5 Pushing and Pulling Files

Note: You must have USB debugging enabled on your device in order to perform this step. See these instructions for help.

There are several times during app development where you will need to push and pull files to and from the phone. You will have to open one of the ODK tools on the device before these commands succeed.

- The **`push`** command is used to push the entire app directory to the mobile device.
- The **`pull`** command is used to pull the database or exported CSVs from the device to the desktop computer.

Tip: Exported CSVs can be used to set up `tables.init` to load test data.

`Grunt` tasks have been written in `Gruntfile.js` that perform these operations for you.

These commands can be run anywhere within the `Application Designer` directory.

- **`grunt adbpush`**: Pushes everything under the app directory to the device.
- **`grunt adbpull-db`**: Pulls the database from the device to the PC.
- **`grunt adbpull-csv`**: Pull the exported CSVs from the device to the PC.

The pull commands will place the pulled content in the `app/output/` directory.

The database is a **`SQLite`** database and can be viewed using **`SQLite Browser`**. This tool can also be used to view the content of the database used by **`Chrome`** on your computer (the location of that file is OS dependent).

If you pull the CSV files, they will be under the `output/csv/` directory. You can then copy them to the `config/assets/csv/` directory and set up the `tables.init` file to read them in order to provision test data for your development effort. If you need any of this data in production, you will want to sync to a server then export the CSV files and copy them to the `config/assets/csv/` directory so that they have all of their metadata field values populated.

Tip: Running **`grunt adbpull`** will perform all the pull tasks.

Tip: There are a number of additional grunt tasks available. Assuming you have installed grunt and node, you can view the available tasks by running **`grunt --help`** anywhere in the repo.

4.11.6 Deploying an Application

This step requires that you first set up a *ODK Cloud Endpoints*.

1. Push your application to a clean device (guide: *Pushing and Pulling Files*).
2. Authenticate as a user in the table administrator group (guide: *Authenticating Users*).
3. Reset the App Server (guide: *Reseting the App Server*).

The application is now deployed to your server. Other devices can synchronize with that server to download the application and start collected data.

Updating an Application

To update any app level or table level files, or to modify the database schema (like adding a new field to your form that adds a database column), you will need to reset the app server. Make the changes on your PC as normal, push them to the device, and reset the app server.

Warning: Resetting the app server will start a new data set. If you want to keep the old data, you should download it to a separate database.

4.12 ODK Tables Web Pages

Tables does not impose any structure to the HTML files a user may write. While the *odkDataIf* and *odkCommonIf* interfaces will always be injected into the WebKit, it is up to

the user whether or not they make use of them (which generally means loading the *odkData* and *odkCommon* wrapper objects). A typical HTML file might be as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.
↪org/TR/html4/loose.dtd">
<html>
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <link href="../../../assets/css/your.css" type="text/css" rel="stylesheet
↪" />
        <script type="text/javascript" src="../../../assets/commonDefinitions.js
↪"></script>
        <script type="text/javascript" src="../tableSpecificDefinitions.js"></
↪script>
        <script type="text/javascript" src="../../../../system/js/odkCommon.js">
↪</script>
        <script type="text/javascript" src="../../../../system/js/odkData.js"></
↪script>
        <script type="text/javascript" src="../../../../system/tables/js/
↪odkTables.js"></script>
        <script type="text/javascript" src="../../../assets/libs/jquery-3.2.1.js
↪"></script>
        <script type="text/javascript" src="../js/example.js"></script>
    </head>
    <body>
        <br>
        <a href="#" onclick="odkTables.openTableToListView(null,
          'visit',
          'plot_id = ?',
          [exampleResultSet.getRowId(0)],
          'config/tables/example/html/example_list.html');">Examples</a>
        <script>
            $(display);  // calls the display() function in example.js when␣
↪document ready
        </script>
    </body>
</html>
```

The `DOCTYPE` header defines the file compliance level (in this case, HTML 4.01 Transitional).

The `<head>` section sets the viewport and loads your CSS file. It will then typically load the 5 standard JavaScript files needed to support translations and the injected interfaces (`commonDefinitions.js`, `tableSpecificDefinitions.js`, `odkCommon.js`, `odkData.js`, and `odkTables.js`). This example then loads the 3rd-party JavaScript libraries that the user needs and which the user has provided in the `/config/assets/libs` directory and, finally, loads the JavaScript file they crafted that is specific to this web page (`example.js`).

Once all the scripts and resources on the page have loaded, the script tag at the bottom of the `<body>` section will invoke the display() function which was presumably specified in the user's `example.js` file.

If the web page can launch other web pages or external applications, and if it cares about the status of those requests or needs to process the extras in the result intents returned from those requests (e.g., to interpret a barcode), then the user's `display()` function, after it has initialized the page, should register a listener for action outcomes and call that listener once, directly, to process any outcome received prior to that registration (it will commonly be the case that these will have been received prior to the registration of this listener).

See the comments at the top of the `odkCommon.js` file for details.

4.13 Injected JavaScript Interfaces

- *odkCommon.js*
- *odkData.js*
- *odkTables.js*
- *odkSurvey.js*
- *Other system/survey/js files*

The Java framework on the Android device injects two Java interfaces (*odkCommonIf* and *odkDataIf*) into both Tables and Survey WebKits. Additionally, it injects one additional Java interface into each: *odkTablesIf* into Tables WebKits and *odkSurveyStateManagement* into Survey WebKits.

Within the Javascript, it is expected that all interactions with these interfaces will be done through wrapper objects. Specifically, for *odkCommonIf* and *odkDataIf*, Javascript programmers would invoke methods on the *odkCommon* and *odkData* objects defined in

- `system/js/odkCommon.js`
- `system/js/odkData.js`

The Tables-specific interface is interacted with via the *odkTables* object defined in:

- `system/tables/js/odkTables.js`

This wrapper object mostly invokes *odkCommon* to perform its actions, but does call the *odkTablesIf* injected interface's one method to load the list view portion of the split-screen detail-with-list-view layout.

The Survey interface is invoked within the Javascript that implements the survey presentation and navigation logic and should not be directly called by form designers.

4.13.1 odkCommon.js

This creates a *window.odkCommon* object that wraps calls to the injected *odkCommonIf* Java interface. When loaded inside the App Designer, it also creates a mock implementation of the injected interface.

This class provides support for:

1. obtaining information about the runtime environment (e.g., Android OS version, etc.)
2. obtaining information about the currently-selected locale.
3. obtain the active user.
4. obtain system properties (e.g., deviceId).
5. emitting log messages to an application log.
6. translations of text, media files and urls.
7. conversion functions for retrieving and storing timestamps and intervals.
8. storing and retrieving session variables (transient values that persist for the lifetime of this WebKit).
9. converting relative paths of configuration files and of row-level attachments into URLs suitable for use in HTML documents (e.g., image src attributes).
10. constructing form references used to launch ODK Survey.
11. invoking arbitrary intents (external programs) on Android devices.
12. obtaining the results from an intent that was previously invoked.
13. exiting the current WebKit and specifying a return intent status value and extras bundle.

The explicit session variable interfaces (`odkCommon.getSessionVariable(elementPath)` and `odkCommon.setSessionVariable(elementPath, value)`) provide a mechanism to preserve the state of a webpage within the Java Activity stack so that no matter how nested the call stack to external applications becomes, it can be unwound and the state of the webpage recovered. Similarly, the invoking of arbitrary intents and the retrieving of their result intent status and extras bundle (excluding byte arrays) provides direct access to Android's native application inter-operation capabilities from within the WebKit. This interface is used within Survey for media captures; the internal methods that accomplish this are in `system/survey/js/odkSurvey.js`. Within Tables, this capability is used to navigate between HTML pages for general content, list views, and detail views (largely via the higher-level methods of the *odkTables* wrapper object). As a webpage designer, there is nothing preventing you from performing media captures from Tables web pages, or from defining custom prompts within Survey that launch into Tables list views, etc. by leveraging one or the other of the *odkSurvey* or *odkTables* objects.

4.13.2 odkData.js

This creates a `window.odkData` object that wraps calls to the injected *odkDataIf* Java interface. When loaded inside the App Designer, a mock implementation of the injected interface is loaded that uses W3C SQL to emulate the injected interface's capabilities.

This class provides support for asynchronous interactions with a SQL database (internally, this is implemented via a SQLite database).

The interaction to get the active user's roles would be:

```
// declare a success function
var successFn = function( resultObj ) {
  // do success handling
  var roles = resultObj.getRoles();
  // this will be a list of the roles and groups the user
  // belongs to.
};
// declare the failure function
var failureFn = function( errorMsg) {
  // errorMsg is a text string. Typically the getMessage()
  // of the Java Exception that occurred during processing.
  // do failure handling
};
//
// make the asynchronous request
odkData.getRoles(successFn, failureFn);
```

If the request failed, the *errorMsg* is the message returned from within the Java layer. As noted, this is typically the `getMessage()` of an exception.

Otherwise, the `resultObj` returned contains information about the outcome. This object is a wrapper object with accessor methods defined here.

Note:

1. the color information is only present within Tables. It is not computed and returned within Survey.
2. the display names will need to be localized before use. See the APIs provided by *odkCommon*.

4.13.3 odkTables.js

As noted, this is here:

```
system/tables/js/odkTables.js
```

It provides methods to open Tables generic web pages and list and detail views. These are generally just wrappers for calls to *odkCommon* to invoke the intents for those views.

4.13.4 odkSurvey.js

As noted, this is here:

```
system/survey/js/odkSurvey.js
```

It provides methods to capture media files and. like *odkTables* these are generally just wrappers for calls to *odkCommon* to invoke the intents for those actions.

4.13.5 Other system/survey/js files

These files are generally not used by web page developers. They implement the survey form execution logic and their functions will be broadly covered later in this document.

4.14 Reference Applications

The reference applications are presented here as real world examples of Data Management Applications built on the ODK 2 platform. The following documentation provides a guided tour of these application's workflow and modules, as well as an overview of the internal structure of its source files. The intention is to provide a useful example for organization's to use when creating their own Data Management Applications.

4.14.1 Geographic Health Survey App

EpiSample is a geographic health survey application originally developed by Belendia Serda at MACEPA. It is a prototype application for health surveys regarding malaria prevention behavior, and has been used Ethiopia and Zambia. This article, written about an earlier version of EpiSample, provides some context for its development: The scrum master, the coder, and the phone.

Key Features

The EpiSample application provides a good example of the following ODK 2 platform features:

- **Data Synchronization and Reuse**: Collected data is reused to generate tasks across devices. Shared configuration is set by a supervisor and synchronized across devices.
- **Custom Web Views**: Location data is displayed and updated in real time. Custom data visualizations are used on data entry screens to help guide collection.
- **Complex Workflows**: Custom logic is implemented in JavaScript to generate tasks using collected data and to dynamically launch Survey forms.
- **Mapping and Navigation**: A map of of homes is generated using collected geo-data. *Navigate View* is integrated into the workflow to guide data collectors to homes and launch follow up surveys.
- **Adaptability**: The application is designed to be flexible to differing usage scenarios. Different Survey forms can be loaded into the same workflow to adapt to different data collection needs. Data quantity and location accuracy requirements can be configured in the app and updated as needed.

Installing EpiSample

Source code for this Data Management Application can be found in the malaria-demo branch of the App Designer repository. As with all of the ODK 2 reference applications (and the ODK 2 platform itself), the code is free and open source. Feel free to reuse, modify, or extend this application and its component parts to suit your organization's needs.

Note: The EpiSample application (and all other Data Management Applications provided in these docs) come with a full copy of the Application Designer they were developed in.

After you have have downloaded the application, you can set it up according to the *Application Designer setup* instructions. Similarly, you can push the application to your device using the *Pushing and Pulling Files* instructions.

Guided Tour

A walk-through of the features and the application and an overview of how they are implemented is provided in the guide below.

EpiSample Guided Tour

This document guides you through each of the modules of the *Geographic Health Survey App*, named EpiSample. They are presented in a logical order, so this guide can be read from top to bottom to mimic the flow of the application's use in the field. You can also skip to the module that most interests you. Each section provides both a brief description of the purpose and function of the module, as well as an overview of how that functionality is implemented.

Note: All file paths in this document are inside of the Application Designer directory. Additionally, all user defined files in a Data Management Application are inside the `app/config/` directory. For convenience this document omits these portions of the file paths.

For example, let us assume I have stored the Application Designer directory on my computer in `/home/bobsmith/workspace/app-designer`. If this guide were to reference a file as :*assets/index.html* that indicates the file located on my computer at `/home/bobsmith/workspace/app-designer/app/config/assets/index.html`.

Configuration

Function

The Configuration module is the first screen the user sees after launching the application. It allows the user to customize the behavior of the application via the Settings Screen. It also requires the user to specify the location that the surveys will take place through a series of dropdown menus. Below you can see that the *Select* button is not revealed until the full location is specified.

The location chosen on this screen is saved and all future modules will make use of that information.

The Settings screen is a submodule of Configuration. It allows you to customize the behavior of each module. This includes the acceptable GPS accuracy range, the number of data points to collect, and the Survey form to use for the *Household Data Collection*. These settings are shared across all devices that share *ODK Cloud Endpoints*.

The settings can be protected behind a user defined administrative password. If a password is set, the settings cannot be viewed or modified until it is entered, as shown below.

Implementation

This is the home screen first shown when the application is launched, so its HTML file must be: `assets/index.html`.

In the `<head>` of `index.html` notice that, among the standard ODK 2 JavaScript imports there are also `libs/sha256.js` and `js/epsConfigLib.js`. The `sha256.js` file is used for protecting the admin password. The `epsConfigLib.js` file provides an interface for reading and writing the configuration to the *Config* table of the database. Since the configuration is stored in the ODK database, any time the application is synchronized, these settings will be synchronized with the server. In this way an administrator can remotely control the settings on all the field workers phones. This custom library is included across all the files in this application.

The library `libs/bootstrap-3.3.7...` and the file `js/eps_select_place_name.js` are also linked at the bottom of the `<body>`. **`Bootstrap`** is a third party library used for formatting and look-and-feel. `eps_select_place_name.js` implements the dynamic portions of the user interface of the home screen including: the series of dropdowns that specify the place name, the *Settings* button, and the login screen for password protected settings.

The place names dropdowns are populated dynamically by reading from the *Place Names* table. These are read via `odkData.query` and `odkData.arbitraryQeury` calls. When the place name is selected, it is stored for later use by the rest of the modules with `odkCommon.setSessionVariable`. The *Select* button launches `eps_main_menu.html`, which is covered in the next module.

When the *Settings* or *Login* buttons are pressed, they will launch `assets/eps_config.html`. This file implements the Settings screen and all its inputs. It links to `assets/js/eps_config.js` to handle its logic. This file handles reading the stored configuration from the database (via `epsConfigLib.js`), populating that into the form fields, and saving the new configuration back to the database after the *Save* button is pressed.

To populate the *Choose Form* dropdown, the `populateChooseFormControl()` function in `eps_config.js` reads the list of available Survey forms via the `odkData.getAllTableIds` function.

Files

- `assets/index.html`
- `assets/js/epsConfigLib.js`
- `assets/js/eps_select_place_name.js`
- `assets/eps_config.html`
- `assets/js/eps_config.js`

Forms

None

Database Tables

- *Config*
- *Place name*

Main Menu

Function

The Main Menu module is the hub to launch the other modules. The currently selected place name is displayed along the top for convenience.

The buttons that launch other modules can be dynamically added and removed via the Settings screen in the *Configuration* module.

Implementation

The file `assets/eps_main_menu.html` is launched by the *Select* button in the *Configuration* module. It provides a basic skeleton for this UI, but most of this screen's elements are dynamic. They are defined in `assets/js/eps_main_menu.js`.

The file `eps_main_menu.js` creates the buttons to launch the various modules of this application. It selectively hides these buttons if the configuration dictates this (see the `init()` function). It also handles setting up the state for launching each of these modules.

To launch the *Village Geographic Survey* (with the *Collect* button), no setup is required. A direct call to `odkTables.launchHTML(...)` will suffice. This is true of the *Task List Generation* (with the *Select* button) as well.

To launch the *Geographic Navigation*, a query must be passed that selects the points to which this particular user should navigate. The call happens with this function: `odkTables.openTableToNavigateView(...)` which queries the *Census* table for these points (see the *Village Geographic Survey* for how this table is populated). The preceding code dynamically generates the SQL `WHERE` clause and `SELECT` arguments. This view is also opened with a `dispatchStruct`, which triggers the Action-Callback workflow.

The launch of the *Geographic Navigation* is automatic, unlike the manual button pressing of the other modules. This occurs via the Action-Callback workflow triggered when the Navigation module is launched. See the `actionCBFn()` function and the `odkCommon.registerListener(...)`, `odkCommon.viewFirstQueuedAction(...)`, and `odkCommon.removFirstQueuedAction(...)` functions. When the Navigation module completes its action, the result is handled by this function. If the result indicates to do so, the Household Survey module will be launched via `odkTables.editRowWithSurvey(...)` (using the configured Form ID).

The Household Survey is also launched with the Action-Callback workflow. When it returns, the results are used to update the *Census* table to match its corresponding form's completion status.

The currently selected location is displayed at the top by reading the value from `localStorage`.

Files

- `assets/eps_main_menu.html`
- `assets/js/eps_main_menu.js`

Forms

None

Database Tables

- *Config*
- *Census*

Village Geographic Survey

Function

The Village Geographic Survey module is used to gather census data about each household. It records basic information: a house number, a head-of-household name, and the GPS coordinates of that household. The house number field is automatically increased with each saved record.

Recorded households are listed in the bottom portion of the screen. This list includes the name and house number, an *Edit* button that allows you to update the record, and an icon indicating whether the record is *Valid* or not. The validity of a record is determined by the accuracy of its GPS coordinates. The thresholds are set in the *Configuration* module.

The quality of the GPS signal is depicted by the color of the spinner and the specific rating listed.

The above screen on the left depicts a GPS signal that is not accurate enough, and is displayed in yellow. The screen on the right shows that the icon next to *Beth* is an *I* for

Invalid, rather than the *V* for *Valid* next to Alex.

A running total of records is indicated in between the collection portion of the screen and the record list. It is separated into the *Valid*, *Invalid*, and *Excluded* categories. The difference between *Invalid* and *Excluded* is that an *Excluded* record is manually excluded via the *Exclude* checkbox.

In *Invalid* record can only be made valid by recapturing the GPS coordinates when the accuracy is sufficient. The image below shows the record editing screen.

To replace the GPS, the *Replace GPS* check-mark should be checked, and the *Update* button should be pressed.

After all of the household data has been recorded, the user should synchronize their results to the server (*Syncing instructions*).

Implementation

The basic structure of the user interface is defined in `assets/eps_collect.html`, including all the input fields and the container for the list. Dynamic adjustments to this user interface, as well as calls to the database and device hardware, are made in `assets/js/eps_collect.js`. This file's makes heavy use of the third party **`Backbone`** JavaScript library. Also notice that the third party library **`Underscore`** is included, along with template HTML, for dynamically adding list items.

The file `eps_collect.js` handles:

1. Keeping track of the GPS coordinates and accuracy in real time. It also updates the user interface as necessary when these change. The thresholds for GPS accuracy are read from the settings with the `epsConfigLib.js` file.
2. Reading, Creating, and Updating records in the *Census* table. This data is also validated before being recorded. The records are read through a number of calls to `odkData.query(...)` and `odkData.ArbitraryQuery(...)`. They are recorded with calls to `odkData.addRow(...)` and the are updated with calls to `odkData.updateRow(...)`.
3. Dynamically creating the visualization of the list of records from the *Census* and updating it as that list changes. This list is also paginated. The running totals of *Valid*, *Invalid*, and *Excluded* records are populated with `odkData.arbitraryQuery` calls.

The file `assets/js/util.js` is included to generate UUIDs (unique ids and primary keys in the database) for each new record as it is created.

Files

- `assets/eps_collect.html`
- `assets/js/eps_collect.js`
- `assets/js/util.js`

Forms

None

Database Tables

- *Config*
- *Census*

Task List Generation

Function

The Task List Generation module is used to create a randomized task list for each data collector to perform.

Note: It is important that the Task List Generation module has access to all of the census data. Every data collector should synchronize their device to the server (*Syncing instructions*) so that all the records are available. After that, a synchronization can be performed to receive the full record set.

The *Main*, *Additional*, and *Alternate* points parameters are set in the *Configuration* module.

Tip: To allow these fields to be set in this module, set the parameters to zeros in the Config module.

If there are sufficient records available to meet the parameters, the user can press the *Select* button to generate the task list. A progress bar will appear while this process takes place,

followed by a completion notification. This process can take some time if the data set is large. After the task list is generated it can be synchronized to the server, followed by each data collector synchronizing to receive the list.

This list of tasks is used to determine where to perform follow up surveys.

Implementation

The file `assets/eps_select.html` implements the look of this screen. This screen is not nearly as dynamic as the others, and as such most of the user interface is hard coded here. This includes the loading screen that appears while the list is being generated.

The file `assets/js/eps_select.js` reads the *Main*, *Alternate*, and *Additional* point configuration and populates the user interface with it. If these are configured to zeros, it will read in the user defined values for these fields.

When the *Select* button is pressed, the configuration is used to determine the points to select for the task list. The third party **`Async`** library is used to handle these long running calculations in the background without locking up the user interface. While they run, the *Please Wait* loading screen is shown. The records are read from the *Census* table with an `odkData.arbitraryQuery` call, and then randomized. When this process is complete, the affected records in the *Census* table are updated with `odkData.updateRow` calls.

Files

- `assets/eps_select.html`
- `assets/js/eps_select.js`

Forms

None

Database Tables

- *Config*
- *Census*

Geographic Navigation

Function

The Navigation Module helps guide a data collector to the households specified in the task list. The compass, distance, and other navigational indicators will update in real time.

The map will show the points on the task list. The households displayed can be configured on the *Main Menu* module to show only *Main* points, or show *Main* and *Additional* and so on.

When the data collector arrives at the household, they can tap the *Arrive* button to launch the *Household Data Collection* module. Or they can press *Cancel* at any time to return to the Main Menu.

Implementation

This view is provided by the ODK 2 platform and is not customizable. The view is launched by a call to `odkTables.openTableToNavigateView(...)` with query parameters to select the map markers. The query that selects the map markers is discussed in the *Implementation*

section. The handling of the results of the *Arrive* and *Cancel* button presses are also discussed in that section.

Files

None

Forms

None

Database Tables

- *Census*

Household Data Collection

Function

The Household Data Collection module is an ODK Survey form that is configured in the *Configuration* module. This is intended to be provided by the Deployment Architect for their particular use case, which makes this application flexible to different scenarios. For example, this could be used for follow up after a Malaria outbreak or it could be adapted to handle another disease by swapping this form.

The data collected in this form is available in the same database as the rest of the application and can be used by it.

Implementation

The Household Data Collection form is user configured and not provided in this reference application. The provided form could be a simple survey or could include complex skip logic, data quality checks, and customizations to the look-and-feel. Instructions for creating these forms are available in *ODK Survey: Designing a Form* guide as well as the *ODK XLSX Converter* guide.

Prepopulated forms could also be included with CSV files. See the files in the `assets/csv` directory and the `assets/tables.init` file for examples.

Files

- The files associated with the selected form.

Forms

- The form configured in the *Configuration* module.

Database Tables

- The table corresponding to the selected form.

4.14.2 Longitudinal Clinic Study App

The Hope Study is a longitudinal clinical trial originally developed at the University of Washington as a collaboration between the Computer Science and Global Health departments. It was a randomized control trial studying pregnant, HIV discordant couples and their health outcomes, and was used for eight months by health workers in western Kenya. More info about the study can be found in this article. The study was conducted using ODK Collect,

and then ported to the ODK 2 tools to demonstrate the features that could be added on this platform.

Key Features

The Hope Study application provides a good example of the following ODK 2 platform features:

- **Data Synchronization and Reuse**: The study takes place over a number of months, revisiting the same clients and reusing previously collected data. Launching the synchronization interface is built into the application's workflow. Known data is prepopulated into form prompts.
- **Custom Web Views**: Navigation to the current client and appropriate form is made simple and easy. Custom data visualizations provide statistics on the full data set.
- **Custom Form Linkage**: Multiple Survey forms update the same records in a single database table.
- **Complex Form Navigation**: Forms will jump between screens based on client eligibility and response validation.

Installing Hope Study

Source code for this Data Management Application can be found in the master branch of the App Designer repository. It is one of the five demo applications for ODK Tables. As with all of the ODK 2 reference applications (and the ODK 2 platform itself), the code is free and open source. Feel free to reuse, modify, or extend this application and its component parts to suit your organization's needs.

Note: The Hope Study application (and all other Data Management Applications provided in these docs) come with a full copy of the Application Designer they were developed in.

After you have have downloaded the application, you can set it up according to the *Application Designer setup* instructions. Similarly, you can push the application to your device using the *Pushing and Pulling Files* instructions.

Guided Tour

A walk-through of the features and the application and an overview of how they are implemented is provided in the guide below.

Hope Study Guided Tour

This document guides you through each of the modules of the *Longitudinal Clinic Study App*, named Hope Study. They are presented in a logical order, so this guide can be read from top to bottom to mimic the flow of the application's use in the field. You can also skip to the module that most interests you. Each section provides both a brief description of the purpose and function of the module, as well as an overview of how that functionality is implemented.

Note: All file paths in this document are inside of the Application Designer directory. Additionally, all user defined files in a Data Management Application are inside the `app/config/` directory. For convenience this document omits these portions of the file paths.

For example, let us assume I have stored the Application Designer directory on my computer in `/home/bobsmith/workspace/app-designer`. If this guide were to reference a file as :*assets/index.html* that indicates the file located on my computer at `/home/bobsmith/workspace/app-designer/app/config/assets/index.html`.

Main Menu

Function

The Main Menu module is the first screen a user sees after launching the application, and provides a basic choice of the three main workflow options available:

- *Screen Female Client*: to launch the female client screening form: *Screen Client*.
- *Follow Up with Existing Client*: to find the record for an existing client and launch the appropriate follow up form (the *Existing Client List* module).
- *Send Data*: To launch the synchronization interface and sync the database with the server: *Send Data*.

Implementation

This is the first screen a user sees, which would usually imply that its html file is `assets/index.html`. However, this application is embedded within the larger Tables Sample Application, which claims the `index.html` file. That demo launches `assets/hope.html`, which defines this screen. If this application were extracted from this sample application and deployed on its own, this file would need to be renamed `index.html`.

This file creates its HTML `<body>` dynamically with embedded JavaScript. This JavaScript defines the three buttons:

- *Screen Female Client*: Calls `odkTables.addRowWithSurvey(...)` to launch the *screenClient* form. Notice that the database table being written to is *femaleClients* which is shared among other forms. See the *Follow Up Forms module implementation details*.
- *Follow Up with Existing Client*: Calls `odkTables.openTableToListView(...)` to launch the Existing Client module.
- *Send Data*: Calls `odkCommon.doAction(...)` to launch the *SyncActivity*. This is the same functionality as pressing the sync button in the upper right of the screen, but with two advantages.
  1. The call is embedded within the custom workflow of the application so the user can be instructed to use it at th appropriate time.
  2. The `doAction` function supports the Action-Callback workflow, which means further actions could be triggered after synchronization is completed.

Files

- `assets/hope.html`

Forms

None

Database Tables

None

Screen Client

Function

The Screen Client form is used to conduct an interview with a potential new client. It provides a script for the interviewer and ask questions relating to the potential client's eligibility along with some basic demographic information. If the client is eligible, it asks for consent and instructs the interviewer in the process of randomizing the client into Hope or the control group.

Implementation

The form is defined in `tables/femaleClients/forms/screenClient.xlsx`. This file is not directly consumed by the Android tools, but instead is input into the *ODK XLSX Converter* to create the files that the Android tools use (`formDef.json`, `definition.csv`, and so on). However we will only directly interact with the `.xlsx` file.

The *survey* worksheet contains the main flow of the form. Interviewer scripts are provided with note prompt types, and data is collected mostly using select_one, select_multiple, and integer prompts. The flow of the form is controlled by if, else:, and goto elements in the clause column, equations in the condition column, and branches in the branch_label column.

Data is validated with the required column and select_multiple prompts define their choices with links in the values_list column.

The *choices* worksheet defines all the options for each select_multiple prompt.

The *settings* worksheet defines the form_id and the table_id to store form instances. Notice that the table_id is *femaleClients*, which will also be the table_id for other forms in the *follow up forms*.

The *model* worksheet is used to specify the data model for the *femaleClients* table. See the *XLSX Converter Reference* for more details

Files

- `tables/femaleClients/forms/screenClient.xlsx`

Forms

- *Add Client Form* with form ID *screenClient*

Database Tables

- *femaleClients*

Send Data

Function

This module is the synchronization interface, see the *Services User Guide*. It is included here because it is embedded into the Hope Study workflow. After registering new clients or interviewing existing clients, this module is used to submit that data. Also, before work is started each day, this should be used to ensure the latest forms from all the clients are available on the device.

Implementation

The synchronization interface is implemented completely by the ODK 2 platform and it is not customizable. For details on how to launch it through your application files (rather than the default provided sync button), see the *Main Menu implementation details*.

Files

None

Forms

None

Database Tables

None

Existing Client List

Function

The Existing Client module is used to perform follow up interactions with existing clients, or to view statistics about the current client pool. It displays a list of all of the currently registered clients with their client ID, age, and randomization status. This list will grow to be quite long, so searching by client ID is supported. Typing in the desired client ID will leave only the matching client in the list below. If you were to search for client ID 44176, the interface would update as shown in the following image.

Tapping a client in the list will launch the *Client Details* module.

If a new client needs to be added, bypassing the screening process (for example, if a registered client is scheduled for an interview but they are not showing up in the system because a worker didn't synchronize recently enough) the *Add Client* button will launch the survey: *Add Client Brief*. This form contains a brief questionnaire (a subset of the full *Add Client* form from the *Screen Client* module) that writes to the same database table (*femaleClients*). After the form is saved it will appear on the Existing Clients list and a follow up survey can be performed.

Pressing the *Graph View* button will launch the *Graph View* module.

Implementation

The HTML file `tables/femaleClients/html/femaleClients_list.html` provides a a form to contain the search function and a division to contain the list of records. The interface is implemented in `tables/femaleClients/js/femaleClients_list.js`.

The JavaScript in `femaleClients_list.js` implements that standard workflow for a *List View*. The API call that launches the view provides a query to define the list of records that will compose the *List View*. This query is queued and ready, and the results are retrieved with the `odkData.getViewData(...)` function. The returned list of female clients are used by the `render()` function to build the client list's HTML elements and display them. Each

list item is coded to call the `handleClick(...)` function if they are pressed, which launches the *Detail View* of the selected record with `odkTables.openDetailView(...)`.

The `render()` function also creates the *Add Client* and *Graph View* buttons. These are hard coded to launch `odkTables.addRowWithSurvey(...)` and `odkTables.openTableToListView(...)` respectively. This particular graph view is a special case of a *List View*, and the query provided (selecting every client) provides the data the graph will use to render its visualizations. Each of these calls modifies or reads the *femaleClients* table.

The survey that is launched by the *Add Client* button is defined by `tables/femaleClients/forms/addClient.xlsx`. It is quite short and simple, and is a subset of the form described in the *Screen Client implementation details*.

The function `getResults()` implements the search functionality. It queries the database with `odkData.query(...)`, searching for the provided client ID. If that client is found, it opens up a new instance of this module with the same query, ensuring the *List View* will only show the desired client. Opening a new client, rather than updating the current list, ensures that when the user presses the back button they will return to the current instance of the module with the full list of clients instead of returning to the home screen.

Files

- `tables/femaleClients/html/femaleClients_list.html`
- `tables/femaleClients/js/femaleClients_list.js`
- `tables/femaleClients/forms/addClient.xlsx`

Forms

- *Add Client Brief* with form ID *addClient*

Database Tables

- *femaleClients*

Client Details

Function

The Client Details module is a hub to perform follow up and additional data collection for the selected client. The three main data points about the client are displayed at the top: client ID, age, and randomization status. The follow up forms are organized into *Client Forms* (forms pertaining to the female client) and *Partner Forms* (forms pertaining to the female clients partner). Tapping the desired section will expand it to display the collection options.

The *Home Locater* button launches the *Home Locater* module. The rest of the forms are discussed in the *Follow Up Forms* section.

Implementation

The HTML file `tables/femaleClients/html/femaleClients_detail.html` is minimal, like its sister *List View* HTML file, and provides only a basic skeleton of the user interface which will be filled in by `tables/femaleClients/js/femaleClients_detail.js`

The JavaScript file `femaleClients_detail.js` implements the basic *Detail View* workflow.

The call that launched the view provided a query as a parameter that selects which record will be displayed in the *Detail View*. This record is retrieved with the `odkData.getViewData(...)` call. The returned record is used to fill in the basic info at the top of the screen, and then the *Client Forms* and *Partner Forms* are created. These buttons link to the *Follow Up Forms* using `odkTables.editRowWithSurvey(...)` and `odkTables.addRowWithSurvey(...)` API calls. All the buttons in the *Client Forms* section use the table ID *femaleClients* and the client ID of the selected record, while all the buttons in the *Partner Forms* section use the table ID *maleClients* and the male client ID (read from the selected record).

Additionally, the *Home Locater* module is launched with the *Home Locater* button using an `openTableToListView(...)` call.

Files

- `tables/femaleClients/html/femaleClients_detail.html`
- `tables/femaleClients/js/femaleClients_detail.js`

Forms

None

Database Tables

- *femaleClients*

Home Locater

Function

The Home Locater module is used to navigate to the client via provided instructions and waypoints on a map. The main screen shows you this list of instructions, including the means of transportation to travel between each one (such as walking or Tuk-tuk). These instructions are populated by pressing the *Add Waypoint* button. This button launches a survey that records the transportation type, GPS coordinates, and enters it in its proper place in the list.

A button is also provided to launch a *Map View*.

This map view shows the same list of instructions on the top, but uses most of the screen real-estate to show the waypoint markers on the map. Tapping a map marker highlights the instruction. Tapping the instruction on the list opens a *Detail View* of the instruction.

Implementation

The Home Locater module functions almost as a miniature version of the rest of the application. The root *List View*, much like the *Existing Client List* module, receives its list from the caller query via the `odkData.getViewData(...)` call and uses that to render a list of clickable items that will launch the *Detail View*. The *Detail View* shows to basic data about the record, similarly retrieved with the `odkData.getViewData(...)` call.

The *Add Waypoint* button acts similarly to the *Add Client* button in the Existing Clients module, but launches the *Home Locater* form. This form contains a few basic text and select_one prompts, as well as a geopoint prompt to collect location data. The *properties* worksheet is what makes this form distinct from the other forms in this application. It specifies all the mappings to set up the *Map View*, such as the mapListViewFileName and the defaultViewType as a MAP.

The `tables/geopoints/html/geopoints_map_list.html` and `tables/geopoints/js/geopoints_map_list.js` files fine a basic list that should be recognizable in structure to the other *List View* files. However, it has added logic to handle the ordering of the list items basic on selected points on the map in the `render(...)` function.

Files

- `tables/geopoints/html/geopoints_list.html`
- `tables/geopoints/js/geopoints_list.js`
- `tables/geopoints/html/geopoints_detail.html`
- `tables/geopoints/js/geopoints_detail.js`
- `tables/geopoints/html/geopoints_map_list.html`
- `tables/geopoints/js/geopoints_map_list.js`
- `tables/geopoints/forms/geopoints.xlsx`

Forms

- *Home Locater* with form ID *geopoints*

Database Tables

- *geoopoints*

Follow Up Forms

Function

The follow up forms are the bulk of the longitudinal study. After clients are screened and entered into the study, six week and six month follow ups will take place. Furthermore, their partner may be screened to enter the study with them, and also receive a six month follow up.

Each of these are launched from the *Client Details* module. They are survey forms that provide the interviewer with a script and ask detailed medical questions. Some previously collected data will be prepopulated in the forms prompts.

Implementation

The two *Client Forms* both read and write from the *femaleClients* table (as can be seen on the *settings* worksheet of both forms). This is true for both *Partner Forms* and the *maleClients* table as well.

These surveys are similar in structure to the initial *Screen Client* form. There is basic navigation logic via if and other condition clauses. Simple data collection occurs with select_one,

select_multiple, integer, decimal, text, and date prompts. Interviewer scripts are provided with note prompts. The client ID is marked as necessary with the required: column, however, this field should always be prepopulated in follow up forms. This is because the form is modifying an existing record in the database, and the field already has a value. In general this could be changed, though in this workflow this would be rare. The *model* worksheet provides the linkage with the database table.

There are also files to define *List Views* and *Detail View* for the male partners, even though they are not reachable through the normal workflows. These can be launched by opening the table directly via the *Tables Manager* screen.

Files

- `tables/femaleClients/forms/client6Week.xlsx`
- `tables/femaleClients/forms/client6Month.xlsx`
- `tables/maleClients/forms/screenPartner.xlsx`
- `tables/maleClients/forms/partner6Month.xlsx`

Forms

- *Client 6 Week* with form ID *client6Week*
- *Client 6 Month* with form ID *client6Month*
- *Screen Partner* with form ID *screenPartner*
- *Partner 6 Month* with form ID *partner6Month*

Database Tables

- *femaleClients*
- *maleClients*

Graph View

Function

The Graph View module is a simple pair of pie chart displaying statistics about the current client pool. The first pie chart, titled *Intervention Arms*, displays the ratios of clients in each of the intervention arms: Hope, Control, or Ineligible. The second pie chart, titled *HIV Status*, shows the ratios of HIV+, HIV-, and untested respondents.

This view is currently displaying static data and does not reflect the true values in the database. This was done for simplicity in showing Tables features in the Sample Application: showing how a graph might look, despite not having real patient data initialized on the device. However, with a little a couple calls to the database and a little more JavaScript implementing the math, this could be updated dynamically.

Implementation

The implementation of this module is less interesting considering it does not show real data. However, it is still a useful display of how complex data visualizations can be rendered on the device, without outside processing or internet access. The file `tables/femaleClients/`

`html/graph_view.html` makes use of the third party **D3** JavaScript library to draw the pie charts.

To ratios are fed into the graph rendering in the `display(...)` function. If these were replaces with variables, and those variables were populated by summing up results of calls to the database with `odkData.query(...)` and `odkData.arbitraryQuery(...)`, the graphs would be update according to the real data. This can be performed by the diligent reader as an exercise.

Files

- `tables/femaleClients/html/graph_view.html`

Forms

None

Database Tables

None

4.14.3 Inventory Management and Maintenance App

The Cold Chain application is a health facility maintenance application originally developed at the University of Washington in collaboration with PATH and Village Reach. It is a prototype meant to be deployed at national scale to manage refrigerator inventory and maintenance at health facilities across the country. The purpose of this is to ensure vaccines are kept at sufficiently cold temperature and maintain their efficacy.

Key Features

The Cold Chain application provides a good example of the following ODK 2 platform features:

- **Data Synchronization and Reuse**: The health facility, refrigerator, and maintenance log data set is stateful rather than the traditional model of being actively collected. If a refrigerator breaks, it is synchronized and the state is updated so other users can see this state change.
- **Custom Web Views**: Finding and viewing the details of health facilities, refrigerator models, and individual refrigerators is completely customized to the needs of this application. Custom visualizations provide statistics on the full data set, such as refrigerator age.
- **Complex Workflows**: This application does not follow the traditional data collection model. A custom workflow for managing and repairing inventory is implemented in JavaScript.
- **Mapping**: An up to date map of health facilities is rendered from the data set, and is organized into regions, to provide context for where problems may occur and to help navigate to different sites.
- **Advanced Form Design**: The Survey forms use database queries, choice

filters, and other advanced features.

- **User and Group Permissions**: This application is meant to be deployed at national scale with thousands of data points. Each user is only given access to a subset of that data that is relevant to their region, and is not be able to modify data outside of their area of responsibility.
- **Translations**: This application supports both English and Spanish locales.

Installing Cold Chain

Source code for this Data Management Application can be found in the cold-chain-demo branch ofthe App Designer repository. As with all of the ODK 2 reference applications (and the ODK 2 platform itself), the code is free and open source. Feel free to reuse, modify, or extend this application and its component parts to suit your organization's needs.

Warning: This application requires user and group permissions to be set up on the server before use. Please review the documentation for *Data Permission Filters* and *Sync Endpoint LDAP*.

You will need at least one user that is a table administrator, and to set up the groups:

- REGION_NORTH
- REGION_CENTRAL
- REGION_CENTRAL_EAST
- REGION_CENTRAL_WEST
- REGION_SOUTH
- REGION_SOUTH_EAST
- REGION_SOUTH_WEST

You should add users to the various groups, and set their default group as the region where they can edit records. For example, user *dana* might belong to groups *synchronize_tables*, *region_south* and *region_south_east* and have their default group set to *region_south_east*. In this scenario *data* can modify data in the group *region_south_east* and can see but not modify the rest of *region south* (namely, *region_south_west*).

Note: The Cold Chain application (and all other Data Management Applications provided in these docs) come with a full copy of the Application Designer they were developed in.

After you have have downloaded the application, you can set it up according to the *Application Designer setup* instructions. Similarly, you can push the application to your device

using the *Pushing and Pulling Files* instructions.

Guided Tour

A walk-through of the features and the application and an overview of how they are implemented is provided in the guide below.

Cold Chain Guided Tour

This document guides you through the *Inventory Management and Maintenance App*, named Cold Chain. This application is stateful and does not have a single workflow to follow. Therefore it is organized by area of interest, with each classification broken down into different workflows and modules contained within it. Each individual module contains both a brief description of the purpose and function of the module, as well as an overview of how that functionality is implemented.

Note: All file paths in this document are inside of the Application Designer directory. Additionally, all user defined files in a Data Management Application are inside the `app/config/` directory. For convenience this document omits these portions of the file paths.

For example, let us assume I have stored the Application Designer directory on my computer in `/home/bobsmith/workspace/app-designer`. If this guide were to reference a file as :*assets/index.html* that indicates the file located on my computer at `/home/bobsmith/workspace/app-designer/app/config/assets/index.html`.

Initial Data Set

Function

The initial data set is not part of the actual workflow of the application, but is described here to provide context for how this application is set up and how it might be used in the field. This initial data set is the full list of health facilities, refrigerators, and refrigerator types that currently exist in the health system to be modeled. The initial Deployment Architect could choose to collect this data via survey forms within the application itself (which are already provided, and used for adding data as the current state changes), but if they already have a database of health facility inventory, this might be the more convenient method.

For demonstration purposes the provided default data set also has maintenance logs that are seeded into the database.

Implementation

The initial data set is codified in `.csv` files specified in `assets/tables.init` that will be imported into the database on the initial startup. Instructions for how this works are available here: *Configuring an App at Startup*.

Each `.csv` matches the schema of its corresponding table in the database. The appropriate values are filled in for each data field. Additionally, the _group_modify, _group_privileged, and group_read_only columns include the group names to properly organize the data. The full set of custom groups in the provided data set include:

- **REGION_NORTH**
- **REGION_CENTRAL**
- **REGION_CENTRAL_EAST**
- **REGION_CENTRAL_WEST**
- **REGION_SOUTH**
- **REGION_SOUTH_EAST**
- **REGION_SOUTH_WEST**

Note: The group **GROUP_ADMIN** is also used, but is a default group provided to all Data Management Applications.

The _group_priviledged column should be set to GROUP_ADMIN for all rows in all tables. This provides full access to administrators to modify any data they choose. In `refrigerator_type.csv` the columns _group_modify and _group_read_only should both also be set to GROUP_ADMIN to restrict changes to this protected table. In `health_facility.csv` and `refrigerator.csv` the _group_modify column should be set to the most specific group the item belongs to, while the _group_read_only column should be set to the broader region. For example, a health facility in the South East region should have its _group_modify set to GROUP_REGION_SOUTH_EAST and its _group_read_only column set to GROUP_REGION_SOUTH. This will allow users in the South East region to update this facility, but only view facilities in the South West region.

Note: This group organization and permissions set up is specific to the default data set provided with the reference application. However, this is not a requirement of the ODK 2 tools. Groups could be set up to modify regions and view everything, or only read the region they belong to, or even restrict some users from modifying anything and only reading data. See *Data Permission Filters* for more details.

The JavaScript is configured to expect these groups and this set up. To use the application you will need to configure your *ODK Sync Endpoint* to have at least one table administrator. You should also add users to the various groups, and set their default group as the region where they can edit records. For example, user *dana* might belong to groups *synchronize_tables*, *region_south* and *region_south_east* and have their default group set to *region_south_east*. In this scenario *data* can modify data in the group *region_south_east* and can see but not modify the rest of *region south* (namely, *region_south_west*).

Files

- `assets/tables.init`
- `assets/csv/health_facility.csv`
- `assets/csv/refrigerators.csv`
- `assets/csv/refrigerator_types.csv`
- `assets/csv/refrigerator_types/...`

Forms

None

Database Tables

- *Health Facility*
- *Refrigerators*
- *Regrigerator Types*

Authentication

Function

The Authentication screen is the gateway to the Cold Chain application. The application filters data and restricts access based on the authenticated user's group and assigned permissions. Therefore a user with a recognized set of permissions must be authenticated before the Cold Chain application can be used, and the application will lock itself until that condition is met.

The application checks the authenticated user's credentials at startup, so if a valid user is already logged in, this screen will be completely bypassed. If not, the screen shown to the left above is shown. When you press the *Log In* button, the *Change User* screen is directly launched. This allows you to authenticate as your desired user. After you have authenticated, you can press back until you return to the Cold Chain application. The same authentication check will occur, and either valid credentials will be found or the same Authentication screen will be shown.

If the authenticated user is an administrator, the *Administrator Options* will be shown. If the authenticated user is a normal data synchronizer assigned to a valid group, the appropriate *Regions* screen will be shown.

Implementation

This screen is the first view launched when the application is opened, which means it must start with `assets/index.html`. This contains only few `<div>` elements that will be filled in by `assets/js/menu.js`. This file also makes use of the library: `assets/js/util.js`.

The Cold Chain application supports both English and Spanish locales, and the first thing `index.html` does is check the current locale, and use it to localize text throughout the page with the API `odkCommon.getPreferredLocale()` and `odkCommon.localizeText(...)`.

Next the JavaScript checks for a query parameter in the URL. Throughout this application arguments and query parameters will be passed across files as URL parameters. However, since this is the first launch, this argument will be null, which will trigger the check for the user's default group: `odkData.getDefaultGroup(...)`.

Depending on the returned default group, the next action is:

- *Valid Group*: the corresponding *Regions* page is shown.
- *Table Administrator*, the *Administrator Options* page is shown.
- *Null*: the user is either anonymous or has not been set up with a default group. The user is prompted to authenticate with a different identity.
- *Anything else*: The user does not have privileges within this application and is prompted to authenticate with a different identity.

In either of the bottom two options, the log in screen is shown. The *Log In* button launches an intent to the log in screen of ODK Services. The API `odkCommon.doAction(...)` takes the intent as an argument, which should be pointed at the *SyncActivity*. To specifically get the log in screen the bundle should include the *showLogin* value set to *true*.

When the user returns from authentication this process will repeat until a *Valid Group* or a *Table Administrator* is found.

Files

- `assets/index.html`
- `assets/js/menu.js`
- `assets/js/util.js`

Forms

None

Database Tables

None

Regions

Regions are tiered geographic groupings that contain a list of health facilities. The Cold Chain application is intended to scale to national deployments, and support for regions allows the large data set to be subdivided into manageable pieces. Furthermore, users in different regions are restricted to viewing and modifying data in their region. This has the dual advantage of clearing up the extra clutter that is not relevant to the user, and preventing them from making changes to data they should not have access to.

Regions affect health facility, refrigerator, and maintenance data. Every row in these three tables will be tied to a particular region. The refrigerator type data is not tied to particular regions.

Regions have multiple tiers. For example, the Balaka region is contained inside the South East Region, which is contained in the larger South region. This example has three tiers, but this is not required. The Chitipa region is contained in the North region (there is no East/West subdivision).

Selecting a Region

Function

The Region Selection screen is the first screen you will encounter after being authenticated and granted access. This will differ based on the authenticated user's particular group. For example, the user authenticated in the image to the left above has a default group of **REGION_SOUTH_EAST** and the user in the image to the right above has a default group of **REGION_NORTH**. This is the same first screen shown to each user, it is populated to fit their location.

Each button in the list signifies a smaller region contained in the lager region, and the list is generated dynamically based on the region shown. The number of tiers needed to get to the smallest region level will depend on the default group of the authenticated user and the number of tiers it contains.

Implementation

This screen is shown when `assets/index.html` and `assets/menu.js` execute the *authentication code* and find a user with a valid region. This will trigger the function `showSubregionButtonsAndTitle(...)` in `menu.js`, which handles the tiered subregions and creates a list of buttons based on the regions one tier below the authenticated users default group region. The default group region is parsed by the `util.getMenuOptions(...)` function in `assets/util.js`. This default group string might, itself, contain its subgroups if the group is a higher tier. If that is the case those subregions will be listed as the buttons, and when pressed they will relaunch this screen with a URL parameter indicating the subregion as the new focus region. See `addMenuButton(...)` in `menu.js` to see how `index.html` is relaunched with a new parameter.

If the sub regions are not found by the above method, then the database is queried for the sub regions: `util.getDistrictsByAdminLevel2(...)` calls `odkData.arbitraryQuery(...)` and queries the *health_facility* table.

When *leaf regions* or regions at the child tier that contain *Health Facilities* are reached, the region buttons will launch the *Region Menu* screen.

Files

- `assets/index.html`
- `assets/js/menu.js`
- `assets/js/util.js`

Forms

None

Database Tables

- *Health Facility*

Region Menu

Function

The region menu is the hub of most of the activities you might want to perform. It contains buttons to:

- *View All Health Facilities*: Launch a list of *health facilities* located in this region.
- *Filter Health Facilities By Type*: Launch a menu listing types of *health facilities*.
- *View All Refrigerators*: Launch a list of *refrigerators* located in health facilities in this region.

- *View All Refrigerators Needing Service*: Launch a list of *refrigerators* located in health facilities in this region that are marked as needing service. This is particularly useful for a maintenance worker looking for refrigerators to fix, or for administrators looking to see how many refrigerators are in need of service in a particular region.
- *View Refrigerator Models*: Launch a list of *refrigerator types* contained in the region.

Implementation

The HTML file `assets/leafRegion.html` contains all five button definitions as they are the same no matter what region is selected or user is authenticated. The corresponding JavaScript file `assets/js/leafRegion.js` localizes the strings, fills in the region name on top (from `util.getQueryParameter(...)`), and adds actions to each button.

The actions are the same for each region, but the query parameters are passed along to the next view. The are generated with `util.getKeysToAppendToColdChainURL(...)`. The actions are:

- *View All Health Facilities*: Open the *Lists of Health Facilities* to the *Map View* with `odkTables.openTableToMapView(...)`. The query specifies to select all rows matching the current region from the *health_facility* table.
- *Filter Health Facilities By Type*: Open the *Lists of Health Facilities* to the type filtered navigation menu with `odkTables.launchHTML(...)`. This API is used for customized web views.
- *View All Refrigerators*: Open the *Lists of Refrigerators* with `odkTables.launchHTML(...)`. This does not take a query as a parameter as a normal *List View* would, but rather performs its own queries as needed based on the URL parameters passed and the user interactions on the page.
- *View All Refrigerators Needing Service*: Same as above but with appropriate URL parameters and HTML file arguments.
- *View Refrigerator Modesl*: Open the *Lists of Refrigerator Types* with `odkTables.openTableToListView(...)`

Files

- `assets/leafRegion.html`
- `assets/js/leafRegion.js`
- `assets/js/util.js`

Forms

None

Database Tables

None

Health Facilities

Health Facilities in the Cold Chain application contain the state of real world health facilities. They include general information about their status, power, location, and vaccine stock, as well as an active record of their refrigerator inventory. The intention of this application is that a user or administrator could navigate the application to a health facility and learn its basic advantages and disadvantages, and if it needs attention.

Lists of Health Facilities

Function

The list of health facilities is presented as a *Map View*. The list portion of the view provides a clickable list of health facilities: clicking a facility will launch the *Health Facility Menu*. The map portion renders that list according to the location data stored in those facilities. This can be used to navigate to facilities or to find a facility in which you may know the location better than the name.

- **View All Health Facilities**: This screen is reached by pressing the *View All Health Facilities* button on the *Region Menu* page. It lists every health facility located inside of the specified region. It is pictured above.
- **Filter Health Facilities By Type**: This screen is reached by pressing the *Filter Health Facilities by Type* button on the *Region Menu* page. It lists each type of health facility contained in the region, and a the number of health facilities that match the type:

When one of those health facility types is selected, a list similar to the full health facility list above is rendered, but only containing health facilities within the specified region that match the chosen type. The image below is the list of *Regional Vaccine Store* type facilities in the Balaka region:

Implementation

- **View All Health Facilities**: This path to the health facilities list launches `tables/health_facility/html/hFacility_list.html`. It contains a `<div>` for the wrapper and for the list of facilities, which is populated with `tables/health_facility/js/hFacility_list.js`. This JavaScript file follows a standard *List View* pattern: it retrieves the query data with `odkData.getViewData(...)`, creates list items for each of the rows, adds them to the HTML view, includes a link to `odkTables.openDetailView(...)` for each list item, and handles paging with `resumeFn(...)` called with an index.
- **Filter Health Facilities By Type**: This path launches `assets/filterHealthFacilitiesByType.html`. It contains `<div>` tags for buttons that will be filled in by `assets/js/filterHealthFacilitiesByTypes.js`. This file uses the provided district to construct a query, which is executed by `util.getFacilityTypesByDistrict(...)` in `assets/js/util.js` and calls `odkData.arbitraryQuery(...)`. This query runs on the *health_facility* table and finds all facilities in the region, groups them by type, and returns the count in each type. These results are then fed back into `filterHealthFacilitiesByType.html` which creates the buttons. Each button, if pressed, will use `odkTables.openTableToMapView(...)` to launch the same `hFacility_list.html` used above, but with the query narrowed by facility type.

Files

- `tables/health_facility/html/hFacility_list.html`
- `tables/health_facility/js/hFacility_list.js`
- `assets/filterHealthFacilitiesByType.html`
- `assets/js/filterHealthFacilitiesByType.js`
- `assets/js/util.js`

Forms

None

Database Tables

- *Health Facility*

Health Facility Menu

Function

The Health Facility Menu is a *Detail View* that lists all the information about the chosen health facility. This includes *Basic Facility Information*, *Power Information*, *Location Information*, and *Stock Information*. If any of this information is out of date or needs to be modified, the *Edit Health Facility* button launches an ODK Survey form that allows you to modify these values:

The prompts in this form will be prepopulated with the values shown on the menu page. All correct values can be safely skipped, so you can edit only the fields that need to be corrected.

The menu also provides a button to view the *Refrigerator Inventory*. This will launch the *list of refrigerators* contained within this health facility.

The *Add Refrigerator* button will launch a Survey form to create a new refrigerator. When the form is completed, this refrigerator will automatically be added to the inventory of this health facility and organized into the containing region.

Implementation

The *Detail View* for a health facility is defined by `tables/health_facility/html/health_facility_detail.html`. This file lists each user interface element (including all the data values of the health facility as well as the buttons). These elements contain their labels, and the values are filled in by `tables/health_facility/js/health_facility_detail.js`.

After localizing its text, this JavaScript retrieves the health facility data with the standard `odkData.getViewData(...)` call. It also makes a call to `odkData.query(...)` to the *refrigerators* table and finds all refrigerators that belong to this health facility. These two data sets are combined to fill in the fields on the detail view and the size of the refrigerator inventory on the button.

If the *Edit Facility* button is pressed, `odkTables.editRowWithSurvey(...)` is called for

the form *Health Facility* and pointed at this particular row ID. This form can be viewed at `tables/health_facility/forms/health_facility/health_facility.xlsx`. It condenses its prompts into only a few screens with extensive use of begin screen and end screen clause values. Notice that all text in this form also has Spanish translations provided. The form contains many static select_one prompts with their choices defined in the *choices* worksheet. Additionally, the Admin Region select_one_dropdown has its choices populated dynamically from a query defined in the *queries* worksheet. This list is then filtered by the value in the choice_filter column back in the *survey* worksheet. The *settings* worksheet contains the supported languages in addition to the normal settings. The *properties* worksheet defines the default *Detail View*, *List View*, and *Map View* files and settings. The *model* links the region levels from to the database.

If the *Refrigerator Inventory* button is pressed, `odkTables.launchHTML(...)` is called to launch the *Lists of Refrigerators* screen with this health facility as the filter.

If the *Add Refrigerator* button is pressed, `odkTables.addRowWithSurvey(...)` is called for the *Refrigerators* form. The permission and group values of the current health facility are passed in as arguments as well, in order to create this new refrigerator with the same values. This form can be viewed at `tables/refrigerators/forms/refrigerators/refrigerators.xlsx`. It is similar to the *Health Facility* form: short and compressed into a small number of screens. The refrigerator model and health facility choices are both queried from the database (see the *queries* worksheet). The necessary fields are linked in the *models* worksheet. The *choices*, *properties*, and *settings* worksheets are similar to those found in the *Health Facility* form, but with their own values.

Files

- `tables/health_facility/html/health_facility_detail.html`
- `tables/health_facility/js/health_facility_detail.js`
- `assets/js/util.js`
- `tables/health_facility/forms/health_facility/health_facility.xlsx`
- `tables/health_facility/forms/health_facility/regions2-3.csv`
- `tables/refrigerators/forms/refrigerators/refrigerators.xlsx`
- `tables/refrigerators/forms/refrigerators/refrigerators.xlsx`
- `tables/refrigerators/forms/refrigreators/regions1-2.csv`
- `tables/refrigerators/forms/refrigreators/regions2-3.csv`

Forms

- *Health Facility* with form ID: *health_facility*

Database Tables

- *Health Facility*
- *Refrigerators*

Refrigerators

Refrigerators are a key element of the Cold Chain application. Their working status is necessary for keeping vaccines cold and effective. This application focuses on providing easy access to the working status of lists of refrigerators organized in a multitude of different ways. It also tracks basic information about the refrigerator, such as its age and model.

Refrigerators belong to *Health Facilities* and they contain *Maintenance Records*.

Lists of Refrigerators

Function

A Refrigerator List contains a clickable list of refrigerators: clicking the refrigerator will open the *Refrigerator Menu*. The list can be searched by the refrigerator's ID, the tracking ID, the health facility name, or the health facility ID:

The search string does not need to be a perfect match. Substrings and approximate matches can be searched and all matching records will be displayed. For example, if you searched *225* then you might get back refrigerators with ID *22500172*, *22500035*, and *22500032*.

This page is paginated by default to 10 refrigerators per page. This can be adjusted to 20, 50, 100, or 1000 by selecting the option from the drop menu. To navigate between pages of refrigerators, use the *Next* and *Prev* buttons.

Tapping the *Edit* button will launch the full Survey form for this refrigerator. Each field will be prepopulated with the values shown in the menu, so that only the values that are incorrect need to be filled in.

These lists can be organized a number of ways:

- **By Region**: This lists all refrigerators in the region and is launched by pressing the *View All Refrigerators* button on the *Region Menu*. This is what is shown above.
- **By Health Facility**: This lists all refrigerators in a particular health facility. It is launched by pressing the *Refrigerator Inventory* button on the *Health Facility Menu*.

- **By Type**: This lists all refrigerators in a particular region organized by type. This is launched by pressing the *View All *Model ID* Refrigerators* button on a *Refrigerator Type Menu*.

- **Needing Service**: This lists all refrigerators in a particular region that are in need of service. This is launched by pressing the *View All Refrigerators Needing Service* button on the *Region Menu*.

Implementation

The refrigerator lists launched **By Region**, **By Health Facility**, and **By Type** all use `config/tables/refrigerators/html/refrigerators_list.html` and achieve their different lists by passing different query parameters. This file defines the search form, the pagination drop menu, and the JavaScript functions to call on button presses. All the rest of the user interface is added dynamically in `config/tables/refrigerators/js/refrigerators_list.js`. However, this file only handles populating the user interface elements defined in `refrigerators-list.html`. All of the logic is handled by the shared library file `config/assets/js/list_view_logic.js`. This file is discussed in the following subsection: *The list_view_logic.js library*.

The refrigerator list launched by **Needing Service** uses `config/tables/refrigerators/html/refrigerators_service_list.html` and `config/tables/refrigerators/js/refrigerators_service_list.js`, but these files work nearly identically to their `refrigerators_list.*`. The only difference is the `listQuery` variable that defines the SQL query to be run. Both files join the *Refrigerators*, *Health Facilities*, and *Refrigerator Types* tables in order to support filtering and sorting on facility name, facility ID, tracking ID, and refrigerator ID (see the `searchParams` variable). The `refrigerators_service_list.js`

file differs in that it adds arguments for refrigerator maintenance priority.

The `list_view_logic.js` library

This library handles the queries, ordering, search, and pagination for all the search *List Views* in the Cold Chain application. In this section the calling files are `refrigerators_list.js` and `refrigerators_service_list.js`, but there are others as well. For the rest of this section I will refer to these as the caller.

When the caller is initializing, it will use the *set* functions to build state. First the table ID must be set with `setTableId(...)`. Then the query parameters with `setListQuery(...)`, `setListQueryParams(...)`, and `setSearchParams(...)`. And finally the user interface elements need to be supplied with `setListElement(...)`, `setSearchTextElement(...)`, and so on, to allow the `list_view_logic.js` file to read and write to them directly.

After state is initialized, the `resumeFn(...)` can be called. This function uses session variables (via `odkCommon.setSesionVariable(...)` and `odkCommon.getSessionVariable(...)` to track search terms, query keys, and pagination indices. It uses these values to build SQL queries and then runs them with a series of `odkData.arbitraryQuery(...)` commands to count the matching records and then retrieve the appropriate subset to display on the page. The results of that final query are used to create the list elements and populated them onto the page. Each list element contains a `odkTables.openDetailView(...)` command embedded in it. This works in a generic file like this because the default *Detail View* for each of these tables has been set in the *settings* page of the corresponding `.xlsx` file.

There is also more complex logic to handle the *Edit* and *Delete* buttons. The file must ensure the authenticated user has the requisite permissions for each record before displaying the button. If they do, and the button is pressed, the functions `odkTables.editRowWithSurvey(...)` and `odkData.deleteRow(...)` are called, respectively.

There are controls for the *Next* and *Prev* navigation buttons that ensure they do not go beyond the bounds of the full result set. Each time they are pressed, the `resumeFn(...)` is called again to re-query and redraw the results. Similarly, the *Search* button parses the text of the search, constructs a new query, and calls `resumeFn(...)`. All of these functions communicate their parameters for the redraw through session variables.

Files

- `config/tables/refrigerators/html/refrigerators_list.html`
- `config/tables/refrigerators/js/refrigerators_list.js`
- `config/tables/refrigerators/html/refrigerators_service_list.html`
- `config/tables/refrigerators/js/refrigerators_service_list.js`
- `config/assets/js/list_view_logic.js`

Forms

None

Database Tables

- *Refrigerators*
- *Health Facilities*
- *Refrigerator Types*

Refrigerator Menu

Function

The Refrigerator Menu is a *Detail View* that shows all the information about the particular refrigerator. Notable fields include *Status* and *Date Serviced*.

It also contains a number of buttons:

- *View Model Information*: Launches the corresponding *Refrigerator Type Menu*.
- *View Facility Information*: Launches the *Health Facility Menu* of the facility this refrigerator belongs to.
- *Add Maintenance Record*: Launches a Survey form to add a new *Maintenance Records*. This record will be associated with this refrigerator and appear in future logs. This is meant to be filled out after a refrigerator is serviced.
- *View All Maintenance Records*: Launches a *Lists of Maintenance Records* of all records associated with this particular refrigerator. It serves as a full service history of this unit.
- *Edit Refrigerator Status*: Launches a Survey form that modifies only the service related details of this refrigerator. To be pressed when this refrigerator breaks or receives maintenance.
- *Edit Refrigerator*: Launches the full Survey form for this refrigerator. Each field will

be prepopulated with the values shown in the menu, so that only the values that are incorrect need to be filled in.

Implementation

The *Detail View* for a refrigerator is defined by `tables/refrigerators/html/refrigerators_detail.html`. This file lists each user interface element (including all the data values of the refrigerator as well as the buttons). These elements contain their labels, and the values are filled in by `tables/refrigerators/js/refrigerators.js`.

After localizing its text, this JavaScript retrieves the refrigerator data with the standard `odkData.getViewData(...)` call. It also makes `odkData.query(...)` calls to the *Health Facility*, *Refrigerator Types* and *Maintenance Logs* tables. All of these resulting data sets are combined to fill in the display fields on the detail view.

If a button is pressed:

- *View Model Information*: Launches `odkTables.openDetailView(...)` to *Refrigerator Type Menu*.
- *View Facility Information*: Launches `odkTables.openDetailView(...)` to *Health Facility Menu*.
- *Add Maintenance Record*: Launches `odkTables.addRowWithSurvey(...)` to the *Maintenance Logs* form. The permission and group values of the current refrigerator are passed as arguments as well, in order to create this maintenance record with the same values. This form can be viewed at `tables/maintenance_logs/forms/maintenance_logs/maintenance_logs.xlsx`. This brief form only contains two screens. There is an if clause that is set to never trigger, because the refrigerator_id will already be supplied by the caller. The rest of this form functions similarly to the rest of the forms in this application.
- *View All Maintenance Records*: Launches `odkTables.launchHTML(...)` to *Lists of Maintenance Records*.
- *Edit Refrigerator Status*: Launches `odkTables.editRowWithSurvey(...)` to the *Refrigerator Status* form. This form can be found at `tables/refrigerators/forms/refrigerator_status/refrigerator_status.xlsx`. This form writes to the *Refrigerators* table the same as the *Refrigerators* form does, but only presents a subset of the data fields. It displays a single screen of prompts relating to the status of the refrigerator. These will be prepopulated and only need be updated as necessary. This mapping is set up in the *settings* worksheet.
- *Edit Refrigerator*: Launches `odkTables.editRowWithSurvey(...)` to the *Refrigerators* form. This performs similarly to the above option, but presents the data fields of the entire table. The form is discussed in more detail in *Health Facilities Menu Implementation* under the *Add Refrigerator* option.

Files

- `tables/refrigerators/html/refrigerators_detail.html`
- `tables/refrigerators/js/refrigerators_detail.js`
- `config/assets/js/util.js`
- `tables/maintenance_logs/forms/maintenance_logs/maintenance_logs.xlsx`
- `tables/refrigerators/forms/refrigerator_status/refrigerator_status.xlsx`
- `tables/refrigerators/forms/refrigerators/refrigerators.xlsx`

Forms

- *Maintenance Logs* with form ID *maintenance_logs*
- *Refrigerator Status* with form ID *refrigerator_status*
- *Refrigerators* with form ID *refrigerators*

Database Tables

- *Refrigerators*
- *Health Facility*
- *Refrigerator Types*
- *Maintenance Logs*

Maintenance Records

Maintenance Records detail service performed on a particular refrigerator. Individually they are a brief record of service performed, and taken together they compose the service history of a refrigerator.

Lists of Maintenance Records

Function

A List of Maintenance Records is similar to a *Lists of Refrigerators*. It is launched by pressing the *View All Maintenance Records* button on the *Refrigerator Menu*.

The list portion contains each maintenance record in the log for a particular refrigerator, with the date serviced highlighted. Clicking that list item will open the *Maintenance Record Menu*.

Records can be searched by refrigerator ID. This page is paginated by default to 10 records per page. This can be adjusted to 20, 50, 100, or 1000 by selecting the option from the drop menu. To navigate between pages of maintenance records, use the *Next* and *Prev* buttons.

Tapping the *Edit* button will launch the Survey form for this maintenance record. Each field will be prepopulated with the values shown in the menu, so that only the values that are incorrect need to be filled in.

Implementation

The maintenance records list uses the files `tables/maintenance_logs/html/maintenance_logs_list.html` and `tables/maintenance_logs/js/meaintenance_logs_list.js` similarly to the *refrigerators list*.

The key differences are the `listQuery` and `searchParams` variables that define the values that will populate the same user interface. This file's versions of `listQuery` finds all maintenance logs that match the refrigerator ID and this versions of `searchParams` searches for matching refrigerator IDs.

That logic that implements that user interface is discussed in *The list_view_logic.js library*.

Files

- `tables/maintenance_logs/html/maintenance_logs_list.html`
- `tables/maintenance_logs/js/maintenance_logs_list.js`
- `config/assets/js/list_view_logic.js`

Forms

None

Database Tables

- *Maintenance Logs*
- *Refrigerators*

Maintenance Record Menu

Function

The Maintenance Record Menu is a *Detail View* that lists the full account of the service, including the *Reason Not Working*, the *Date Serviced*, *Spare Parts*, and other details.

It also includes an *Edit Log* button, which launches the Survey form for this maintenance record. Each field will be prepopulated with the values shown in the menu, so that only the values that are incorrect need to be filled in.

Implementation

The *Detail View* for a maintenance record is defined by `tables/maintenance_logs/html/maintenance_logs_detail.html`. This file lists each user interface element (including all the data values of the maintenance record as well as the buttons). These elements contain their labels, and the values are filled in by `tables/maintenance_records/js/maintenance_records_detail.js`.

After localizing its text, this JavaScript retrieves the maintenance log data with the standard `odkData.getViewData(...)` call. It also makes an `odkData.query(...)` call to the

Refrigerators table. The resulting data sets are combined to fill in the display fields on the detail view.

The *Edit Log* and *Delete Log* buttons are dynamically hidden or shown depending on the privileges of the authenticated user.

If the *Edit Log* button is pressed, the *Maintenance Logs* form is launched with `odkTables.editRowWithSurvey(...)`. The forms `.xlsx` file is located at `tables/maintenance_logs/forms/maintenance_logs/maintenance_logs.xlsx`. This form is discussed in the *refrigerator menu implementation section* under the *Add Maintenance Record* bullet.

If the *Delete Log* button is pressed, `odkData.deleteRow(...)` is called to remove the record from the database.

Files

- `tables/maintenance_logs/html/maintenance_logs_detail.html`
- `tables/maintenance_logs/js/maintenance_logs_detail.js`
- `assets/js/util.js`
- `tables/maintenance_logs/forms/maintenance_logs/maintenance_logs.xlsx`

Forms

- *Maintenance Logs* with form ID *maintenance_logs*

Database Tables

- *Maintenance Logs*
- *Refrigerators*

Refrigerator Types

Refrigerator Types represent the different models of refrigerators available. This is convenient for adding a new refrigerator to a health facility, as the model will already include information about the refrigerator that might not be obvious to the worker installing the unit. It also provides an easy reference if spare parts are needed or the model itself is needed for any reason.

Lists of Refrigerator Types

Function

A List of Refrigerator Types is similar to a *Lists of Refrigerators*. It is launched by pressing the *View Refrigerator Models* button on the *Region Menu*.

The list portion contains each refrigerator type. Clicking that list item will open the *Refrigerator Type Menu*. The list can be searched by the refrigerator type ID, catalog ID, or the manufacturer. A substring of the search string can be provided and all matching records will be displayed.

The list items each show a picture of the refrigerator model to ease navigation and provide a more familiar reference than the model name.

This page is paginated by default to 10 refrigerator types per page. This can be adjusted to 20, 50, 100, or 1000 by selecting the option from the drop menu. To navigate between pages of refrigerator types, use the *Next* and *Prev* buttons.

Implementation

The refrigerator types list uses the files `tables/refrigerator_types/html/regrigerator_types_list.html` and `tables/refrigerator_types/js/refrigerator_types.js` similarly to the *refrigerators list*.

The key differences are the `listQuery` and `searchParams` variables that define the values that will populate the same user interface. This file's versions of `listQuery` performs a simple `SELECT *` from the *Refrigerator Types* table and this versions of `searchParams` searches for matching catalog ID, manufacturer, or model ID.

That logic that implements that user interface is discussed in *The list_view_logic.js library*.

Files

- `tables/refrigerator_types/html/refrigerator_types_list.html`
- `tables/refrigerator_types/js/refrigerator_types_list.js`

Forms

None

Database Tables

- *Refrigerator Types*

Refrigerator Type Menu

Function

The Refrigerator Type Menu is a *Detail View* that lists all the model information about the particular refrigerator type, and shows a picture if available.

It also has a *View all *Model ID* Refrigerators* that shows the number of refrigerators in the region with this particular type. Tapping that button will launch a *Lists of Refrigerators* containing all refrigerators in that region of that type.

Implementation

The *Detail View* for a refrigerator type is defined by `tables/refrigerator_types/html/refrigerator_types_detail.html`. This file lists each user interface element (including all the data values of the refrigerator type as well as the buttons). These elements contain their labels, and the values are filled in by `tables/refrigerator_types/js/refrigerator_types_detail.js`.

After localizing its text, this JavaScript retrieves the refrigerator log data with the standard `odkData.getViewData(...)` call. It also makes an `odkData.query(...)` call to the *Refrig-*

erators table. The resulting data sets are combined to fill in the display fields on the detail view.

If the *View all *Model ID* Refrigerators* button is pressed, `odkTables.launchHTML(...)` is called to launch *Lists of Refrigerators*.

Files

- `tables/refrigerator_types/html/refrigerator_types_detail.html`
- `tables/refrigerator_types/js/refrigerator_types_detail.js`
- `assets/js/util.js`

Forms

None

Database Tables

- *Refrigerator Types*
- *Refrigerators*

Administrator Options

Administrator Options are available when the authenticated user is a Table Administrator. They provide the admin with enhanced permissions, including viewing the entire data set, creating data visualizations, deleting records, and adding new health facilities.

Administrator Options Menu

Function

The Administrator Options Menu is the hub that launches the special actions the administrator can perform. It is only accessible by authenticated Table Administrators. This screen is accessed by pressing the *Administrator Options* button the main menu.

The administrator has access the full hierarchy of regions. Above the *Administrator Options* button is the list of all the highest tier regions. Tapping any of those individual options will filter the data into that region. They admin can keep advancing through the regions until they reach the leaf tiers, which is the same *Regions* interface.

Implementation

This screen is shown the `assets/index.html` and `/assets/menu.js` execute the *authentication code* and find a user that is a Table Administrator. This will trigger the function `showSubregionButtonsAndTitle(...)` to show the top level regions (see *Regions* for details), and the `addMenuButton(...)` function to add the *Administrator Options Button*.

When the *Administrator Options* button is pressed, the function `odkTables.launchHTML(...)` launches `assets/coldchaindemo.html`. This file defines the four option buttons, and `assets/js/coldchandemo.js` handles logic that they trigger.

- *View Health Facilities*: Launches *View Health Facilities* with `odkTables.launchHTML(...)`.
- *View Inventory*: Launches *Inventory* with `odkTables.launchHTML(...)`.
- *View Refrigerator Models*: Launches *Refrigerator Types* with `odkTables.launchTableToListView(...)`.
- *Add Health Facility*: Launches *Add Health Facility* with `odkTables.launchHTML(...)`.

Files

- `assets/index.html`
- `assets/js/menu.js`
- `assets/js/util.js`
- `assets/coldchaindemo.html`
- `assets/js/coldchaindemo.js`

Forms

None

Database Tables

- *Health Facility*

View Health Facilities

Function

The administrator can view the full list of health facilities, unfiltered by region. This gives the administrator full control investigate, modify, or delete any facility in the data set. They can choose to find a facility with one of two methods:

- *Filter By Type*:

This option presents an interface similar to the *Filter Health Facilities By Type* option in the *Lists of Health Facilities*. This version functions the same, but includes all facilities in the system. Selecting a type brings up the normal *Map View* list.

- *Search By Name/ID*:

This interface presents a search list that looks and behaves the same as *Lists of Refrigerators*. It includes *Delete* buttons on each entry. The search box accepts facility ID and facility names.

Implementation

The root level HTML file for this option is `assets/filterHealthFacilities.html`. It defines the two buttons and `assets/js/filterHealthFacilities.js` handles their logic.

- *Filter By Type*: uses `odkTables.launchHTML(...)` to launch `assets/filterHealthFacilitiesByType.html` and `assets/js/filterHealthFacilitiesByType.js`. This JavaScript file uses `util.getFacilityTypesByDistrict(...)` from `assets/js/util.js` to create the facility type buttons and add their facility counts. Tapping one of the facility type buttons will use `odkTables.openTableToMapView(...)` to launch *Lists of Health Facilities*. This workflow is similar to the *Filter Health Facilities By Type* option on that screen.
- *Search By Name/ID*: uses `odkTables.launchHTML(...)` to launch `tables/health_facility/html/health_facility_lists.html` and `tables/health_facility/html/health_facility_lists.js`. These files use the same search list pattern found in *Lists of Refrigerators*. See *The list_view_logic.js library* for details on how `assets/js/list_view_logic.js` renders this user interface. The

`listQuery` value selects all health facilities from the *Health Facility* table. The `searchParams` sets the search fields to facility ID and facility name.

Files

- `assets/filterHealthFacilities.html`
- `assets/js/filterHealthFacilities.js`
- `assets/filterHealthFacilitiesByType.html`
- `assets/js/filterHealthFacilitiesByType.js`
- `tables/health_facility/html/health_facility_list.html`
- `tables/health_facility/js/health_facility_list.js`
- `assets/js/list_view_logic.js`
- `assets/js/util.js`

Forms

None

Database Tables

- *Health Facility*

Inventory

Function

The Inventory option provides two visualizations of the state of the data set, both of which can be customized to chosen parameters.

- *Refrigerator Age*

The refrigerator age visualization presents a bar chart of the current stock of refrigerators, grouped by age. This can be useful as an assessment of the quality of the stock and as an estimate of maintenance demands. This graph can be filtered by region, facility type, and power source. With this option the administrator might compare the age distribution of refrigerators in the North and the South regions when allocating upgrade budgets.

- *Facility Grid Power Available*

The grid power visualization presents a pie chart comparing the ratios of power options available. This can be filtered by region and facility type.

The data sets to be graphed are filtered with a set of drop menus that can be chosen to specify the desired data set.

Implementation

The root level HTML file for this option is `assets/filterInventory.html`. It defines the two buttons and `assets/js/filterInventory.js` handles their logic.

Refrigerator Age uses `odkTables.launchHTML(...)` to launch `assets/filterFrigInventoryForAge.html` an `assets/js/filterFrigInventoryForAge.js`. The HTML file defines the three drop menus. The values specified by these drop menus are read and used as query parameter arguments when launching `assets/graphFrigInventoryForAge.html` and `assets/js/graphFrigInventoryForAge.js`. This JavaScript file uses query parameters provided the caller to construct a SQL query run an `odkData.query(...)` call on the *Health Facility* table. The result of this call are used to construct a new query that finds refrigerators that match health facilities with an `odkData.arbitraryQuery(...)` call on the *Refrigerators* table. When these results return, the `frigHistogramByAge()` function uses that data and the `D3` library to render the bar chart.

Facility Grid Power Available follows the same pattern as *Refrigerator Age* to present drop menus and use their values as query parameters. The file that renders this graph is `assets/js/graphFacilityInventoryForGridPower.js`. This file also operates similarly to `graphFrigInventoryForAge.js` but only performs a single query on the *Health Facilities* table. That data set is used, along with `D3` by the `displayHealthFacilityGridPower()` function to render the pie chart.

Files

- `assets/filterInventory.html`
- `assets/js/filterInventory.js`
- `assets/filterFrigInventoryForAge.html`
- `assets/js/filterFrigInventoryForAge.js`
- `assets/filterFacilityInventoryForGridPower.html`
- `assets/js/filterFacilityInventoryForGridPower.js`
- `assets/graphFrigInventoryForAge.html`
- `assets/js/graphFrigInventoryForAge.js`
- `assets/graphFacilityInventoryForGridPower.html`
- `assets/js/graphFacilityInventoryForGridPower.js`
- `assets/js/util.js`

Forms

None

Database Tables

- *Health Facility*
- *Refrigerators*

Refrigerator Types

The Refrigerator types list is identical to the interface presented in the *Lists of Refrigerator Types* guide. The button is included here as a convenience. See the linked documentation for details.

Add Health Facility

Function

The Add Health Facility interface provides a method for an administrator to add a new health facility to the data set. The administrator must specify the region that should contain the facility (the region must be a leaf tier, it cannot contain other regions). When the *Add Facility* button is pressed, a form is launched to fill in the details of the facility.

Implementation

The root level HTML file for this option is `assets/addHealthFacility.html`. It defines the button and drop menu and `assets/js/addHealthFacility.js` handles their logic.

The JavaScript file reads the value from the drop menu and uses it to construct the `defaults` argument to `odkTables.addRowWithSurvey(...)`. The variable also includes the group permissions. The form launched is `tables/health_facility/forms/health_facility/health_facility.xlsx`

This form resembles many of the other forms in this application. Mostly select_one prompts are grouped into screens. The region choices are populated by a query from the *queries* work-

sheet. The *settings*, *properties*, and *model* worksheets all contain their typical values, setting the form and table IDs, setting the default view files, and mapping to the database, respectively. The *properties* file includes security properties including unverifiedUserCanCreate and defaultAccessOnCreation that restrict which users can use this form.

Files

- `assets/addHealthFacility.html`
- `assets/js/addHealthFacility.js`
- `assets/js/util.js`
- `tables/health_facility/forms/health_facility/health_facility.xlsx`

Forms

- *Health Facility* with form ID *health_facility*

Database Tables

- *Health Facility*

4.15 Deployment Architect Advanced Topics

This section covers advanced topics useful to Deployment Architects. A Deployment Architect is an author of a data management application or a consumer of collected data. This person might create forms and edit Javascript on their computer to deploy to the Android device. Or they might download data from the server and use Excel to perform analysis. Examples include technical staff and data analytics staff.

Other perspective definitions can be found *here*.

4.15.1 Data Permission Filters

Limitations

Traditional access control frameworks provide strong protections for data and the management of which users can modify that data. The permission filtering introduced in ODK 2.0 is weaker. When syncing devices with the server, all data rows for all data tables are currently synced and shared across all devices. Every device gets a full copy of all data. Permission

filtering enables a supervisor to restrict the visibility of that data and to manage who can modify or delete the data through the programmatic means provided by the ODK 2.0 tools.

This is weaker than traditional access control frameworks in that application designers can:

- Circumvent via software. There are specific ways in which application designers can write their applications to defeat these filters. When those mechanisms are not employed, permission filtering provides equivalent policy enforcement to that of a traditional access control framework.
- Circumvent via external access. The data and attachments are stored as plaintext on the device. Anyone can copy this data off of the device and access it, or write their own apps and directly modify it.

It is important to understand these limitations when designing your applications.

Overview

By default, all tables can be altered by all users.

The ODK 2.0 data access filtering mechanism relies on five interacting features:

- Verified user identities
- Verified user capabilities
- Table-level security configuration (whether data in the table can be modified by unprivileged users).
- Row-level access filters (to specify whether a row is visible to a given user, whether the user can modify the row's data value, and whether the user can change this row's access filters).
- Sync status of the individual row.

Verified User Identities

Enforcing restrictions on who can see or modify data requires that the identity of the user has been verified.

When configuring the *Server Settings*, any changes to any of the settings (such as the server URL, type of credential (or anonymous access), username, password or Gmail account) will clear any prior user identity and capability information and flag the user identity as unverified.

When leaving the *Server Settings* screen, a user-verification screen will then be presented (unless no server sign-on credential is specified, in which case anonymous access to the server will be attempted):

Clicking the *Verify User Permissions* button on this screen will initiate a series of requests to the configured server. These requests verify that the server URL is correct, that the server works with this application name, and then verify the server sign-on credential that has been configured on the *Server Settings* page.

> **Warning:** If the server sign-on credential is rejected, the user identity will be flagged as unverified and any further interactions on the device will be performed as if by an anonymous user.

Verified User Capabilities

As part of the user-verification process, once the user's identity has been verified, the list of groups to which this user belongs and the capabilities (roles) assigned to that user are downloaded from the server. These are cached on the device for use during data access filtering until the user logs out of the ODK 2.0 tools on the device or a different server sign-on credential is specified.

For the purposes of the data access filtering mechanism, there are 4 user capabilities of interest:

- **ROLE_USER** – a user who is able to verify their identity.

- **ROLE_SYNCHRONIZE_TABLES** – a user who is able to execute the sync protocol.
- **ROLE_SUPER_USER_TABLES** – a privileged user who can edit all rows, change how rows are visible, and change who has special permission to edit a given row.
- **ROLE_ADMINISTER_TABLES** – a privileged user who can *Reset App Server* and who can edit all rows, change how rows are visible, and change who has special permission to edit a given row.

The first two of these identify users that are unprivileged. These users may be granted privileges to individual rows by being designated the owner of that row or through their membership in one or more user groups identified in the row's access filter columns.

The second two of these identify privileged users that have full control of the device. Additionally, the last of these capabilities (**ROLE_ADMINISTER_TABLES**) identifies a user that can alter the configuration of the Cloud Endpoint.

Application designers that wish to restrict access by unverified users or manage anonymous access to the server can further restrict table and row access in these scenarios.

Row Access Filter Columns

Management of which unprivileged users can see, modify or manage access to a given row is controlled through five access filter columns. The first of these columns specifies the access to the row that is granted to all unprivileged users. The second identifies the owner of this row. Row owners have modify privileges on a row. The other three are either null or specify a user group that is granted that specific access right:

- **_DEFAULT_ACCESS** – one of HIDDEN, READ_ONLY, MODIFY or FULL.
- **_ROW_OWNER** – this user has FULL privileges on this row.
- **_GROUP_READ_ONLY** – a user who is a member of this group will be able to read this row of data
- **_GROUP_MODIFY** – a user who is a member of this group will be able to read and modify this row of data but not delete it.
- **_GROUP_PRIVILEGED** – a user who is a member of this group will be able to read, modify, delete and change privileges on this row of data.

Note: Privileged users are not governed by these settings – they have unlimited access to all tables on the device.

Individual users can belong to any number of groups, enabling arbitrarily complex row-level access management. Users may also be assigned a default group. Management of group

memberships is dictated by the server being used. Refer to the *ODK Cloud Endpoints* for the capabilities of the different servers. More detail will be given regarding these filter columns in the *Row-level Access Filters* section.

Obtaining a User's Groups and Roles

Inside ODK Survey and ODK Tables web pages, the groups and roles of the current verified user are available in JavaScript via the API:

```
odkData.getRoles(function(result) {
  var roles = result.getRoles();
  // roles is an array of capabilities granted to the verified user.
  // It will be null for anonymous and unverified users.
}, function(errorMsg) {
  // error handler
});
```

Obtaining a User's Default Group

Inside ODK Survey and ODK Tables web pages, the default group of the current verified user is available in JavaScript via the API:

```
odkData.getDefaultGroup(function(result) {
  var defaultGroup = result.getDefaultGroup();
  // defaultGroup is null or a string
}, function(errorMsg) {
  // error handler
});
```

Note: Default groups are not directly used within the ODK 2.0 framework. These are provided for use by an application designer when crafting their application.

Obtaining Information About Other Users

Whenever the server is contacted to verify a user's identity, if the user is determined to be a privileged user, the server will, additionally, provide a list of all users configured on the server and all of the groups and roles assigned to those users. This list can be useful when performing task assignments via assigning row ownership.

This list will contain entries of the form:

```
{
  user_id: "verified_identity_token",
  full_name: "content of the Full Name field on the server",
  default_group: "default group of the user"
  roles: [...]
}
```

The *Full Name* field on the server (on the *Site Admin* → *Permissions* sub-tab) is provided here to allow super-users and administrators to select people by *name*. *user_id* should be stored in the _ROW_OWNER column to assign ownership to this user. The list of roles (and groups) is provided to allow super-users and administrators to choose users based upon their capabilities.

If the user has been assigned to a default group it will be provided. Default groups are not directly used within the ODK 2.0 framework. These are provided for use by an application designer when crafting their application.

Inside ODK Survey and ODK Tables web pages, the list of all configured users is available in JavaScript via the API:

```
odkData.getUsers(function(result) {
  var users= result.getUsers();
  // users is an array of the above objects.
  // It will be null for anonymous and unverified users.
  // It will be a singleton list if the user lacks permissions.
}, function(errorMsg) {
  // error handler
});
```

Table-level Security Configuration

As mentioned earlier, by default, all tables can be altered by all users.

Data permission filtering introduces the notion of a *locked* table. Only super-users and administrators can create and delete rows in locked tables. Anonymous, unverified, or ordinary users are unable to do so.

A table property is used to specify that a table is *locked.*

Two other table properties control the creation of a row. The first property specifies whether an anonymous or unverified user can create a row in the table (this only applies if a table is not *locked;* it has no effect if the table is *locked*, since row creation is prohibited for all but super-users and administrators). The second property specifies the type of row-level access filter to assign to this newly-created row. Row-level access settings are covered more completely in the *following section*.

These three table properties can be specified in the properties sheet of the XLSX file. If they are not specified, the default values for these three properties are:

partition	aspect	key	type	value
Table	security	locked	boolean	false
Table	security	unverifiedUserCanCreate	boolean	true
Table	security	defaultAccessOnCreation	string	FULL

Row-level Access Filters

Control of who can see, modify, or delete an individual row is governed by the row-level access filter columns of that row and that row's sync status. As described earlier in this page, these filters are stored in the row itself under the _default_access, _row_owner, _group_read_only, _group_modify, and _group_privileged metadata columns. The sync status of the row is also stored in the row itself under the _sync_state metadata column.

Row-level access will always be one of:

- Not visible
- **r** – Read-only access to the row
- **rw** – Read and modify access to the row. Deletion is not allowed. Modification of the row-level access filter columns is not allowed.
- **rwd** – Read, modify and delete access to the row. Modification of the row-level access filter columns is not allowed.
- **rwdp** – Read, modify and delete access, plus the ability to modify the row-level access filter columns.

The rules for the row-level access filter are as follows (stop at the first rule that applies):

1. Super-users and administrators have full read/write/delete(rwd) capabilities on all rows, regardless of their row-level access filters and independent of the table's *locked* status. These privileged users also have the ability to change the row-level access filter column values (ordinary users cannot).

User Capability	unlocked table	*locked* table
ROLE_SUPER_USER_TABLE	rwdp	rwdp
ROLE_ADMINISTER_TABLE	rwdp	rwdp

2. If a row has not yet been synced to the server, the current user has full read/write/delete (rwd) capabilities on that row. This includes the anonymous and unverified users and is independent of the table's *locked* status.

_sync_state	unlocked table	*locked* table
new_row	rwd	rwd

3. If the _row_owner column contain the user_id of the current user, then this user has full read/write/delete (rwd) capability on this row or, for *locked* tables, can modify the row (but cannot delete it).

_row_owner	unlocked table	*locked* table
user_id of current verified user	rwd	rw

4. If the user is a member of one the following groups, their corresponding privileges are shown below.

group columns	unlocked table	*locked* table
_group_privileged	rwdp	rwdp
_group_modify	rw	r
_group_read_only	r	r

5. Otherwise, row-level access is governed by the _default_access column and whether or not the table is locked, as follows:

_default_access	unlocked table	*locked* table
FULL	rwd	r
MODIFY	rw	r
READ_ONLY	r	r
HIDDEN	not visible	not visible

Note: _row_owner can be null or any arbitrary placeholder string. If you use placeholder strings, it is recommended that they not begin with *username:* or *mailto:* or be *anonymous* to prevent any possible collisions with existing usernames. Placeholder strings might be useful in workflows to designate queues of unassigned-work.

Super-users and administrators can update the row-level access filters via the JavaScript API:

```
odkData.changeAccessFilterOfRow(tableId, defaultAccess, rowOwner, groupReadOnly,
  groupModify, groupPrivileged, rowId,
  function(result) {
    // success outcome
    // result holds the result set: SELECT * FROM tableId WHERE _id = "rowId"
  },
  function(error) {
```

```
  // error handler
});
```

Alternatively, super-users and administrators can also use the `updateRow` API.

Ordinary users will receive a not-authorized error if they attempt to set any of these metadata fields (even if the values they set are unchanged from the current values of those fields).

Implementation of the HIDDEN filter on queries

When a SQL query is processed inside the ODK Services layer, it is first examined to see if the result set contains the columns _sync_state, _default_access, _row_owner, _group_read_only, _group_modify, and _group_privileged. If it contains all six columns, then the query is wrapped with a `where` clause to exclude hidden rows and that, in turn, is wrapped by whatever `limit` and `offset` you have specified for the query.

Warning: If you issue a query that omits one or more of these six columns from the result set, then no HIDDEN filtering will be applied. This is one way to circumvent data permission filtering in software – by crafting queries that omit one or more of these fields.

For example, queries that return the maximum value in a field:

```
SELECT MAX(crop_height) as max_height FROM crop_plantings
```

Would return the maximum crop height across all crop planting – even if the current user only had access to the crop height data for their own plantings (and the crop information from other farms was hidden from them).

If you want to restrict such calculations to just the data visible to the current user, you must manually construct the query to do so. This would be the revised query:

```
SELECT MAX(crop_height) as max_height FROM crop_plantings WHERE _default_
↪access != ? or _row_owner = ? bind parameters = [ "HIDDEN", odkCommon.
↪getActiveUser() ]
```

Effective Access

As mentioned above, when a SQL query is processed inside the ODK Services layer, it is first examined to see if the result set contains the columns _sync_state, _default_access, _row_owner, _group_read_only, _group_modify, and _group_privileged. If it contains all six columns, then a synthesized column, _effective_access is added to the result set. That column returns one of *r*, *rw*, *rwd*, or *rwdp* (with the *p* indicating that a user can change permissions for the row as well) to indicate the level of access the current user has on the rows in the result set.

Additionally, once a result set is returned for a given table, you can determine whether the current user can create new rows on the table by calling `getCanCreateRow`

```
odkData.query(tableId, whereClause, sqlBindParams, groupBy, having,
              orderByElementKey, orderByDirection, limit, offset, includeKVS,
function(result) {
  // success outcome
  // result holds the result set. Assume this has at least one row.
  // obtain the effective access for the first row in the result set
  // this will be one of "r", "rw", "rwd", or "rwdp"
  var effectiveAccess = result.getData(0, "_effective_access");
  // obtain the boolean indicating whether the current user can
  // create new rows in this tableId.
  var ableToCreate = result.getCanCreateRow();
},
function(error) {
  // error handler
});
```

Usages Within Applications

Consider a workflow application where a first group of field agents create work requests, those requests are then sent to a supervisor who assigns them to a different set of field agents for processing.

In this case, you might configure a work_requests table to create rows with a HIDDEN default access (via `defaultAccessOnCreation`). Then create a form for opening work requests.

The first group of agents (ordinary users) uses that form to create new work requests. Each agent would only see the work requests they themselves create because all other rows in that table would be hidden due to the _default_access being HIDDEN and due to their being ordinary users.

After the field worker in the first group syncs to the server, and the supervisors sync to the server, the set of work requests the field worker created will have become available on the supervisors' devices. The supervisor (a super-user or administrator) can then see and change the _row_owner on each work request to one of the field agents in the second group.

When the supervisor syncs to the server, and then the field agent in the second group (another ordinary user) syncs to the server, that field agent will see the work items that have been assigned to them (and they will not see any other work items because they are ordinary users of the system).

When the agent in the first group next syncs, their created work item will disappear from their view because it is HIDDEN and the _row_owner no longer matches this field agent's verified user id (it was assigned to the second agent).

Upon completion of the task and after syncing to the server, after the supervisor next syncs, the supervisor could then change the _row_owner to null or to a special placeholder value to remove it from the second agent's list of work items (and that removal would occur when that second agent next syncs with the server after the supervisor syncs his _row_owner change).

Example Application

The app designer has a row-level access demo using the *geoweather* and *geoweather_conditions* tables and forms.

Note: This demo only works on the device.

To install the demo on the device:

1. Force close all the apps.
2. Delete the `/sdcard/opendatakit/default/` directory on the device.
3. From the app designer, execute

```
$ grunt adbpush-tables-rowlevelaccessdemo
```

1. Start ODK Survey and exit it.
2. Start ODK Tables.

You will be presented with a demo launch screen.

At this point, all the rows in all the tables have a _sync_state of new_row and are fully editable and deletable. The demo will not become interesting until you set up and sync with a server.

Set up an ODK Cloud Endpoint or ODK Aggregate 1.4.15 server with 2 ordinary users, 1 super-user and 1 tables administrator. *Reset App Server* to push the configuration and data up to the server.

You are now an administrator (you needed to be in order to reset the server). You can choose *Change Row-Level Access Filters* to view and perhaps modify the default access and row owner of one or more rows. All rows in all tables are fully editable and deletable.

Now, change your *Server Settings* to one of the ordinary users (a username other than *olive* or *sue*). Notice that the list of conditions from the *geoweather_conditions* table no longer contains the *Light Rain* option. That was hidden and will only be visible to a username of "olive" or a super-user or administrator.

Use the table display on the *Change Row-Level Access Filters* page to examine what the _effective_access for each row is in the various tables and verify that those settings are

enforced.

Change your *Server Settings* to different users to see how their effective accesses change.

4.15.2 Internationalization

- *Defining the Available Locales*
- *Referencing Translations in Survey XLSX Files*
- *Referencing translations in Tables Web Pages*
- *Accessing the List of Locales and Default Locale*
- *XLSXConverter Production of Translations*

Internationalization of web page content is achieved through the API exposed in the *odkCommon* object and the configuration contained in the XLSX files for the framework form and for the forms with formIds that match their tableIds:

```
/opendatakit/{appName}/config
   /assets/framework/forms/framework/framework.xlsx
   /tables/{tableId}/forms/{tableId}/{tableId}.xlsx
```

Within the framework XLSX file, the *framework_translations* sheet defines the translations for all of the Survey form labels and prompts. On this page, the string_token column contains the identifier for a particular label or prompt translation. The text.default column provides the default translation for that label or prompt, and all subsequent text.{langCode} columns provide translations for each of those specific language codes (e.g., {langCode} might be es for Spanish). If the label or prompt supports image or media enhancements, there will also be image.default and image.{langCode} or audio… or video… columns providing differing content for those.

Also within the framework XLSX file, there can be an optional *common_translations* sheet following the same format as the *framework_translations* sheet. This can be used to provide an application-wide set of translations.

Similarly, within the ***tableId***`.xlsx` file, there can be an optional *table_specific_translations* sheet that also follows the same format as the *framework_translations* sheet. This is used to provide a table-specific set of translations.

Defining the Available Locales

The list of *{appName}*-wide locales and the default locale are specified on the framework form's *settings* sheet. Individual Survey forms may define additional translations, but those

will be form-specific and are not available to Tables web pages.

Locales are defined in two steps.

First, the survey row under the setting_name column on the *settings* sheet should have an explicit translation for each locale. Do this by creating columns labeled display.title.text.{langCode} for each locale *{langCode}*. It is recommended that you use the Android 2-letter language codes. These are the 2-letter ISO 639-1 language codes, with the exception that iw is used for Hebrew, ji is used for Yiddish, and in is used for Indonesian. Using these 2-letter codes will enable use of the Android system locale for selection of the display language if the *{apppName}* is so configured.

Second, for each of these *{langCode}* values, create a row on the *settings* sheet with that *{langCode}* value under the setting_name column. Then create columns labeled display.locale.text.{langCode} across the top of the *settings* sheet. Provide translations for this language choice or, alternatively, define those translations on the *common_translations* sheet and reference the corresponding *string_token* under a single display.locale column.

The default locale will be the top-most *{langCode}* locale that you specify on the *settings* sheet.

Referencing Translations in Survey XLSX Files

To reference these translations within a question prompt in a survey, instead of specifying a value under a display.prompt.text column, you would create a display.prompt column and place the *string_token* from the translations sheet into that column, leaving any display.prompt…. columns empty. The same applies for hint and title text.

And, finally, in all surveys, you can always provide on-the-spot translations for a prompt label by creating another column display.prompt.text.{langCode} and specifying the translation for that language code directly on the survey sheet.

Referencing translations in Tables Web Pages

In order to access translations, your web page must load the `commonDefinitions.js` file and the `tableSpecificDefinitions.js` files. Within Tables web pages, you can then obtain the appropriate translation via:

```
var locale = odkCommon.getPreferredLocale();
// obtain the text translation for the 'my_string_token' token.
var translatedString = odkCommon.localizeText(locale, 'my_string_token');
```

Additional methods are available within *odkCommon* to test whether a translation exists, and to obtain a localized image, audio or video URL. Refer to that file for the available methods.

Accessing the List of Locales and Default Locale

And finally, to access the list of locales, you can directly access that via:

```
window.odkCommonDefinitions._locales.value
```

This is an array of objects (no particular order). Each object has a display.locale entry that can be translated to the current display language, and a name which is the *{langCode}* for that locale.

And the default locale is available at:

```
window.odkCommonDefinitions._default_locale.value
```

XLSXConverter Production of Translations

After defining your translations on the framework and tableId XLSX files, the XLSXConverter must be run on these files to generate the translation files.

When the XLSXConverter processes the `framework.xlsx` file and emits two files (in addition to the `formDef.json`):

```
/opendatakit/{appName}/config
   /assets/commonDefinitions.js
   /assets/framework/frameworkDefinitions.js
```

Of these, the `frameworkDefinitions.js` just contains a representation for the content of the *framework_translations* sheet.

The `commonDefinitions.js` contains the content of the *common_translations* sheet and the list of locales and the default locale from the *settings* sheet (as described in the previous section)

When the XLSXConverter processes the ***tableId***`.xlsx` file, it emits three files (in addition to the `formDef.json`):

```
/opendatakit/{appName}/config
   /tables/{tableId}/definition.csv -- data definition
   /tables/{tableId}/properties.csv -- table properties
   /tables/{tableId}/tableSpecificDefinitions.js
```

The last of these, `tableSpecificDefinitions.js`, holds a representation for the content of the *table_specific_translations* sheet.

4.15.3 Application Configuration File Structure

A complementary description is provided in the user-level documentation. The tool suite stores its configuration and data files on the SDCard in a directory structure rooted at *opendatakit*. The directories underneath this are *application names* and provide isolation of data and configuration from one user-defined application to the next. By default, if not specified, the *application name* is assumed to be *default*. Underneath this are 4 top-level directories:

```
/opendatakit/{appName}
    /config
        /assets -- common shared configuration
            app.properties -- application properties
            index.html -- HTML home page for Tables (if configured)
            /css -- common css files
            /fonts -- common font files
            /img -- common image files
            /js -- common javascript files written by you
            /libs -- common 3rd party javascript libraries
            ... additional directories and files
            ...
            tables.init -- identifies what to process during initialization
            /csv/{tableId}[.{qualifier}].csv
            /csv/{tableId}/instances/{cleanrowId}/{row-level-attachment-files}
            ...
            /commonDefinitions.js -- shared translations (available in all␣
↪webkits)
            /framework/frameworkDefinitions.js -- Survey template translations
            /framework/forms/framework/framework.xlsx -- XLSX framework form
            /framework/forms/framework/formDef.json   -- created from XLSX

        /tables
            /{tableId}/definition.csv  -- defines the data model of the table
            /{tableId}/properties.csv  -- properties of the tableId
            /{tableId}/forms/{formId}/{formId}.xlsx   -- XLSX form
            /{tableId}/forms/{formId}/formDef.json    -- created from XLSX
            ... optional additional form definition files
            /{tableId}/forms/{formId}/customTheme.css
            /{tableId}/forms/{formId}/customStyles.css
            /{tableId}/forms/{formId}/customPromptTypes.js
            /{tableId}/forms/{formId}/customScreeenTypes.js
            /{tableId}/forms/{formId}/{other-files}
            ... end optional additional form definition files
            /{tableId}/html/{optional-html-files}.html
            /{tableId}/js/{optional-js-files}.js
            ... any other directories and files you might want
            ...
```

```
/data       - database and file attachments for this application
/system     - internally managed javascript files and configuration
    /js/odkData.js
    /js/odkCommon.js
    /tables/js/odkTables.js -- wrapper for odkTables functionality
    /survey/js/odkSurvey.js -- wrapper for odkSurvey functionality
    ... the remaining files are not directly accessed
    /lib/...  -- 3rd party libraries used by Survey
    index.html -- Survey top-level HTML
    /js/mock/... -- mock interfaces used in App Designer
    /survey/js/... -- Survey javascript
    /survey/templates/... -- ODK Survey handlebars templates
/output     - holds logging files, exported data
/permanent  - available for device-only content (e.g., map tiles)
```

4.15.4 ODK Survey Controller Actions

The 10 possible controller actions for Survey's controller are:

- **assign** – a synchronous assignment statement storing a calculation in a field.
- **begin_screen** – push this operation on the navigation history stack and render its screen (this object is both an operation and an instance of a *screen*)
- **goto_label** – jump to an operation identified by the given label. This may be a conditional jump. i.e., the 'condition' property, if present, will be a boolean predicate. If it evaluates to false, then skip to the next question; if it evaluates to true or is not present, then execute the 'goto'
- **do_section** – push this operation onto the section history stack and mark it as 'advanceOnReturn' then start processing operation 0 in the specified section.
- **exit_section** – pop the section history stack (removing the entire navigation history for this section) and if the last operation from the calling section is marked 'advanceOnReturn' then advance to the next operation otherwise execute the operation (unused). This undoes *do_section*.
- **back_in_history** – pop the navigation history stack (unwind to the previous rendered screen) or, if this would exit the current section, display the contents screen for that section, or, if there is no previous screen, navigate to the screen specified when the form was initially loaded.
- **advance** – either execute **back_in_history** if the current operation is marked 'popHistoryOnExit' or advance to the next operation.
- **validate** – if this operation (*validate*) is not yet on the section history stack, push the operation and mark it with 'validateInProgress' (this enables the controller to return to

the validate operation and resume the validation check after the user corrects the first invalid field value). Then traverse all fields with the indicated *sweep_name* and verify their compliance. Upon finding a field that fails validation, retrieve the *begin_screen* operation that contains that prompt, set that operation as the new current operation and mark it as 'popHistoryOnExit'. If all fields validate successfully, pop the section history stack (removing the 'validateInProgress' entry) and execute **advance**.

- **resume** – pop the history stack (unwind to the previous rendered screen) and render that screen. If popping the history stack would have exited the section, exit the section and advance to the next screen after the 'do_section' command.
- **save_and_terminate** – save all changes and close the WebKit.

4.16 Platform Developer Advanced Topics

This section covers advanced topics useful to Platform Developers. A Platform Developer is a programmer that intends to modify the source code of the ODK 2 tools themselves. This person might want to add a new view type or a fix a bug.

Other perspective definitions can be found *here*.

4.16.1 ODK Survey Form Processing

- *Survey Calling Contexts (ctxt)*
- *Survey JavaScript Modules*
- *Survey Control Flow Overview*
  - *`index.html` Initialization Sequence*
  - *Main*
  - *Parsequery*
  - *Builder*
  - *Database*
    - *Retrieving Information About a Database Table*
    - *Creating and Deleting a Database Row*
    - *Getting and Modifying Fields In a Database Row*
    - *Utility Functions for Parsing Selection and Order-By Clauses*

- *Controller*
- *ScreenManager*
- *Screen*
- *Prompt*
- *Survey Controller Actions*

The XLSXConverter converts the XLSX file defining an Survey form into a JSON representation of that form. This representation is then layered on top of the generic Survey JavaScript framework to produce the JavaScript code that is executed when filling out the form.

The primary building blocks of this generic Survey JavaScript framework are:

- bootstrap - for prompt UI and behavior
- Handlebars - for HTML content rendering
- Backbone - for event handling within prompts
- requirejs - for module dependencies and loading
- jQuery - generic utility functions
- underscore - generic utility functions
- moment - date and time handling support

Some additional libraries are use for specific widgets and capabilities (for example, d3 for graphing, combodate for calendar widgets).

The Survey JavaScript framework then adds form navigation, data validation, data storage and data retrieval functions. Central to this framework is the **calling context** which provides a continuation abstraction for chaining and resuming processing during asynchronous interactions.

- *Survey Calling Contexts (ctxt)*
- *Survey JavaScript Modules*
- *Survey Control Flow Overview*
  - *`index.html` Initialization Sequence*
  - *Main*
  - *Parsequery*
  - *Builder*

- *Database*
  - *Retrieving Information About a Database Table*
  - *Creating and Deleting a Database Row*
  - *Getting and Modifying Fields In a Database Row*
  - *Utility Functions for Parsing Selection and Order-By Clauses*
- *Controller*
- *ScreenManager*
- *Screen*
- *Prompt*
- *Survey Controller Actions*

Survey Calling Contexts (ctxt)

The success and failure callbacks used within the *odkData* API are also used throughout the Survey JavaScript. These are so common, that they are passed into functions as a single "calling context" argument, generally named *ctxt*. Whereas many libraries have success and failure callbacks:

- `obj.action(successCallbackFn, failureCallbackFn);`

the Survey JavaScript would just pass in the ctxt object:

- `obj.action(ctxt);`

This *ctxt* object consists, at a minimum, of a `success` function and a `failure` function. The `failure` function generally takes one argument which is an object containing a *message* field that holds an error message. The `success` function may pass in an argument or not.

These calling contexts are created, tracked and managed by the *controller* class via:

- `window.controller.newContext( event )` – when needed during event processing
- `window.controller.newCallbackContext()` – on callbacks from Java shim
- `window.controller.newStartContext()` – special case
- `window.controller.newFatalContext()` – special case

The ctxt object extends the baseContext defined within controller, which has:

```
{
    contextChain: [],
    append: function( method, detail ) {...},
    success: function() {...},
```

```
    failure: function(msg) {...},
}
```

A well-written `success()` or `failure(msg)` function will perform its actions then call the success or failure function of the parent instance from which it is extended. So you will often see code like this in Survey JavaScript:

```
var that = this;
this.render($.extend({}, ctxt, { success: function() {
        that.postRender(ctxt);
    }, failure: function(msg) {
        ctxt.append("mymethod", "unable to render");
        ctxt.failure(msg);
} });
```

Where `postRender(ctxt)` will be responsible for calling the success or failure methods of the ctxt object that was extended and passed into the render() method. The failure(msg) code, in contrast, just logs a message to the context log (via append(), discussed below), and calls the parent instance's failure function.

By always calling the parent instance's success or failure function, you can do interesting things, like implement mutexes (an advanced software construct) – because you are always assured that if you extend a ctxt, that one of your failure(msg) and success() functions will always be called.

The failure(msg) function takes an argument, which is an object that may contain an optional 'message' parameter, which could be a description of what the failure was. This is used during validation.

The use of the ctxt object enables you to store values within the ctxt, and ensure that these are available later in your code, or, via extending it, to change the success function so that it takes an argument, etc., as needed by your code (the database layer quite frequently needs to pass values into the ctxt success method).

The append() function on the context enables you to append a log record to the context. The baseContext's success() and failure(msg) methods both cause the accumulated log messages to be written via the odkCommon.log(). On Chrome, the log message is suppressed. On Android, it is written to the `/opendatakit/`*`appName`*`/output/logging` directory and emitted in the system log if an error or warning.

The 'seq:' and 'seqAtEnd:' values emitted in these logs are useful for understanding what events are processed concurrently within the JavaScript. 'seq' is the sequence number of this context, and 'seqAtEnd' is the sequence number of the newest context in-process at the time this context completes.

Note that when interacting with other asynchronous frameworks, it is easy to convert from ctxt-based style to the success/failure function style:

```
fwk.action( function() { ctxt.success(); }, function() { ctxt.failure(); } );
```

Finally, these calling contexts are very similar to JavaScript promises. However, within the Survey JavaScript, the typical construction is to insert processing steps before taking the success or failure action of the incoming calling context. In contrast, with promises, the typical construction is to append processing steps upon completion of the promise.

In the rare cases when it is necessary to append actions after a calling context chain completes (like the Promise model), two APIs are provided:

- `ctxt.setChainedContext(aCtxt);`
- `ctxt.setTerminalContext(aCtxt);`

Chained contexts are executed in-order, depth-first, from first registered to last registered, after which all terminal contexts are executed in the order in which they were collected from within all of the executed chained contexts. In practice, the Survey JavaScript framework only makes use of terminal contexts, and those usages only register a single terminal context.

Survey JavaScript Modules

All user forms processed within Survey load the same HTML file. Form-specific content and behaviors are specified via the *window.location.hash* portion of the URL. The common HTML file is here:

```
/opendatakit/{appName}/system/index.html
```

and its contents are:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.
↪org/TR/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>OpenDataKit Common Javascript Framework</title>
    <link rel="stylesheet" type="text/css" id="custom-styles" />
    <link rel="stylesheet" type="text/css" id="theme" href="libs/bootstrap-3.3.7-
↪    dist/css/bootstrap.min.css" />
    <link rel="stylesheet" type="text/css" href="../config/assets/css/odk-survey.
↪css" />
    <link rel="stylesheet" type="text/css" id="theme" href="libs/spinner/waitMe.
↪css" />
    <script type="text/javascript" src="../config/assets/framework/
↪frameworkDefinitions.js"></script>
    <script type="text/javascript" src="../config/assets/commonDefinitions.js"></
↪script>
```

```
    <script type="text/javascript" src="js/odkCommon.js"></script>
    <script type="text/javascript" src="js/odkData.js"></script>
    <script type="text/javascript" src="tables/js/odkTables.js"></script>
    <script type="text/javascript" src="survey/js/odkSurvey.js"></script>
    <script type="text/javascript" src="survey/js/odkSurveyStateManagement.js"></
↪script>
    <noscript>This page requires javascript and a Chrome or WebKit browser</
↪noscript>
</head>
<body>
    <div id="block-ui"></div>
    <div class="odk-page">
        <div class="odk-screen">
            <div class="odk-toolbar"></div>
            <div class="odk-scroll">
                <div class="odk-container">Please wait...</div>
            </div>
            <div class="odk-footer"></div>
        </div>
    </div>
    <script type="text/javascript" data-main="survey/js/main" src="libs/require.
↪2.3.3.js"></script>
</body>
</html>
```

This loads a `/config/assets/css/odk-survey.css` file that users can customize, loads the common JavaScript wrapper objects and translation files, and finally triggers *requirejs* to load the framework and (eventually) process the *window.location.hash* to load and interpret the form definition.

The *requirejs* module management framework, under the direction of the `/system/survey/js/main.js` configuration and initialization file, loads the JavaScript files used by the Survey form framework.

Listed alphabetically, these are:

- **builder** - responsible for reading the formDef.json and initializing the controller with the list of prompts in the survey.
- **controller** - handles the logic for moving from one prompt to the next; this includes pre- and post- actions and performing the validation logic.
- **database** - Handles the interactions with the *odkData* interface to the database. This also constructs and maintains the in-memory model description holding the form definition and the instance's data and of the structure of the table in which it is stored.
- **databaseUtils** - contains utility functions for transforming between the database storage strings and the JavaScript reconstructions in the model.

- **formulaFunctions** - common functions accessible from the user's JavaScript eval environment (for use within their formulas).
- **handlebarsHelpers** - Handlebars helper functions for use within handlebars templates. These are invoked via `{{helperFunction arg1}}` or `{{helperFunction arg1 arg2}}` within the handlebars templates.
- **main** - the *requirejs* configuration and initialization file loaded via `index.html` that guides the JavaScript loading process. It waits for various components to load, cleans up the WebKit URL, and invokes `parsequery.changeUrlHash(ctxt)`.
- **odkSurvey** - simple wrapper for invoking the various media capture actions exposed by Survey
- **odkSurveyStateManagement** - this is used only within App Designer to simulate the injected Java interface of the same name.
- **opendatakit** - a random collection of methods that don't quite belong anywhere. Some of these cache and wrap requests to the *odkCommon* layer.
- **parsequery** - responsible for parsing the hash fragment and triggering the building of the form, the triggering the initialization of the data table, changing of the viewed page, etc.
- **prompts** - the core set of prompts defined by the Survey JavaScript framework. The first of these, *base*, defines the basic operation of a prompt.
- **promptTypes** - due to the way *requirejs* works, this defines an empty object into which the prompts (above) are inserted.
- **screenManager** - handles the rendering of a screen, including any please-wait or other in-progress notifications, and the events that initiate actions on that screen (for example, change language, swipe left/right, back/forward button clicks). Many of those actions invoke methods on the *controller* to complete. Note that rendering of the prompts within a screen (equivalent to an ODK Collect field-list) are handled within the definition of the screen.
- **screens** - the core set of screen renderers defined by the Survey JavaScript framework. This includes the templating screen for customized layouts and the standard screen renderer.
- **screenTypes** - due to the way *requirejs* works, this defines an empty object into which the screens (above) are inserted.

Survey Control Flow Overview

`index.html` Initialization Sequence

The `index.html` file explicitly loads these script files:

- **frameworkDefinitions.js** - translations for standard Survey buttons and prompts
- **commonDefinitions.js** - application-wide translations defined by the user
- **odkCommon.js** - wrapper object for *odkCommonIf* injected Java interface
- **odkData.js** - wrapper object for *odkDataIf* injected Java interface
- **odkTables.js** - wrapper object for *odkTablesIf* injected Java interface and convenience methods for Tables navigation actions.
- **odkSurvey.js** - wrapper object providing convenience methods for media capture interactions.
- **odkSurveyStateManagement.js** - mock object used only within App Designer to provide functionality equivalent to the injected Java interface by the same name.
- **require.js** - the requirejs module management library
- **main.js** - loaded indirectly by requirejs to begin the module-load process

The relatively rapid loading of `index.html` very quickly presents 'Please wait…' to the user. This is not internationalized. Once the Survey framework is initialized, this will change to an internationalized prompt (using the *waiting_text* translations), and then be replaced by the requested screen in the form (or first screen of the form) when the form definition is fully processed.

Main

The `main.js` file declares the interdependencies among the various JavaScript frameworks. It relies on *requirejs* for package dependency management and loading. The code first loads jQuery and an extended regex library (for Unicode strings). Once those are loaded, it then loads additional 3rd party libraries and the main Survey JavaScript framework files via:

```
require([ 'spinner', 'databaseUtils', 'opendatakit', 'database', 'parsequery',
                        'builder', 'controller', 'd3', 'jqueryCsv', 'combodate'],
  function(...) {...})
```

Once the ODK frameworks has loaded, the body of the function is executed. The body then initializes the parsequery object (needed to avoid circular references):

```
parsequery.initialize(controller,builder);
```

And then either triggers a reload to clean up the *window.location* value or initiates the parsing of the `formDef.json` specified in the URL location.hash via:

```
parsequery.changeUrlHash(ctxt);
```

Parsequery

parsequery has two main entry points. The first:

```
parsequery.changeUrlHash(ctxt) {
    parsequery._parseParameters(wrappedCtxt);
    // when complete:
    that.controller.registerQueuedActionAvailableListener(ctxt, opendatakit.
↪getRefId());
```

parses the formDef and calls the controller to initiate the processing of data callbacks from the Java layer.

The second entry point is *_prepAndSwitchUI*, which is called deep within the processing performed inside `changeUrlHash(ctxt)` and also by the *controller* when opening a specific *instanceId* within a form. That entry point assumes that the tableId and formId have not changed from what they currently are.

`parsequery._parseParameters(ctxt)` has the following flow (accomplished with many asynchronous processing steps – arguments are omitted):

```
parsequery._parseParameters() {
    if ( !sameForm ) {
        controller.reset( function() {
            // webpage now displays "Please wait..." with translations
            parseQuery._parseFormDefFile();
        });
    } else {
        parseQuery._parseQueryParameterContinuation();
    }
}

// called to load the (new) formDef.json
parseQuery._parseFormDefFile() {
    requirejs( "formDef.json", function() {
        parseQuery._parseQueryParameterContinuation();
    })
}

// called to interpret hash parameters after formDef.json loaded
// If the tableId is changed, load information about the tableId
// from the database layer so we know what fields are in it.
// Otherwise, interpret the formDef.json and construct the
// javascript objects that are used to render that form.
```

```
// And, once the object tree is initialized, call
// _prepAndSwitchUI() to render the specified screen in that form.
parseQuery._parseQueryParameterContinuation() {
    if ( !sameTable ) {
        controller.reset( function() {
            // webpage now displays 'Please wait...' with translations
            // Load information about the tableId from the database
            // layer so we know what fields are in it.
            database.initializeTables(function() {
                // parse and construct form objects
                builder.buildSurvey( function() {
                    // render the specified screen in this form
                    parseQuery._prepAndSwitchUI();
                });
            });
        });
    } else if ( !sameForm ) {
        controller.reset( function() {
            // webpage now displays 'Please wait...' with translations
            // parse and construct form objects
            builder.buildSurvey( function() {
                // render the specified screen in this form
                parseQuery._prepAndSwitchUI();
            });
        });
    } else if ( !sameInstance ) {
        controller.reset( function() {
            // webpage now displays 'Please wait...' with translations
            // render the specified screen in this form
            parseQuery._prepAndSwitchUI();
        });
    } else {
        // render the specified screen in this form
        parseQuery._prepAndSwitchUI();
    }
}

// retrieve and cache information for the instanceId (row)
// being manipulated (if any) and render the specified screen
// in the current form
parseQuery._prepAndSwitchUI() {
    database.initializeInstance( function() {
        controller.startAtScreenPath(ctxt, screenPath);
    });
}
```

From this flow, you can see that the rough sequence of flow is:

1. `controller.reset()` is called to display 'Please wait…'
2. `database.initializeTables()` to retrieve metadata about the tableId.
3. `builder.buildSurvey()` to process the raw formDef.json file.
4. `database.initializeInstance()` creates the initial (largely empty) row of an instanceId (if it is new) and reads the data for the instanceId from the database (if it is pre-existing), sets the current instance id and populates the mdl with the values for that instance id.
5. `controller.startAtScreenPath()` is called to direct the Survey JavaScript framework to display the requested screen.
6. `controller.registerQueuedActionAvailableListener()` is called to initiate the processing of any Java data callbacks (for instance, responses from intents).

Builder

Builder's only entry point is *buildSurvey*. This attempts to load several well-known files and then processes the `formDef.json`.

It begins by attempting to load (in order):

```
/opendatakit/{appName}
     /config/tables/{tableId}/tableSpecificDefinitions.js
     /config/tables/{tableId}/forms/{formId}/customScreenTypes.js
     /config/tables/{tableId}/forms/{formId}/customPromptTypes.js
```

The file `tableSpecificDefinitions.js` contains the translations described earlier.

The `customScreenTypes.js` file contains user-defined screen types. These should follow the constructions of the basic screens defined in `/system/survey/js/screens.js` and should be stored as property fields inside the *screenTypes* object.

The `customPromptTypes.js` file contains user-defined prompt types. These should follow the constructions of the basic prompts defined in `/system/survey/js/prompts .js` and should be stored as property fields inside the *promptTypes* object.

The *column_types* field in the *specification* object within the `formDef.json` is a map consisting of column names and their expected column types. This is used to convert ordinary text describing a calculation into JavaScript functions that perform the calculation (via *eval*). For simplicity, these column names are interpreted independent of the sheet within the XLSX file from which the `formDef.json` is constructed. The allowed values for column types is only partially extensible as it must be interpreted and processed within the builder. The valid column types are:

- function
- formula

- formula(arg1[, arg2[,...]])
- requirejs_path

Columns with the *function* type are expected to contain column values ({columnValue}) that are a text string that can be evaluated as a function definition – for example, {columnValue} would be something like: `function() { return 3; }`.

The *formula* type and the `formula(...)` type are expected to have {columnValue} be an expression that is the return value of a function. These are wrapped by the builder to construct either

```
function() { return ({columnValue}); }
```

or

```
function(arg1[, arg2[,...]) { return ({columnValue}); }
```

Function and formula column types have their content evaluated in the context of the methods exposed by *formulaFunctions* to produce JavaScript functions. Because they are evaluated within the *formulaFunctions* context, they only have limited access to the internals of the Survey framework. This intentionally limits their power and the potential for damage that they might otherwise wreak.

The *requirejs_path* type causes builder to prefix the path to the form's directory. This supports referencing custom prompt templates and, potentially, images and other media, that are stored in the form directory.

The default *column_types* map can be extended in the XLSX file by defining a *column_types* sheet with headings that are column names and a single row beneath that defines the column type for that column name.

The default *column_types* map consists of:

```
{
    _screen_block: 'function',
    condition: 'formula',
    constraint: 'formula',
    required: 'formula',
    calculation: 'formula', // 'assign' prompt and on calculates sheet.
    newRowInitialElementKeyToValueMap: 'formula',
    openRowInitialElementKeyToValueMap: 'formula',
    selectionArgs: 'formula',
    url: 'formula', // external_link prompt
    uri: 'formula', // queries
    callback: 'formula(context)', // queries
    choice_filter: 'formula(choice_item)', // expects "choice_item" context arg.
    templatePath: 'requirejs_path'
}
```

Builder uses the *column_types* field in the *specification* object within the `formDef.json` to convert fields (column names) into their appropriate types. This conversion consists of a a full traversal of content from the calculates, settings, choices, queries, and all the survey sheets in the original XLSX file.

Next, for each of the survey sheets, builder creates Backbone instances of the prompt types referenced on those sheets, one instance for each declared prompt. These instances fold the field definitions the user specified in the XLSX file on top of the default values provided by the prompt definitions (and custom prompt definitions), allowing the user to customize the prompt through explicit changes in the XLSX file. These prompt instances are used when rendering the survey.

Lastly, the builder attempts to load:

```
/opendatakit/{appName}
     /config/tables/{tableId}/forms/{formId}/customStyles.css
```

It then attempts to load:

```
/opendatakit/{appName}
     /config/tables/{tableId}/forms/{formId}/customTheme.css
```

Or, if that doesn't exist, it examines the formDef.json to see if there was a *theme* defined on the *settings* sheet of the XLSX file and attempts to load:

```
/opendatakit/{appName}
     /config/assets/css/{theme}.css
```

And, lastly, it examines the formDef.json to see if there was a *font-size* defined on the *settings* sheet of the XLSX file and attempts to set it in the body:

```
$('body').css("font-size", fontSize.value);
```

Database

The Survey database layer is a fairly thin wrapper around the *odkData* object. It maintains a cache of all of the field values in the referenced instanceId (row) within the current form. This cache is synchronously referenced and modified within the presentation layer and asynchronously updated via calls to the *odkData* object. In general, these asynchronous writes occur during lose-focus event processing.

Additionally, it maintains a copy of the properties of that table (for example, display name of the table and display names of the fields) and a description of the field types in the database table (the table definition). These are returned via the *odkData* object. This information is used within Survey to enable formulas to refer to field values either via their *elementPath* or via the database column in which they are stored (*elementKey*). A prime

example of this is a *geopoint*. If the name of the *geopoint* field is *mylocation* then the individual *latitude*, *longitude*, etc. values are maintained within the cache as individual keys within a *mylocation* object – you can refer to them naturally as *mylocation.latitude*, *mylocation.longitude*, etc. This is the *elementPath* representation of these fields. However, within the database layer, these are stored as individual columns with column names of *mylocation_latitude*, *mylocation_longitude* etc. That is the *elementKey* representation. A similar transformation occurs for file attachments and any user-defined complex data type (multi-valued prompts). Simple select-multiple prompts, which manipulate arrays of values, have an *elementPath* representation within the cache as a Javascript array of selected values. Within the database layer, their *elementKey* representation is a JSON serialization of this array (in contrast, select-multiple prompts that reference linked tables would not store their selections in the dominant data table but rely upon filter conditions and storing a (foreign) key in the subordinate table, or in an association table, to establish their linkage).

The support this synchronous cache and this data abstraction, the main entry points for this layer can be divided into 4 sections:

1. *Retrieving Information About a Database Table*
2. *Creating and Deleting a Database Row*
3. *Getting and Modifying Fields In a Database Row*
4. *Utility Functions for Parsing Selection and Order-By Clauses*

Retrieving Information About a Database Table

Two methods:

- `initializeTables(ctxt, formDef, tableId, formPath)`
- `readTableDefinition(ctxt, formDef, tableId, formPath)`

The first is called during the initial loading of the form; the second is used by linked table prompts.

Creating and Deleting a Database Row

Five methods:

- `initializeInstance(ctxt, model, formId, instanceId, sameInstance, keyValueMap)`
- `get_linked_instances(ctxt, dbTableName, selection, selectionArgs, displayElementName, orderBy)`
- `save_all_changes(ctxt, model, formId, instanceId, asComplete)`
- `ignore_all_changes(ctxt, model, formId, instanceId)`

- `delete_checkpoints_and_row(ctxt, model, instanceId)`

The first method, *initializeInstance* is used to initialize the synchronous cache with data values. It takes a boolean, *sameInstance* that is true if this is a reload of values for the current *instanceId* (row). It also takes a map of data changes *keyValueMap* to apply to this instance.

If *sameInstance* is true, this array is ignored.

If *sameInstance* is false and *instanceId* is null (we are not yet editing a row) then any initial values for the form's session variables that are specified in the *keyValueMap* are applied, and any initial values for any of the row's fields are ignored.

If *sameInstance* is false and *instanceId* is not null, the row's values are fetched from the database. If the row does not exist, it is initialized with the default values specified in the form for each of the row's fields, and then those changes are overlaid with the changes specified in the *keyValueMap*. And, finally, any initial values for the form's session variables that are specified within the *keyValueMap* are applied.

The second method, *get_linked_instances* is used by linked table prompts to retrieve rows from other data tables (for example, for linked table prompts).

The remaining methods (*save_all_changes*, *ignore_all_changes* and *delete_checkpoints_and_row*) manage the retention and deletion of the row in the database table.

Getting and Modifying Fields In a Database Row

Five methods:

- `setValueDeferredChange( name, value )`
- `getDataValue(name)`
- `getInstanceMetaDataValue(name)`
- `applyDeferredChanges(ctxt)`
- `setInstanceMetaData(ctxt, name, value)`

The first 3 of these methods are the standard setters and getters of values. In general, the metadata fields of a row are read-only within Survey JavaScript. For this reason, there is no synchronous setter method for these fields.

The last 2 methods, *applyDeferredChanges* and *setInstanceMetaData*, are used internally within the Survey JavaScript framework to flush the changes in the synchronous cache through to the database via calls to *odkData*. Nearly all manipulation of a row's instance metadata is done within the Java layer. The exception is the changing of the current row's locale, which is effected via the call to *setInstanceMetaData*.

Utility Functions for Parsing Selection and Order-By Clauses

Two methods:

- `convertSelectionString(linkedModel, selection)`
- `convertOrderByString(linkedModel, order_by)`

These functions examine where clauses and order-by clauses to replace any *elementPath* expressions with *elementKey* values. Because this is not within the database layer, these conversions are not entirely fool-proof.

Controller

The initial load of a form ends with a call to `controller.startAtScreenPath()` followed by a call to `controller.registerQueuedActionAvailableListener()`.

The *controller* object is responsible for navigating the form, ensuring that required fields are populated, that constraints are applied, that all validation logic is executed, and that appropriate actions are taken when the user launches an external application (for example, media capture), launches a sub-form, saves the form, exits without saving, or elects to delete a row from the database.

To implement *back button* functionality, the controller maintains a history of how the user has navigated through the form. This navigation history is necessary because there is no fixed execution path through an Survey form (user-directed navigation is one of the big changes between the javarosa-based tools and Survey). The *odkSurveyStateManagement* injected Java interface provides the underlying storage mechanism for this functionality and is directly called by *controller* during its processing.

The types of actions that the controller can perform, and how these are defined in the *formDef.json* will be described later in this document. At this time, it is sufficient to know that the controller is executing a program that performs actions, such as the rendering of a screen containing one or more prompts, as well as performing conditional and unconditional branches within that program.

The controller's progress through this program is tracked by the history stack maintained within *odkSurveyStateManagement* and the top of that history stack identifies the operation which the controller is currently executing. The controller's (vastly simplified) form processing flow is as follows:

```
controller.startAtScreenPath(ctxt, screenPath) {
    var op = operation corresponding to screenPath.
    controller._doActionAt(op);
}
//
// starting at the operation referenced by 'op',
```

```
// execute operations until a screen is rendered
controller._doActionAt(op) {
    controller._doActionAtLoop(op);
    // when the above completes, we are
    // given a screenOp (screen rendering
    // operation) to transition to, or
    // have already produced a pop-up to
    // communicate an error to the user.
    if ( screenOp !== null ) {
        controller.setScreenWithMessagePopup(ctxt, screenOp, ...);
    }
}
//
// main execution loop
controller._doActionAtLoop(op) {
    while () {
        switch ( op._token_type ) {
        case "goto_label":
            // jump (possibly conditionally)
            // to another operation
            break;
        ...
        // other control flow options
        // some of these can return out
        // of this while without returning
        // a screen rendering operation.
        // any that do will have already
        // produced an alert or error pop-up
        ...
        case "assign":
            // do assignment
            break;
        case "begin_screen":
            // render a screen
            return op; // the 'screenOp' in _doActionAt();
        }
    }
}
//
// render a screen
controller.setScreenWithMessagePopup(ctxt, screenOp, options, msg) {
    // set up a 500ms delay timer to render the 'msg' pop-up
    // so that the UI can settle on the new page before we
    // display the message. Otherwise, it might be lost
    // during the rendering of the screen.
    setTimeout(function() {
        screenManager.showScreenPopup(m);
```

```
    }, 500);
    screenManager.setScreen(ctxt, screenOp, options);
}
```

Simply put, the processing flow eventually calls *screenManager* to display a screen (via *setScreen(ctxt, screenOp, options)*) and perhaps also shows a pop-up with some sort of alert or error message (via *showScreenPopup(m)*).

When the *next button* is pressed or the screen is swiped forwards, the framework calls `controller.gotoNextScreen()` which verifies that all required fields are filled-in and all constraints are applied. It then triggers much the same processing sequence – calling *doActionAt()* with the operation *after* the currently-rendered screen.

When the *back button* is pressed or the screen is swiped backward, the framework calls `controller.gotoPreviousScreen()` which pops the operation history stack for the current survey sheet until a screen-rendering operation is found, and that screen is then rendered. And, if the history for the current survey sheet is exhausted, then the contents screen for that sheet is displayed.

Finally, returning to the discussion of the control flow on the initial load of a form, after the current screen is rendered, the call to `controller.registerQueuedActionAvailableListener()` causes an action listener to be registered with *odkCommon* and then calls that listener to process any results that became available before the listener was registered. If there are any results from a previous `odkCommon.doAction(...intentArgs...)` request (for example, a media-file capture request), then the controller's action listener will interpret the results to identify what prompt in the current screen should receive and process these results and then invoke that prompt to complete the processing. Otherwise, if there are no results, no additional actions are taken. This completes the control flow on the initial load of the form.

ScreenManager

The screenManager provides event handling for swiping and the navigation bars at the top and bottom of a screen. It delegates to the *screen* object to construct the DOM representation for that content and also delegates to the *screen* object to register and unregister event handlers for any other DOM elements via calls to `recursiveUndelegateEvents()` and `recursiveDelegateEvents()`. Those event handlers are expected to be defined in the Backbone-based *screen* objects and *prompt* objects.

The high-level actions of the screen manager are:

```
screenManager.setScreen(ctxt, screen) {
    // show "loading..." spinner
    screenManager.showSpinnerOverlay();
    // stop processing all events on the current screen
    screenManager.disableSwipeNavigation();
```

```
    screenManager.activeScreen.recursiveUndelegateEvents();
    // construct the DOM objects in the page (heavily nested)
    screen.buildRenderContext(... {
        screen.render(... {
            screenManager.activeScreen = screen;
            // replace the screen
            screenManager.$el.find(".odk-page").replaceWith(screen.$el);
        });
    });
    //
    // and via a ctxt.terminalContext()  registration
    // so that the DOM replacement and redraw can take effect
    screenManager.activeScreen.afterRender();
    screenManager.activeScreen.recursiveDelegateEvents();
    screenManager.hideSpinnerOverlay();
}
```

Screen

The *screen* object determines the set of prompts that should be displayed and lays them out. The custom screen example shows how this can be done within an arbitrary HTML template by using ids on DOM elements to identify where the inner HTML for a prompt should be injected.

Immediately prior to screen rendering, any unsaved changes in data values are asynchronously flushed to the database.

The *screen* object also enforces required fields and constraints and can reject any attempts by the *controller* object to move off of this screen or pop-up a confirmation for the user to accept.

See the `screens.js` file.

Prompt

Prompts register event handlers for their DOM elements and are responsible for restoring and saving values displayed in those DOM elements into the synchronous data cache and for validating those values and enforcing any constraints (if so directed).

See the `prompts.js` file.

Survey Controller Actions

As mentioned earlier, the main processing loop within the *controller* executes a program derived from the form's XLSX file and encoded in the *formDef.json*. The 10 primitive operations in this program are described in *ODK Survey Controller Actions*.

4.16.2 Survey formDef.json Structure

- xlsx *Component*
- specification *Component*
- sections *Sub-Component*
- operations *Sub-Sub-Element*
  - *assign*
  - *begin_screen*
  - *goto_label*
  - *do_section*
  - *exit_section*
  - *back_in_history*
  - *advance*
  - *validate*
  - *resume*
  - *save_and_terminate*

The XSLSXConverter in the AppDesigner reads the XLSX form definition file and produces a set of output files, including `formDef.json`, as described earlier in this document.

In general, users of Survey will not directly interact with this file. Instead, they would write their forms in the XLSX file. Several features of Survey support that simplified usage:

1. custom screens and prompts can be defined in separate JavaScript files (see the earlier discussion of the Builder processing sequence). These custom prompts and screens can then be referenced by name in your XLSX form definition. This allows new widgets to be defined without extending the XLSX syntax or the XLSXConverter.
2. if additional member variables or functions are needed in your prompt logic, you can define these simply by adding a column with that member variable name to the survey sheet. If there is a value in that field, the member variable will be created and present

in the prompt instance when the form is being rendered. Use the *column_types* sheet to define the interpretation of these members (e.g., to interpret them as functions or formulas). Member variables that are objects can be created by using "." to separate the member variable name from the field name, as is done with the *display.** columns, which define a member variable that is an object with various fields (e.g., *display.prompt*, *display.text*, etc.). Array-valued member variables can be created using *[0]*, *[1]*, etc. to specify the values in the array.

3. custom CSS styles and themes can be defined. Additionally, the full power of JavaScript is available when needed.

A primary goal of the `formDef.json` format was to preserve enough of the original source document (XLSX) to enable that document to be roughly reconstructed. The cell locations and values of all the cells in the XLSX file are retained, but formatting and coloring of the cells is not preserved. One can theoretically write an inversion program that would take this information and reconstruct an XLSX file without formatting or cell coloring that would be functionally equivalent to the original source XLSX file. This solves a common issue in the JavaScript-based tools where the conversion process fails to preserve enough information about the source document to enable it to be recreated.

Additionally, while the XLSXConverter performs numerous cross-checks, some errors cannot be reported until the form is executed. When these occur, error messages must identify the sheet, row and column in which the error occurred so that it is easy for form designers to correct the issues.

With this understanding, the structure of the `formDef.json` file consists, at the top level, of an object with 2 fields:

```
{
    xlsx: {...},
    specification: {...}
}
```

xlsx Component

Note that alternative form description environments, such as drag-and-drop form builders, are expected to produce content that might be stored under a different field name. As those other tools develop, it is expected that some error message handling will need to be revised to properly report errors against those other source descriptions.

The purpose of this component is to enable an inversion tool to generate an XLSX file that is functionally equivalent to the source XLSX that generated this `formDef.json`.

Note: Writing custom prompts that directly reference the content of the *xlsx* component is fragile and should be discouraged. Future versions of the Survey JavaScript framework may delete this component from the `formDef.json` structure that is retained during form

execution. Remember that prompts will automatically possess member values and functions that you have defined on the survey sheets, so there is little need to retrieve information by directly access the `formDef.json` structure.

The *xlsx* object has field names that correspond to the names of the sheets in the originating XLSX file. Each sheet in the XLSX file is assumed to have a header row followed by data rows beneath it. The values for these sheet-name fields are arrays of objects, one or each data-row on that sheet. i.e., the header row is omitted. Each of these row objects will contain a *_row_num* field with the corresponding row number in the original XLSX file.

If a cell in the originating XLSX file's data-row was not empty, the corresponding data-row object will have a field with the column name from the header-row and this value as the field-value. For complex header-row column names, like *display.prompt.text*, the resulting data-row object will have a *display* field with an object value with a *prompt* field with an object value with a *text* field with the cell content. In cases where a value for the root cell: *display.prompt.text* and a cell field: *display.prompt.text.en* are both specified, the value in the root cell (*display.prompt.text*) will be pushed down into a *default* field.

Here is a portion of the *xlsx* structure showing the content of the first data row of the survey sheet from the example form:

```
"xlsx": {
    "survey": [
      {
        "type": "integer",
        "name": "default_rating",
        "display": {
          "prompt": "first_prompt",
          "hint": {
            "text": "If the form does not yet have a rating, this will be␣
↪proposed for the rating value. This value is not retained in the survey result␣
↪set and exists only for the duration of this survey session."
          }
        },
        "model": {
          "isSessionVariable": true
        },
        "_row_num": 2
      },
  ...
```

Note: Recreating the XLSX file from this structure is mechanical but the reconstruction cannot preserve the order of the header columns, since that information has already been discarded.

specification Component

The *specification* component of the `formDef.json` object is the only part of that is active used by the Survey JavaScript framework. This component contains the following fields:

- **column_types** – used by builder. Can be extended by adding a *column_types* sheet in the XLSX file.
- **settings** – content from the settings sheet in the XLSX file. This is an object with field names corresponding to the *setting_name* on that sheet with values corresponding to the data-row matching that setting name. Retrieve a given *setting_name* via a call to `opendatakit.getSettingObject(opendatakit.getCurrentFormDef(), setting_name)` There are accessor methods defined in the *opendatakit.js* JavaScript file for retrieving common settings values.
- **choices** – content from the choices sheet in the XLSX file. This is an object with field names corresponding to the *choice_list_name* on that sheet. The values for these fields are arrays of objects, one object per row matching that *choice_list_name* in the order in which they appear in the choices sheet. This information is returned as part of all data row fetches and queries and is accessible on the *odkData* result object via calls to `resultObj.getColumnChoicesList(elementPath)` and, for individual data values, you can access the object corresponding to that data value most efficiently via `resultObj.getColumnChoiceDataValueObject(elementPath, choiceDataValue)`. Within Survey, the prompts use a wrapper function: `opendatakit.getChoicesDefinition(choice_list_name)` to access the choices list. The choices field should eventually be removed as the above calls on the result object are definitive and those choice lists come from the choices sheet of the form whose formId matches the tableId. The choices sheet within each form XLSX file will be retained until the AppDesigner and XLSXConverter can become smarter.
- **queries** – content from the queries sheet in the XLSX file. This is an object with field names corresponding to each *query_name* on that sheet. The values for these fields are objects corresponding to the data-row matching that setting name. Retrieve a given *query_name* via a call to `opendatakit.getQueriesDefinition(query_name)`
- **calculates** – content from the calculates sheet in the XLSX file. This is an object with field names corresponding to each *calculation_name* on that sheet. The values for these fields are objects corresponding to the data-row matching that setting name.
- **section_names** – an array of the survey sections from the XLSX file. This includes the synthesized *initial* sheet if one is not explicitly specified.
- **sections** – an object with field names corresponding to each of the *section_names*. Each such field defines the form content for that section (that sheet in the XLSX file).

After the *builder* has processed the form definition, the following fields are added:

- **currentPromptTypes** – a list of all standard and custom Backbone prompt classes.

- **currenScreenTypes** – a list of all standard and custom Backbone screen classes.

Additionally, the *builder* also scans and alters all of these fields from their original `formDef.json` content by applying the *column_types* field mappings to their content. See the *Builder* section, earlier, for how it replaces or modifies some string value content.

The following fields are present, but _are not the authoritative source for this information and **may be removed in future releases**._ They are present only to support the emulation of the Java environment when running in App Designer and are candidates for removal as that environment evolves:

- **dataTableModel** – the authoritative version of this content is returned in response to a database query on a table. This is used within the App Designer to emulate the Java environment.
- **model** – this content is an intermediate synthesis of the model sheet and all datatype attributions in the survey and survey sections. It is used to generate the `definition.csv` file. And, on the Java side, that file is used to create the database table and construct the *dataTableModel* returned by the *odkData* object.
- **properties** – this content is used to generate the `properties.csv` file and is returned through a database query as metadata by the *odkData* object. It is only used directly when rendering the *framework* form, which is only done within the App Designer. The App Designer also uses it during database initialization.
- **table_specific_definitions** – this content is written to `tableSpecificDefinitions.js`
- **framework_definitions** – this content is written to `/config/assets/framework/frameworkDefinitions.js`
- **common_definitions** – this content is written to `/config/assets/commonDefinitions.js`
- **choices** – as noted above, this should eventually disappear and the choices sheet should eventually only be present in the form whose formId matches the tableId. That can't happen until XLSXConverter and the AppDesigner get smarter (i.e., this will likely persist in the `formDef.json` for longer than any of the above fields).

sections Sub-Component

Each section object in the *sections* sub-component contains a heavily processed and cross-checked version of that section of the survey. These objects have the following fields:

- **section_name** – the section name – i.e., the name of the sheet in the original XLSX file.
- **nested_sections** – a map of all the section names that are targets of *do_section* actions within this section.

- **reachable_sections** – a map that is the closure of all section names that can be recursively reached by all nested sections and by this section. This is used to ensure there are no cycles among the sections.
- **prompts** – a list of all prompts within this section.
- **validation_tag_map** – a map of all validation tag names and the array of prompts that reference that tag name (and that have value constraints). Prompts can specify a list of validation tag names that will enforce the prompt's constraints by specifying a space-separated list of values for a *validation_tags* column in their XLSX sheet. Intermediate validation of some prompt values can be achieved via the *validate {tagName}* action at any point in a survey. If nothing is specified for the *validation_tags* column, the prompt is automatically added to the *finalize* validation tag, which is processed when the *Save as Complete* action is initiated within the form.
- **operations** – an array of operations that the *controller* iterates through to process this section of the form. Unless otherwise specified, processing starts at index zero in this array.
- **branch_label_map** – a map of all branch (go-to) label names and the index within the operation array to which they correspond. Used to map the 'goto label' operation to a destination within the operations array.

After the *builder* has processed the form definition, the following fields are added:

- **parsed_prompts** – a list of Backbone instances corresponding to the extension of the referenced Backbone prompt type with the field values found in the **prompts** list.

And *builder* also scans the *operations* list applying the *column_types* rules.

During form navigation, the *parsed_prompts* list of Backbone instances will be used to render DOM content and handle events. _The *prompts* array may be removed by the *builder* in some future release._ Each of these prompts has an XLSXConverter-generated field, *_branch_label_enclosing_screen* that identifies the branch label for the operation that will render the screen containing this prompt. This is used during validation to map back from the prompt whose constraints are violated to the *begin_screen* operation that will render the screen containing that prompt. Prompts also have *_row_num* and *___rowNum_* fields that reference the XLSX row in the section that defines the prompt and the line number within the section (one less than the XLSX row number due to the presence of the header row), respectively. These are used for reporting exceptions during form loading and processing (i.e., malformed formulas, etc.).

operations Sub-Sub-Element

Each element in the *operations* array describes an action the *controller* should execute when processing the form. The 10 primitive operation types were described in an earlier section. Below are brief examples of these various primitive operations.

Within all operations objects:

1. an *operationIdx* field contains the index into the *operations* array under which this operation object is stored.
2. the *_token_type* field contains the operation type.
3. the *_row_num* field contains the (first) row in the section that corresponds to this action. i.e., if a screen contained multiple prompts, this would be the row containing the *begin screen* action.

Here are specifics for each operation type:

assign

assign actions can appear within *begin screen* … *end screen* regions or outside of them. If they appear outside of them, they are interpreted as a separate operation by the *controller*. Here is an example of such an assign action:

```
{
  "type": "assign",
  "name": "default_rating",
  "calculation": 8,
  "_row_num": 2,
  "__rowNum__": 1,
  "_token_type": "assign",
  "operationIdx": 0
},
```

The key fields in this are:

1. *name* – the field (session variable or a field in the data row) to assign.
2. *calculation* – the expression to evaluate and assign in the field. This is converted by *builder* into a JavaScript function (i.e., transforming it into: `function() { return (8); }` which is then evaluated).

begin_screen

This is an example of a *begin_screen* operation object:

```
{
  "clause": "begin screen",
  "_row_num": 20,
  "__rowNum__": 19,
  "_token_type": "begin_screen",
  "_end_screen_clause": {
```

```
    "clause": "end screen",
    "_row_num": 23,
    "__rowNum__": 22,
    "_token_type": "end_screen"
  },
  "_screen_block": "function() {var activePromptIndicies = [];\nassign('coffee_
↪today', (( data('coffee_today') == null ) ? data('avg_coffee') : data('coffee_
↪today')));\nactivePromptIndicies.push(11);\n\nreturn activePromptIndicies;\n}\n
↪",
  "operationIdx": 22
},
```

In addition to the standard fields, this contains:

- *clause* – the action clause that this corresponds to. If this were generated by a lone prompt, the *clause* field would be missing.
- *__end_screen_clause* – the clause that marks the *end screen* statement.
- *__screen_block* – this field will be processed by *builder* to generate a JavaScript function. It encapsulates any assign operations and any if-then-else logic within the *begin screen … end screen* region that determined which prompts should be shown on that screen. The function returns an array of the prompt indices that should be rendered at this time.

Note that the above example shows how an if-then-else clause and assign action are transformed into a *__screen_block*

Here is another example, this one for a prompt that is not wrapped by a *begin screen … end screen* action:

```
{
  "_row_num": 2,
  "_token_type": "begin_screen",
  "_screen_block": "function() {var activePromptIndicies = [];
↪\nactivePromptIndicies.push(0);\n\nreturn activePromptIndicies;\n}\n",
  "operationIdx": 0
},
```

goto_label

If-then-else clauses outside of *begin screen … end screen* regions are converted into branch labels and conditional and unconditional *goto_label* commands. Additionally, users may explicitly jump to a label using a *goto* clause in the XLSX file (and do that conditionally if they specify a *condition* predicate). Here is an example of a conditional *goto_label* operation object generated from an *if* clause.

```
{
  "clause": "if",
  "condition": "selected(data('examples'), 'intents')",
  "_row_num": 4,
  "__rowNum__": 3,
  "_token_type": "goto_label",
  "_branch_label": "_then4",
  "operationIdx": 2
},
```

The key fields for this are:

- *condition* – present if the goto is conditional. If present, this is converted by *builder* into a JavaScript function.
- *_branch_label* – where the goto should jump to.

Here is another example of an unconditional goto generated as a result of an *end if* clause:

```
{
  "clause": "end if",
  "_token_type": "goto_label",
  "_branch_label": "_else9",
  "_row_num": 9,
  "operationIdx": 3
},
```

do_section

Here is an example of a *do_section* operation:

```
{
  "clause": "do section household",
  "_row_num": 2,
  "__rowNum__": 1,
  "_token_type": "do_section",
  "_do_section_name": "household",
  "operationIdx": 0
},
```

The key field here is:

__do_section_name which identifies the section name that should be jumped into.

exit_section

Here is an example of an *exit_section* operation:

```
{
  "_token_type": "exit_section",
  "clause": "exit section",
  "_row_num": 7,
  "operationIdx": 3
},
```

back_in_history

This is primarily used as a pseudo-instruction (an instruction injected into the operation stream) when the user hits the *Back* button or swipes backward. This is also emitted as a real operation when a *back* clause is specified in the XLSX file. Used in that manner, it can create a "dead-end" screen that the user cannot swipe through (they can only go backward) and can be useful when presenting a user with a *user_branch* prompt (where the user must choose the next action and there is no default action).

```
{
  "clause": "back",
  "_row_num": 4,
  "__rowNum__": 3,
  "_token_type": "back_in_history",
  "operationIdx": 3
},
```

advance

This is only used as a pseudo-instruction (an instruction injected into the operation stream) when the user hits the *Next* button or swipes forward.

validate

This is an example of a *validate* operation:

```
{
  "clause": "validate user_info",
  "_row_num": 12,
  "__rowNum__": 11,
  "_token_type": "validate",
```

```
  "_sweep_name": "user_info",
  "operationIdx": 7
},
```

Partial validation of a form is one of the advanced features of Survey. In this instance, only the fields tagged with the *user_info* validation tag will be verified. The key field for this operation is:

> *_sweep_name* – the name of a validation tag. Any fields that have this name in their space-separated list of validation tags under the *validation_tags* column in the XLSX file will have their constraints validated.

If a field has a constraint but no values under the *validation_tags* column, *finalize* will automatically be assumed to be in that list. 'validate finalize' is called when a form is saved-as-complete.

resume

None of our examples explicitly use this clause in the XLSX file. However, it is used in the construction of the default Contents screen handler for a section which is emitted if the form designer did not specify their own '_contents' branch label and define their own screen for this purpose. Choosing to view the contents screen causes a jump to the '_contents' branch. The default implementation of that branch is a *begin_screen* operation to display the Contents screen followed by a *resume*. The default Contents screen has its *hideInBackHistory* field set to true. This causes that screen to not be saved in the back history. When a user swipes forward, the *resume* operation will scan backward to the screen before the Contents screen (since it is skipped) and will render that screen (returning the user to the screen they were last at).

```
{
  "_token_type": "resume",
  "clause": "resume",
  "_row_num": 9,
  "operationIdx": 12
}
```

save_and_terminate

This is not explicitly used in our examples, but it is used within the automatically-generated 'initial' section if the user has not defined their own. This operation corresponds to a 'save and terminate' clause. That clause takes a 'condition' expression that indicates whether the content should be saved-as-complete or saved-as-incomplete (this clause does not itself determine the validation status and hence completeness of the data). Because of this, any

save-as-complete action should be preceded by a 'validate finalize' clause to ensure that the form is validated. After saving the form contents, the Survey window is then closed.

```
{
   "_token_type": "save_and_terminate",
   "clause": "save and terminate",
   "calculation": true,
   "_row_num": 9,
   "screen": {
     "hideInBackHistory": true
   },
   "operationIdx": 11
 },
```

4.16.3 ODK 2 Sync Protocol

Introduction

This documents the Synchronization API used in ODK 2.

The ODK 2 tools utilize a REST API to exchange configuration and data values with the server.

REST URL formats

This document summarizes the API and the usage of the API. The URLs for the REST API have a common URL prefix. E.g.,

- `https://hostname:port/path/of/prefix/`

That is assumed to be supplied by a configuration setting.

When describing the REST URL, path elements surrounded by curly braces (*{}*) indicate the use of the value for that term in that location within the path. There are a handful of these substitution terms used within the REST URLs. The most common of these are:

- *appId*– identifies the 'application', which is a collection of configuration files and data tables that provide a self-contained user experience. e.g., a survey campaign, a specific set of workflows, etc. Applications live on the Android device under different subdirectories within the `/sdcard/opendatakit` directory. The name of the subdirectory is the *appId* of the application contained in the directory. The default application, with an *appId* of *default* lives under the `/sdcard/opendatakit/default/` directory.
- *odkClientVersion* – the "major version" of ODK 2 software on the device. This is the 100's digit of the Android manifest version code. Also referred to as the "rev number"

of the release. I.e., for rev 206, the *odkClientVersion* would be 2. Non-backward-compatible changes to the JS API would bump this up. It allows groups to maintain and move across incompatible API changes by supporting different versions of the `formDef.json`, HTML and JS configuration files. Until we reach a release candidate, we are not strictly tracking non-backward-compatible client versions. The exception being the transition from jQuery-mobile-based JavaScript (version 1) and the current bootstrap-based JavaScript (version 2).

- *tableId* – identifies a particular data table.
- *schemaETag* – identifies a particular manifestation of a table. If you drop the table and recreate it, the re-creation will have a different *schemaETag* that the original table, even if it is otherwise identical. In contrast, adding, updating or deleting individual rows in a table does not change the *schemaETag* for that table.
- *rowId* – the primary key for a particular row within a table.
- *rowETag* – identifies a particular revision of a row within a table.

When defining the REST API, we use modified version of the JAX-RS annotations to describe the interface. For example, the API to create a table on the server is described as:

```
@PUT
@Path("{appId}/tables/{tableId}")
@Consumes({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*TableResource*/ createTable(TableDefinition definition)
    throws ODKDatastoreException,
           TableAlreadyExistsException,
           PermissionDeniedException,
           ODKTaskLockException;
```

`@PUT`, `@POST`, `@GET` and `@DELETE` indicate the type of HTTP request.

`@Path` indicates the URL path to invoke this method, with the curly brace substitutions of the indicated substitution terms. This is appended to the common URL prefix provided by the configuration setting.

`@Consumes` indicates the mime types of message bodies accepted by the server. In general, the server accepts JSON and XML in UTF-8 format; JSON is preferred.

`@Produces` indicates the mime types of the message bodies returned to the client. In general, the server can return JSON or XML in UTF-8 format; JSON is preferred.

The method may have zero or more arguments qualified by `@QueryParam(...)`. These identify query parameters for the request, with the … indicating the query parameter name.

Methods with entity bodies (PUT and POST methods) will generally have an additional unqualified argument that identifies the content of that entity body. In our documentation, this will generally be a Java class that uses Jackson2 parsers to marshal its content into or out of XML or JSON representations (in the above example, the body of the HTTP PUT request is a *TableDefinition* object).

The return type is indicated in a comment. The *Response* return type is a generic response type that encapsulates both the successful return type (*TableResource* in this example) and the error codes for the various exceptions. As this API gets fleshed out, the error codes for each specific exception will be documented at the bottom of this page.

In general, the server supports GZIP compression of entity bodies in both directions.

Requests should specify 3 or 4 headers:

- **X-OpenDataKit-Version** – this should be set to *2.0*
- **X-OpenDataKit-Installation-Id** – this should be set to a UUID that identifies this client device. This UUID will generally be generated on first install of the ODK Services APK. Using "Clear Data" in the device settings will cause a new UUID to be generated. This is used to track the devices responsible for changes to the configuration (resetting the server) and for tracking the status of all devices as they synchronize with the server.
- **User-Agent** – this is required by Google App Engine infrastructure before it will honor requests for GZIP content compression of response entities (i.e., it ignores "Accept-Encoding" directives on requests if this is not present). The value supplied must end with " (gzip)". Services uses a value of: *"Sync " + versionCode + " (gzip)"* where *versionCode* is is the revision code of the software release (e.g., 210). While optional, it is highly recommended that all requests supply this header.
- **Accept-Encoding** – this should be set to "gzip" when an entity body is returned.

REST Data Structures

We use Jackson 2.0 for transforming Java objects to and from XML and JSON representations. To understand the representations, it is best to use curl or any other REST client to send requests to the server and view the returned structures.

In the following presentation, we provide the Jackson 2.0 annotations used in our code.

Data Groupings

Before discussing the API, it is useful to identify the data on the system. The ODK 2 tools assume all data fall into one of six groupings:

1. (**Data Grouping #1**) HTML, JavaScript and tool configuration files that are not specific to any data table. These include custom home screens, CSS, logo icons, and settings for the tools (e.g., default font size, what settings options to show or hide).
2. (**Data Grouping #2**) Data table definition, properties, HTML and JavaScript associated with a specific data table. These include all ODK Survey forms used to create or edit this data table, ODK Tables HTML and CSS files for list views, map displays and graphical displays of the data, and ODK Scan mark-sense form definitions.
3. (**Data Grouping #3**) Data rows and the file attachments (e.g., images, audio, video or other files) associated with specific revision(s) of each data row.
4. Other files and data that are not synchronized with the server and are for internal use only; e.g., the tools' internal configuration files and device-specific configuration.
5. Other files that are not synchronized with the server but are generated for external use such as exported csv files and detailed log files for troubleshooting.
6. content that is independently downloaded and managed by other means (e.g., cached map tiles). I.e., this is content that is not synchronized with the server via the Synchronization REST API.

Directory Hierarchy and Naming Convention

A directory hierarchy and naming convention partitions files into each of the above 6 groupings. This is described here.

The mapping of these directories to the 3 data groupings that are synchronized with the server through the Synchronization REST API are as follows:

All table-level configuration files (**Data Grouping #2**) are either located under:

- `.../config/tables/`*`tableId/`*

Or, they are files or directories under the *csv* folder:

- `.../config/assets/csv/`*`tableId`*`.csv`
- `.../config/assets/csv/`*`tableId/*`*
- `.../config/assets/csv/`*`tableId.qualifier`*`.csv`
- `.../config/assets/csv/`*`tableId.qualifier/*`*

Note that the file:

- `.../config/tables/`*`tableId`*`/definition.csv`

Defines the schema for the table. This is stored on the server, but is not verified against the schema as created through the create-table REST API. This file is only processed when initializing a device database from content pushed from app-designer.

Note that the file:

- `.../config/tables/`*`tableId`*`/properties.csv`

Defines the key-value-store values for a data table. These define things such as the formId to use to edit the records in the table, the display names of the columns, etc. Prior to syncing a tableId, the contents of the key-value-store are written to this file, and this file is then compared against the file on the server. If there is any difference, the server file is downloaded. After the file is downloaded, the key-value-store entries for this table are entirely removed and replaced with the content from the server. Thus, with each sync, any changes you had made using the table properties-setting pages in Tables will, in general, be destroyed. These can only be preserved if you reset the app server, pushing your local properties.csv file up to the server. Future versions of the system may eliminate the table properties configuration screens from Tables and move them up to the app-designer (where they rightfully belong).

Everything else under `.../config` is **Data Grouping #1**.

Everything under `.../data` is **Data Grouping #3**.

All remaining files are not synchronized and are managed either as internal state of the application or are output produced by the application.

Overall Sync Workflow

The overall sync workflow is:

1. verify that the server supports the device's *appId* If the server does support the device's application name, then stop and report a server-configuration compatibility failure.
2. authenticate the user
3. request the list of capabilities (roles) the user has been assigned.
4. request the list of users on the server.
5. if the device is syncing (vs resetting the app server), verify that the server supports the device's *odkClientVersion* If the server does not have any files for that client version, then stop and report a server-configuration compatibility failure.
6. ensure that the device's set of files and the tools configuration not specific to any table (**Data Grouping #1**) exactly matches that on the server for the device's *odkClientVersion* – removing any files on the device that are not on the server.
7. for each table, ensure that the device's table definition and table-specific configuration (**Data Grouping #2 part A**) exactly matches that on the server and that all the files and configuration specific to that table exactly matches those on the server for the device's *odkClientVersion* – removing any extraneous files on the device.

8. leave any tables that are on the device but not on the server untouched (do not delete them). By removing the configuration files for this table, it becomes invisible to users. for each table on the device that is not on the server, delete that table and its table-specific files ((**Data Grouping #2 part B**). After this step, the table configuration on the device exactly matches that of the server.
9. for each table, perform a bi-directional sync of the data and file attachments for the rows of that table (**Data Grouping #3**). Log the device's table-level synchronization status for these tables after processing each table.
10. report overall information about the device's synchronization status and information about the device model, etc. at the end of the synchronization interaction.

Verify *appId* support

```
@GET
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*AppNameList*/ getAppNames()
    throws AppNameMismatchException,
           PermissionDeniedException,
                       ODKDatastoreException;
```

Where the response is a list of supported *appId* values.

The current server endpoints only support a single *appId*.

```
@JacksonXmlRootElement(localName="appNames")
public class AppNameList extends ArrayList<String> {
}
```

Authenticate user

```
@GET
@Path("{appId}/privilegesInfo")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*PrivilegesInfo*/ getPrivilegesInfo()
    throws AppNameMismatchException,
               PermissionDeniedException,
                       ODKDatastoreException,
                       ODKTaskLockException;
```

The system current expects a BasicAuth authentication header.

Some server implementations can also accept an "Authorization: Bearer …" header as an, e.g., Oauth2 token.

The authentication header information is verified against the user list.

If successful, a *PrivilegesInfo* object is returned. This object contains the internal user_id that identifies this user and the friendly name (full_name) of the user. It also provides the user's default group, if configured, and the list of privileges that the user has.

That list will consist of *ROLE_…* and *GROUP_…* values. The *ROLE_…* values are predefined permissions within the ODK tools. The *GROUP_…* values are user-defined and generally correspond to organizational groups to which users belong. This allows application designers to create workflows on the device that are appropriate for the organizational privileges of the user on that device.

The returned object is defined as:

```
@JacksonXmlRootElement(localName="privilegesInfo")
public class PrivilegesInfo {

  /**
    * User id -- this may be more fully-qualified than the user identity␣
↪information
    * that the client used for login (the server may have provided auto-
↪completion
    * of a qualifying domain, etc.). The client should update their user
    * identity property to this value.
    */
  @JsonProperty(required = true)
  private String user_id;

  /**
    * Friendly full name for this user. Could be used for display.
    */
  @JsonProperty(required = false)
  private String full_name;

  /**
    * Default group
    */
  @JsonProperty(required = false)
  private String defaultGroup;


  /**
    * The roles and groups this user belongs to.
    * This is sorted alphabetically.
```

```
    */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="roles")
  private ArrayList<String> roles
}
```

Obtain Users List

```
@GET
@Path("{appId}/usersInfo")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*UserInfoList*/ getUsersInfo()
   throws AppNameMismatchException,
              PermissionDeniedException,
                      ODKDatastoreException,
                      ODKTaskLockException;
```

This list may or may not be pruned based upon the privileges of the requesting user. i.e., unprivileged users might only see themselves in this list.

This list is useful if the requesting user has the privileges needed to alter the permissions columns of a table's row. They can use this list to select the user to assign ownership to based upon the user's friendly name (full_name) instead of the user_id (the internal string identifying that user), etc.

The *UserInfoList* and *UserInfo* objects are defined as:

```
@JacksonXmlRootElement(localName="userInfoList")
public class UserInfoList extends ArrayList<UserInfo> {
}
```

and

```
@JacksonXmlRootElement(localName="userInfo")
public class UserInfo {

  /**
  * user id (unique)
  */
  @JsonProperty(required = true)
  private String user_id;
```

```
  /**
  * display name of user (may not be unique)
  */
  @JsonProperty(required = true)
  private String full_name;

  /**
  * The privileges this user has.
  * Sorted.
  */
  @JsonProperty(required = true)
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="roles")
  private ArrayList<String> roles;
}
```

Data Grouping #1 REST Synchronization API

The sync workflow for this step is:

1. obtain a manifest of the application-level files suitable for this client device.
2. compare the application-level files on the device against the manifest entry. If different, download the file, if not present on the server, delete it.

Substitution Term *odkClientVersion*

The *odkClientVersion* substitution term enables different sets of files to be delivered to different clients. The primary need for this is for configuration settings files that must be linked to a specific version of an installed tool (APK), or for HTML files that invoke a JavaScript API exposed by a specific version of a tool (APK), so that the appropriate implementation of that interface is used for the specific version of the tool (APK) present on the device.

This term is the 100's digit of the build revision. E.g., for rev 210, this is '2'.

This term is limited to 10 characters in length.

Obtain Supported *odkClientVersion*

```
@GET
@Path("{appId}/clientVersions")
@Produces({"application/json",
```

```
            "text/xml;charset=UTF-8",
            "application/xml;charset=UTF-8"})
public Response /*ClientVersionList*/ getOdkClientVersions()
    throws AppNameMismatchException,
                PermissionDeniedException,
                        ODKDatastoreException,
                        ODKTaskLockException;
```

This returns a list of the *odkClientVersion* values supported by this server. This is used to fast-fail a synchronization attempt against a server when that server does not have any configuration suitable for the indicated *odkClientVersion*. This commonly happens when an application designer intends to reset the app server with their configuration files, but instead syncs.

Note: Resetting the application server for a '3' client version will not damage or alter the '2' client version files. As long as the data table structures are not altered, the two client versions can coexist on the server.

This provides an upgrade path across incompatible client versions.

The returned list is just a list of strings:

```
@JacksonXmlRootElement(localName="clientVersions")
public class ClientVersionList extends ArrayList<String> {
}
```

Manifest REST API

```
@GET
@Path("{appId}/manifest/{odkClientVersion}")
@Produces({"application/json",
            "text/xml;charset=UTF-8",
            "application/xml;charset=UTF-8"})
public Response /*OdkTablesFileManifest*/ getAppLevelFileManifest();
```

Requests the manifest of all app-level files for an *appId* and *odkClientVersion*.

The data structure returned is:

```
@JacksonXmlRootElement(localName="manifest")
public class OdkTablesFileManifest {

  /**
    * The entries in the manifest.
```

```
    * Ordered by filename and md5hash.
    */
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="file")
  private ArrayList<OdkTablesFileManifestEntry> files;
}
```

and here:

```
public class OdkTablesFileManifestEntry {

  /**
    * This is the name of the file relative to
    * the either the 'config' directory (for
    * app-level and table-level files) or the
    * row's attachments directory (for row-level
    * attachments).
    *
    * I.e., for the new directory structure,
    * if the manifest holds configpath files, it is under:
    *   /sdcard/opendatakit/{appId}/config
    * if the manifest holds rowpath files, it is under:
    *   /sdcard/opendatakit/{appId}/data/attachments/{tableId}/{rowId}
    */
  public String filename;

  @JsonProperty(required = false)
  public Long contentLength;

  @JsonProperty(required = false)
  public String contentType;

  /**
    * This is the md5hash of the file, which will be used
    * for checking whether or not the version of the file
    * on the phone is current.
    */
  @JsonProperty(required = false)
  public String md5hash;

  /**
    * This is the url from which the current version of the file can be
    * downloaded.
    */
  @JsonProperty(required = false)
  public String downloadUrl;
}
```

e.g., for JSON:

```
{
  "files": [
    {
      "filename": "assets\/app.properties",
      "contentLength": 730,
      "contentType": "application\/octet-stream",
      "md5hash": "md5:aa47d6c0c2b63a5b99c54e5b2630be42",
      "downloadUrl": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪files\/2\/assets\/app.properties"
    },
    {
      "filename": "assets\/changeAccessFilters.html",
      "contentLength": 3202,
      "contentType": "text\/html",
      "md5hash": "md5:78d7402bdab8709b7c35d59ac7048689",
      "downloadUrl": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪files\/2\/assets\/changeAccessFilters.html"
    },
    ...
  ]
}
```

e.g., for XML:

```
<?xml version="1.0"?>
<manifest>
    <file>
        <filename>assets/app.properties</filename>
        <contentLength>730</contentLength>
        <contentType>application/octet-stream</contentType>
        <md5hash>md5:aa47d6c0c2b63a5b99c54e5b2630be42</md5hash>
        <downloadUrl>https://msundt-test.appspot.com:443/odktables/default/files/
↪2/assets/app.properties</downloadUrl>
    </file>
    <file>
        <filename>assets/changeAccessFilters.html</filename>
        <contentLength>3202</contentLength>
        <contentType>text/html</contentType>
        <md5hash>md5:78d7402bdab8709b7c35d59ac7048689</md5hash>
        <downloadUrl>https://msundt-test.appspot.com:443/odktables/default/files/
↪2/assets/changeAccessFilters.html</downloadUrl>
    </file>
</manifest>
```

Download App-Level File REST API

```
@GET
@Path("{appId}/files/{odkClientVersion}/{filePath:.*}")
@Produces({"*"})
public Response getFile(@QueryParam("as_attachment") String asAttachment)
       throws IOException, ODKTaskLockException;
```

If a query parameter (*?as_attachment=true*) is supplied, then a *Content-Disposition* header is supplied to trigger a browser to download the file rather than attempt to display it.

Upload App-Level File REST API

```
@POST
@Path("{appId}/files/{odkClientVersion}/{filePath:.*}")
@Consumes({"*"})
public Response putFile(byte[] content)
      throws IOException, ODKTaskLockException;
```

This API is only used for updating the server configuration. During the normal client synchronization workflow, this API is not invoked.

Delete App-Level File REST API

```
@DELETE
@Path("{appId}/files/{odkClientVersion}/{filePath:.*}")
public Response deleteFile()
      throws IOException, ODKTaskLockException;
```

This API is only used for updating the server configuration. During the normal client synchronization workflow, this API is not invoked.

Data Grouping #2 REST Synchronization API

Synchronizing table-level configuration and data involves:

1. Getting the list of available tables from the server
2. Verifying that the table definition on the server and client match
3. Getting the table-level configuration and files to the client.

The first two steps involve the table API and the table definition API. The data structures used by these APIs will be discussed after the APIs are presented.

Data Grouping #2 REST Synchronization – Table API and Table Definition API

The table APIs manipulate *TableResource* objects and lists. A *TableResource* identifies the table, information about the earliest and latest update to the data rows in the table, and the *schemaETag* for the table.

The server generates a new, unique, *schemaETag* every time it creates or modifies the table schema. If you create a table, destroy it, then re-create it, the new table will be given a new *schemaETag*.

Creating a table registers a *TableDefinition* for that dataset with the server and creates the necessary database tables for it. Using the *schemaETag*, clients can request the *TableDefinitionResource* for any dataset on the server; that resource consists of the *TableDefinition* and additional information.

Deleting a table on the server involves deleting the specific *TableDefinition* for that tableId's current *schemaETag*.

To prevent data loss, clients that encounter an unexpected *schemaETag* should sync their data as if for the first time.

List All Table Resources API

```
@GET
@Path("{appId}/tables")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*TableResourceList*/ getTables(@QueryParam("cursor") String␣
↪cursor, @QueryParam("fetchLimit") String fetchLimit)
    throws ODKDatastoreException,
           AppNameMismatchException,
           PermissionDeniedException,
                       ODKTaskLockException;
```

If the server does not return the entire set of tables, it will provide a *resumeParameter* in the *TableResourceList* that can be passed in as a query parameter for subsequent requests. .. _sync-protocol-rest-sync-api-2-table-api-get-resources:

Get Table Resource API

```
@GET
@Path("{appId}/tables/{tableId}")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
```

```
           "application/xml;charset=UTF-8"})
public Response /*TableResource*/ getTable()
  throws ODKDatastoreException,
         AppNameMismatchException,
                 PermissionDeniedException,
                 ODKTaskLockException,
                 TableNotFoundException;
```

Create Table Resource API

```
@PUT
@Path("{appId}/tables/{tableId}")
@Consumes({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*TableResource*/ createTable(TableDefinition definition)
    throws ODKDatastoreException,
           AppNameMismatchException,
                   TableAlreadyExistsException,
           PermissionDeniedException,
           ODKTaskLockException,
                       IOException;
```

Get Table Definition API

```
@GET
@Path("{appId}/tables/{tableId}/ref/{schemaETag}")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*TableDefinitionResource*/ getDefinition()
  throws ODKDatastoreException,
         AppNameMismatchException,
             PermissionDeniedException,
                 ODKTaskLockException,
                 TableNotFoundException;
```

Delete Table Definition API

```
@DELETE
@Path("{appId}/tables/{tableId}/ref/{schemaETag}")
public Response /*void*/ deleteTable()
  throws ODKDatastoreException,
         AppNameMismatchException,
         ODKTaskLockException,
                 PermissionDeniedException;
```

TableResourceList, *TableResource* and *TableEntry* objects

```
@JacksonXmlRootElement(localName="tableResourceList")
public class TableResourceList {

  /**
  * pass this in to return this same result set.
  */
  @JsonProperty(required = false)
  private String webSafeRefetchCursor;

  /**
  * Alternatively, the user can obtain the elements preceding the contents of the
  * result set by constructing a 'backward query' with the same filter criteria
  * but all sort directions inverted and pass the webSafeBackwardCursor
  * to obtain the preceding elements.
  */
  @JsonProperty(required = false)
  private String webSafeBackwardCursor;

  /**
  * together with the initial query, pass this in to
  * return the next set of results
  */
  @JsonProperty(required = false)
  private String webSafeResumeCursor;

  @JsonProperty(required = false)
  private boolean hasMoreResults;

  @JsonProperty(required = false)
  private boolean hasPriorResults;

  /**
  * The entries in the manifest.
```

```
  * This is and ordered list by tableId.
  */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="tableResource")
  private ArrayList<TableResource> tables;

  /**
  * If known, the ETag of the app-level files
  * manifest is also returned.
  */
  @JsonProperty(required = false)
  private String appLevelManifestETag;
}
```

```
@JacksonXmlRootElement(localName="tableResource")
public class TableResource extends TableEntry {

  /**
  * URLs for various other parts of the API
  */

  /**
  * Get this same TableResource.
  */
  private String selfUri;

  /**
  * Get the TableDefinition for this tableId
  */
  private String definitionUri;

  /**
  * Path prefix for data row interactions
  */
  private String dataUri;

  /**
  * Path prefix for data row attachment interactions
  */
  private String instanceFilesUri;

  /**
  * Path prefix for differencing (changes-since) service.
  */
  private String diffUri;
```

```
  /**
  * Path prefix for permissions / access-control service.
  */
  private String aclUri;

  /**
  * table-level file manifest ETag (optional)
  */
  @JsonProperty(required = false)
  private String tableLevelManifestETag;
}
```

and

```
public class TableEntry implements Comparable<TableEntry> {

  /**
  * The tableId this entry describes.
  */
  private String tableId;

  /**
  * The ETag of the most recently modified data row
  */
  @JsonProperty(required = false)
  private String dataETag;

  /**
  * The ETag of the TableDefinition
  */
  @JsonProperty(required = false)
  private String schemaETag;
}
```

e.g., for JSON:

```
{
  "webSafeRefetchCursor": null,
  "webSafeBackwardCursor":
↪"H4sIAAAAAAAAAG2P3QqCQBSEXyW6jVw1SpBtQawgiAKRbuWUJ5XMjbNn2R6_
↪yKAfmsuZb2BGHi0ZTYPbpe3MfFgzX2MhnH0evmJXAs05YU9TJXpwqCQwU30wjFu4oCrSbJnk6922WCT5Uorv9A3vobWo
↪Ixn40Dv08D0JwFk8ibzaZjvyHPro9LC01GzCcIVvqs0zdCrVD4BpJir-AbMxKkwMq0-dkdYLWoBS_
↪tnxdUne90G7_BAEAAA",
  "webSafeResumeCursor":
↪"H4sIAAAAAAAAAG2PzQrCMBCEX0W8Spu2osUSA1IVBKkgxWuJ7VKDNZHNhvj4ihX8wTnOfAMzvHZoDQ5ul07b-
↪fBEdM0Y896H5gq6kSTPikKDLevBoeCSCNXRERTyAqLK96tFudkV1XJRrjj7Tt_
↪wQXYORBLFaRClQRKVSZwl02wyDuN4Nooe-uj2MHeottLSHsihhqZ3WzAeJJ0Aq9roRpEy2nL21-
↪XKrg16iU3-XC8IHXD26_LXOXEHZEOUAg4BAAA",
```

```
  "hasMoreResults": false,
  "hasPriorResults": false,
  "tables": [
    {
      "tableId": "geoweather",
      "dataETag": "uuid:d74fb991-850a-4a4c-add5-858690b97c81",
      "schemaETag": "uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8",
      "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather",
      "definitionUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/
↪default\/tables\/geoweather\/ref\/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8",
      "dataUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather\/ref\/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8\/rows",
      "instanceFilesUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/
↪default\/tables\/geoweather\/ref\/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8\/
↪attachments",
      "diffUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather\/ref\/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8\/diff",
      "aclUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather\/acl",
      "tableLevelManifestETag": "19260e15"
    },
    {
      "tableId": "geoweather_conditions",
      "dataETag": "uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96",
      "schemaETag": "uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1",
      "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions",
      "definitionUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/
↪default\/tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-
↪ac6cd8bf98b1",
      "dataUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1\/
↪rows",
      "instanceFilesUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/
↪default\/tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-
↪ac6cd8bf98b1\/attachments",
      "diffUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1\/
↪diff",
      "aclUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/acl",
      "tableLevelManifestETag": "75a915a5"
    }
  ],
  "appLevelManifestETag": "eded21dd"
}
```

e.g., for XML:


```
<tableResourceList>
    <webSafeRefetchCursor/>
    <webSafeBackwardCursor>
↪H4sIAAAAAAAAAG2P3QqCQBSEXyW6jVw1SpBtQawgiAKRbuWUJ5XMjbNn2R6_
↪yKAfmsuZb2BGHi0ZTYPbpe3MfFgzX2MhnHOevmJXAsO5YU9TJXpwqCQwU3OwjFu4oCrSbJnk6922WCT5Uorv9A3vobWo
↪Ixn40Dv08DOJwFk8ibzaZjvyHPro9LC01GzCcIVvqsOzdCrVD4BpJir-AbMxKkwMq0-dkdYLWoBS_
↪tnxdUne9OG7_BAEAAA</webSafeBackwardCursor>
    <webSafeResumeCursor>
↪H4sIAAAAAAAAAG2PzQrCMBCEX0W8Spu2osUSA1IVBKkgxWuJ7VKDNZHNhvj4ihX8wTnOfAMzvHZoDQ5ul07b-
↪fBEdM0Y896H5gq6kSTPikKDLevBoeCSCNXRERTyAqLK96tFudkV1XJRrjj7Tt_
↪wQXYORBLFaRClQRKVSZwl02wyDuN4Nooe-uj2MHeottLSHsihhqZ3WzAeJJ0Aq9roRpEy2nL2l-
↪XKrg16iU3-XC8IHXD26_LXOXEHZEOUAg4BAAA</webSafeResumeCursor>
    <hasMoreResults>false</hasMoreResults>
    <hasPriorResults>false</hasPriorResults>
    <appLevelManifestETag>eded21dd</appLevelManifestETag>
    <tableResource>
        <tableId>geoweather</tableId>
        <dataETag>uuid:d74fb991-850a-4a4c-add5-858690b97c81</dataETag>
        <schemaETag>uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8</schemaETag>
        <selfUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather</selfUri>
        <definitionUri>https://msundt-test.appspot.com:443/odktables/default/
↪tables/geoweather/ref/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8</definitionUri>
        <dataUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather/ref/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8/rows</dataUri>
        <instanceFilesUri>https://msundt-test.appspot.com:443/odktables/default/
↪tables/geoweather/ref/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8/attachments</
↪instanceFilesUri>
        <diffUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather/ref/uuid:eb4e7240-af0c-4ccb-abc5-4e537a4609f8/diff</diffUri>
        <aclUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather/acl</aclUri>
        <tableLevelManifestETag>19260e15</tableLevelManifestETag>
    </tableResource>
    <tableResource>
        <tableId>geoweather_conditions</tableId>
        <dataETag>uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96</dataETag>
        <schemaETag>uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1</schemaETag>
        <selfUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather_conditions</selfUri>
        <definitionUri>https://msundt-test.appspot.com:443/odktables/default/
↪tables/geoweather_conditions/ref/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1</
↪definitionUri>
        <dataUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather_conditions/ref/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1/rows</
↪dataUri>
```

```
        <instanceFilesUri>https://msundt-test.appspot.com:443/odktables/default/
↪tables/geoweather_conditions/ref/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1/
↪attachments</instanceFilesUri>
        <diffUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather_conditions/ref/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1/diff</
↪diffUri>
        <aclUri>https://msundt-test.appspot.com:443/odktables/default/tables/
↪geoweather_conditions/acl</aclUri>
        <tableLevelManifestETag>75a915a5</tableLevelManifestETag>
    </tableResource>
</tableResourceList>
```

TableDefinition, *Column* and *TableDefinitionResource* objects

```
@JacksonXmlRootElement(localName="tableDefinition")
public class TableDefinition {

  /**
   * Schema version ETag for the tableId's database schema.
   */
  @JsonProperty(required = false)
  private String schemaETag;

  /**
   * Unique tableId
   */
  private String tableId;

  /**
   * The columns in the table.
   */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(localName="orderedColumns")
  @JacksonXmlProperty(localName="column")
  private ArrayList<Column> orderedColumns;
}
```

```
@JacksonXmlRootElement(localName="tableDefinitionResource")
public class TableDefinitionResource extends TableDefinition {

  /**
   * Get this same TableDefinitionResource.
   */
  private String selfUri;
```

```
  /**
   * Get the TableResource for this tableId.
   */
  private String tableUri;
}
```

The *configpath* type's value is relative to the *config* directory. The 'rowpath' type's value is relative to the directory in which a *rowId* attachments are stored.

with columns defined by:

```
public class Column {
  /**
   * The tableId containing this elementKey
   */
  /**
   * The fully qualified key for this element. This is the element's database
   * column name. For composite types whose elements are individually retained
   * (e.g., geopoint), this would be the elementName of the geopoint (e.g.,
   * 'myLocation' concatenated with '_' and this elementName (e.g.,
   * 'myLocation_latitude').
   *
   * Never longer than 58 characters.
   * Never a SQL or SQLite reserved word
   * Satisfies this regex: '^\\p{L}\\p{M}*(\\p{L}\\p{M}*|\\p{Nd}|_)*$'
   */
  private String elementKey;

  /**
   * The name by which this element is referred. For composite types whose
   * elements are individually retained (e.g., geopoint), this would be simply
   * 'latitude'
   *
   * Never longer than 58 characters.
   * Never a SQL or SQLite reserved word
   * Satisfies this regex: '^\\p{L}\\p{M}*(\\p{L}\\p{M}*|\\p{Nd}|_)*$'
   */
  @JsonProperty(required = false)
  private String elementName;

  /**
   * This is the ColumnType of the field. It is either:
   *    boolean
   *    integer
   *    number
   *    configpath
```

```
 *    rowpath
 *    array
 *    array(len)
 *    string
 *    string(len)
 *    typename
 *    typename(len)
 *
 *    or
 *
 *    typename:datatype
 *    typename:datatype(len)
 *
 *    where datatype can be one of boolean, integer, number, array, object
 *
 *    Where:
 *
 *    'typename' is any other alpha-numeric name (user-definable data type).
 *
 *    The (len) attribute, if present, identifies the VARCHAR storage
 *    requirements for the field when the field is a unit of retention.
 *    Ignored if not a unit of retention.
 *
 *    The server stores:
 *
 *      integer as a 32-bit integer.
 *
 *      number as a double-precision floating point value.
 *
 *      configpath indicates that it is a relative path to a file under the
↪'config'
 *             directory in the 'new' directory structure. i.e., the relative␣
↪path is
 *             rooted from:
 *                 /sdcard/opendatakit/{appId}/config/
 *
 *      rowpath indicates that it is a relative path to a file under the row's␣
↪attachment
 *             directory in the 'new' directory structure. i.e., the relative␣
↪path is
 *             rooted from:
 *                 /sdcard/opendatakit/{appId}/data/attachments/{tableId}/
↪{rowId}/
 *
 *      array is a JSON serialization expecting one child element key
 *            that defines the data type in the array.  Array fields
 *            MUST be a unit of retention (or be nested within one).
```

```
   *
   *      string is a string value
   *
   *      anything else, if it has no child element key, it is a string
   *            (simple user-defined data type). Unless a datatype is specified.
   *
   *      anything else, if it has one or more child element keys, is a
   *            JSON serialization of an object containing those keys
   *            (complex user-defined data type).
   *
   */
  private String elementType;

  /**
   * JSON serialization of an array of strings. Each value in the
   * array identifies an elementKey of a nested field within this
   * elementKey. If there are one or more nested fields, then the
   * value stored in this elementKey is a JSON serialization of
   * either an array or an object. Otherwise, it is either an
   * integer, number or string field.
   *
   * If the elementType is 'array', the serialization is an
   * array and the nested field is retrieved via a subscript.
   *
   * Otherwise, the serialization is an object and the nested
   * field is retrieved via the elementName of that field.
   */
  @JsonProperty(required = false)
  private String listChildElementKeys;
}
```

e.g., for JSON

```
{
  "schemaETag": "uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1",
  "tableId": "geoweather_conditions",
  "orderedColumns": [
    {
      "elementKey": "Code",
      "elementName": "Code",
      "elementType": "string",
      "listChildElementKeys": "[]"
    },
    {
      "elementKey": "Description",
      "elementName": "Description",
      "elementType": "string",
```

```
      "listChildElementKeys": "[]"
    },
    {
      "elementKey": "Language",
      "elementName": "Language",
      "elementType": "string",
      "listChildElementKeys": "[]"
    }
  ],
  "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/tables\/
↪geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1",
  "tableUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions"
}
```

Data Grouping #2 REST Synchronization – Table-level Files API

To support table-specific files, a new manifest API is provided

```
@GET
@Path("{appId}/manifest/{odkClientVersion}/{tableId}")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*OdkTablesFileManifest*/ getTableIdFileManifest()
     throws ODKEntityNotFoundException,
                ODKOverQuotaException,
                        PermissionDeniedException,
                        ODKDatastoreException,
                        ODKTaskLockException;
```

The table-level files API is identical to the app-level files API. It relies upon the file naming convention to distinguish between app-level files and table-level files.

Data Grouping #3 REST Synchronization - Overview

Attachments: BLOBs and Documents

BLOBs, long strings (e.g., MySQL TEXT fields) and arbitrary files can be associated with any data row. These are stored as files and viewed as 'attachments' of the row. If a row has an attachment, the row is expected to have one or more columns in its data table that contain the path to that attachment.

For example, the ODK Tools use a *rowpath* elementType (see the Column object, presented earlier), the attachment field definition in Survey (either an *imageUri*, *audioUri* or *videoUri* object) consists of two parts, a *uriFragment* that is a *rowpath* elementType and a *contentType* that is a string containing the mime type of the attachment. The *rowpath* is a path relative to the storage location for files associated with this *rowId*. e.g.,

```
{ uriFragment: "filename.jpg",
  contentType: "image/jpg" }
```

Attachments are immutable. If an attachment is modified, it must be given a new, unique, *filepath*. The server will not accept revisions to an attachment.

Revision States

It is assumed that the client maintains a set of revision states for an individual row. These

1. *synced* - no changes to an existing record obtained from the server and all attachment changes have been handled.
2. *new_row* - a new record on the client.
3. *changed* - the client modified an existing record obtained from the server.
4. *deleted* - the client deleted an existing record obtained from the server.
5. *synced_pending_files* - the client considers the row data to be in the 'rest' state, but the attachments for this row may or may not be up-to-date.
6. *in_conflict* - the client has determined that there was both a local change to the row and another client has pushed a change to the server, so that the local change cannot be directly submitted to the server, but must instead be resolved with the server's version before being uploaded.

For a given tableId, whenever the *schemaETag* for that tableId has changed, if the client w

- reset all rows in the *in_conflict* state to their original local change status (i.e., one of *new_row*, *changed* or *deleted*),
- mark all *synced* and *synced_pending_files* rows as *new_row*.
- reset the table's last-change-processed value so that the next sync of the table's data will attempt to sync every row in the table.

This may cause all the client's rows to become in conflict with the server; it is unclear what should be the default treatment for this condition.

The server maintains a full history of all changes to a given row. Each row is identified by a *rowId*. Each row revision is identified by its (*rowId*, *rowETag*) tuple.

When a client row is synced with the server, the *rowETag* of the prior version of that *rowId* is sent up to the server (sending **null** if this is an *new_row* row) along with all the values in the row.

When a client row is in the *new_row* state, the client may optionally send **null** for the value of the *rowId*, in which case the server will assign an id.

An insert-or-update row request is successful if:

- the *rowId* does not yet exist, or
- the *rowETag* matches the value for the most recent revision to *rowId*, or
- the *rowETag* doesn't match, but the values of the most recent version of the *rowId* on the server exactly match the values sent from the client.

A delete row request is successful if:

- the *rowId* does not yet exist, or
- the *rowETag* matches the value for the most recent revision to *rowId*

If successful, any changes are applied on the server, and the client is returned the updated row (and updated *rowETag*). The client should then either delete the local copy if it was in the *deleted* state, or update its corresponding row to *synced_pending_files* if there are rowpath columns in the dataset or *synced* if not, and set *rowETag* to the value returned for *rowETag* in the updated record.

If unsuccessful, an *ETagMismatchException* error is reported back to the client, and the client should mark the row as *in_conflict*. *in_conflict* rows are not eligible to be synced until the client resolves the conflict state, usually through processing convention or user intervention.

If the row is in the *synced_pending_files* state, then the client must determine what actions it needs to perform to bring this row's attachment(s) state into concordance with the server.

Because data records can be sent up to the server before their associated attachments are sent, clients may obtain data records from the server that lack the attachment files that they reference. I.e., *ClientOne* may sync a row with an updated attachment to the server, but fail to send the attachment itself. *ClientTwo* may then sync with the server, obtain the row updates that *ClientOne* just posted, and therefore have a valid, current, row without the attachments that it references.

This is a normal condition and should be anticipated and gracefully handled by the client.

synced_pending_files treatment

There is a potential for loss of an earlier attachment if the data row is partially synced (transitioning into *synced_pending_files*) and the data row is then updated, changing the attachment, before the earlier version of the attachment is saved on the server.

Because the client is strictly forbidden from modifying the contents of the attachment file, we always know if a new attachment is created because the data row will always be modified to update the attachment path.

Similarly, because the *config* directory is static and dictated by the server, any *configpath* field in a data row does not require syncing of that referenced file with the server. It is assumed that the server already has that file. Only the *rowpath* fields in a data row need to have their attachments synced.

The server maintains a manifest of all *rowpath* attachments uploaded for all versions of the row.

The current implementation only considers attachments specified in 'rowpath' elements. If the attachment has not yet been uploaded, a NOT_FOUND is returned should that attachment be requested.

The sync mechanism first requests all rowpath files, either specifying an ETag if the file exists locally, or omitting it, to pull the file. If a request with an ETag returns NOT_MODIFIED, then the server has that file. If it returns NOT_FOUND, then the client should push the file to the server. If it returns the file, then there is an exceptional condition and the client should log an error (but it is fine to download the file – the server is still the authority for what these files should contain).

Data Grouping #3 REST Synchronization - Workflow

The normal data synchronization workflow is:

1. Request the *TableResource* for a tableId (using the Table API, defined earlier).
2. If the *dataETag* in this resource matches the last-change-processed value maintained by the client, then there are no row-value changes. Proceed to upload our changes.
3. Otherwise, use the *diffUri* to request the list of rows with recent changes. If you have no last-changed-processed value, use the *dataUri* to request all rows in the table.
4. Update client state to reflect changes on server.
5. Update the dataETag of our table to that given in the first result set (RowResourceList) of server rows or changes pulled from the server.
6. Push *new_row*, *changed* and *deleted* records up to server. Specify the table's dataETag in this request (RowList). If a 409 (CONFLICT) is returned, then go to step (3) above. Otherwise, update our table dataETag with that returned on the RowOutcomeList. Update our local state with the outcomes specified in the RowOutcomeList.

7. If the above two stages complete without errors, resolve rows in the *synced_pending_files* state by pushing / pulling attachments to / from the server. If successful, transition that row into the *synced* state.

8. Report status metrics for this table to the server.

And, at some later time: * Resolve any *in_conflict* rows (user-directed) This conflict resolution will transition rows either into a state matching that on the server, or into an updated *changed* state such that on the next synchronization those changes will be able to be successfully pushed to the server (unless those rows were changed, yet again, by another client).

Get All Data Changes Since... API

```
@GET
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/diff")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*RowResourceList*/ getRowsSince(@QueryParam("data_etag") String␣
↪dataETag, @QueryParam("cursor") String cursor, @QueryParam("fetchLimit")␣
↪String fetchLimit)
    throws ODKDatastoreException,
           PermissionDeniedException,
           InconsistentStateException,
           ODKTaskLockException, BadColumnNameException;
```

Unlike the other REST interfaces, this takes a query parameter specifying the *dataETag* from which to report the set of changed rows.

If the server cannot return the entire set of rows, it will provide a *resumeParameter* in the *RowResourceList* that can be passed in as a query parameter to generate the next grouping of rows.

Get Changesets API

```
@GET
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/diff/changeSets")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*ChangeSetList*/ getChangeSetsSince(@QueryParam("data_etag")␣
↪String dataETag, @QueryParam("sequence_value") String sequenceValue)
    throws ODKDatastoreException, PermissionDeniedException,␣
↪InconsistentStateException, ODKTaskLockException, BadColumnNameException;
```

This API is not actively used in the device's Sync implementation.

As with the previous API, this takes a query parameter specifying the *dataETag* from which to report the set of changeSets (subsequent *dataETag* values).

If the server cannot return the entire set of *dataETag* values processed since the specified *dataETag*, it will provide a *sequenceValue* in the *ChangeSetList* that can be passed in as a query parameter to generate the next grouping of set of *dataETag* values.

Get the changeSets that have been applied since the dataETag changeSet (must be a valid dataETag) or since the given sequenceValue.

These are returned in no meaningful order. For consistency, the values are sorted alphabetically. The returned object includes a sequenceValue that can be used on a subsequent call to get all changes to this table since this point in time.

The *ChangeSetList* contains a list of *dataETag* strings and a *sequenceValue* that allows the client to request changeSets that have been processed since this set of changeSets were returned.

```
@JacksonXmlRootElement(localName="changeSetList")
public class ChangeSetList {

  /**
   * The dataETag values.
   */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="changeSet")
  private ArrayList<String> changeSets;

  /**
   * The dataETag value of the table at the START of this request.
   */
  @JsonProperty(required = false)
  private String dataETag;

  /**
   * The sequenceValue of the server at the START of this request.
   * A monotonically increasing string.
   */
  @JsonProperty(required = false)
  private String sequenceValue;
}
```

Get Changeset Rows API

```
@GET
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/diff/changeSets/{dataETag}")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*RowResourceList*/ getChangeSetRows(@QueryParam("active_only")␣
↪String isActive,
                            @QueryParam("cursor") String cursor, @QueryParam(
↪"fetchLimit") String fetchLimit)
    throws ODKDatastoreException, PermissionDeniedException,
                InconsistentStateException, ODKTaskLockException,
                        BadColumnNameException;
```

This API is not actively used in the device's Sync implementation.

This fetches the set of row changes corresponding to this changeSet *dataETag*.

If the "active_only" query parameter is provided, only the changes that are in this change set that are currently active (have not been superseded) will be returned.

Get All Data Rows API

```
@GET
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/rows")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*RowResourceList*/ getRows(@QueryParam("cursor") String cursor,␣
↪@QueryParam("fetchLimit") String fetchLimit)
        throws ODKDatastoreException, PermissionDeniedException,
                InconsistentStateException, ODKTaskLockException,
                BadColumnNameException;
```

If the server cannot return the entire set of rows, it will provide a *resumeParameter* in the *RowResourceList* that can be passed in as a query parameter to generate the next grouping of rows.

The *RowResourceList* returned contains the dataETag of the last change processed on the server.

Note: Later requests with resume cursors may return different values for this dataETag..

The value in the first result should be compared with the value returned at the end of the chain of requests. If this value does change, the client should update its table dataETag to the first value and issue a new request using the first dataETag. This will pull the changes that were occurring as the first result set was being pulled and processed by the client. Only once the dataETag does not change can the client be assured that it does not have any partial changeSets.

Get a Data Row API

```
@GET
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/rows/{rowId}")
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*RowResource*/ getRow()
       throws ODKDatastoreException,
              PermissionDeniedException, InconsistentStateException,
              ODKTaskLockException, BadColumnNameException;
```

Gets the current values for a specific rowId.

Alter Data Rows (Insert, Update or Delete)API

```
@PUT
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/rows")
@Consumes({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
@Produces({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
public Response /*RowOutcomeList*/ alterRows(RowList rows)
       throws ODKTaskLockException, ODKDatastoreException,
              PermissionDeniedException, BadColumnNameException,
                                InconsistentStateException,␣
↪TableDataETagMismatchException;
```

This REST interface takes a *RowList* that must contain the dataETag of the table that matches the one on the server. If the value does not match, the server returns 409 (CONFLICT) and the client should use the diff API to fetch changes from the server before re-attempting to alter data on the server. If the dataETag does match, a *RowOutcomeList* is returned with the actions taken by the server.

> **Warning:** Some row changes may fail, and some may succeed (e.g., due to permissions violations).

The client should process the *RowOutcome* information to update its local database to match that on the server. For bandwidth efficiency, large portions of the *RowOutcome* object will be null upon success.

The *RowOutcomeList* contains the dataETag of the resulting change set on the server. The client should update its table dataETag to match this value.

Row and *RowList*, *RowResource* and *RowResourceList*, *RowOutcome* and *RowOutcomeList* Objects

RowList is a list of rows:

```
@JacksonXmlRootElement(localName="rowList")
public class RowList {

  /**
   * The entries in the manifest.
   */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="row")
  private ArrayList<Row> rows;

  /**
   * The dataETag of the table at the START of this request.
   */
  @JsonProperty(required = false)
  private String dataETag;
}
```

RowOutcomeList is a list of row outcomes:

```
@JacksonXmlRootElement(localName="rowList")
public class RowOutcomeList {

  /**
   * The URL that returns the TableResource for this table.
   */
  @JsonProperty(required = false)
  private String tableUri;

  /**
```

```
   * The entries in the manifest.
   */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="row")
  private ArrayList<RowOutcome> rows;

  /**
   * The dataETag for the changes made by this request.
   */
  @JsonProperty(required = false)
  private String dataETag;
}
```

RowResourceList is a list of row resources:

```
@JacksonXmlRootElement(localName="rowResourceList")
public class RowResourceList {

  /**
   * The entries in the manifest.
   */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(useWrapping=false)
  @JacksonXmlProperty(localName="rowResource")
  private ArrayList<RowResource> rows;

  /**
   * The dataETag of the table at the START of this request.
   */
  @JsonProperty(required = false)
  private String dataETag;

  /**
   * The URL that returns the TableResource for this table.
   */
  private String tableUri;

  /**
   * together with the initial query, pass this in to
   * return this same result set.
   */
  @JsonProperty(required = false)
  private String webSafeRefetchCursor;

  /**
   * Alternatively, the user can obtain the elements preceding the contents of␣
↪the
```

```
   * result set by constructing a 'backward query' with the same filter criteria
   * but all sort directions inverted and pass the webSafeBackwardCursor
   * to obtain the preceding elements.
   */
  @JsonProperty(required = false)
  private String webSafeBackwardCursor;

  /**
   * together with the initial query, pass this in to
   * return the next set of results
   */
  @JsonProperty(required = false)
  private String webSafeResumeCursor;

  @JsonProperty(required = false)
  private boolean hasMoreResults;

  @JsonProperty(required = false)
  private boolean hasPriorResults;
```

RowResource extends a *Row* and supplies a self-reference URL.

```
@JacksonXmlRootElement(localName="rowResource")
public class RowResource extends Row {

  /**
   * The URL that returns this RowResource.
   */
  private String selfUri;
}
```

RowOutcome also extends *Row* with a self-reference URL and an *OutcomeType*:

```
@JacksonXmlRootElement(localName = "rowResource")
public class RowOutcome extends Row {

  /**
   * Possible values:
   * <ul>
   * <li>UNKNOWN -- initial default value</li>
   * <li>SUCCESS -- rowETag, dataETagAtModification, filterScope updated</li>
   * <li>DENIED -- permission denied -- just the rowId is returned</li>
   * <li>IN_CONFLICT -- server record is returned (in full)</li>
   * <li>FAILED -- anonymous insert conflict (impossible?) or
   *               delete of non-existent row -- just rowId is returned</li>
   * </ul>
   */
```

```
  public enum OutcomeType {
    UNKNOWN, SUCCESS, DENIED, IN_CONFLICT, FAILED
  }

  /**
   * The URL that returns this RowResource.
   */
  @JsonProperty(required = false)
  private String selfUri;

  @JsonProperty(required = false)
  private OutcomeType outcome = OutcomeType.UNKNOWN;
}
```

Row contains the data for a row.

```
public class Row {

  /**
   * PK identifying this row of data.
   */
  @JacksonXmlProperty(localName = "id")
  @JsonProperty(value = "id", required = false)
  private String rowId;

  /**
   * identifies this revision of this row of data.
   * (needed to support updates to data rows)
   * (creation is a revision from 'undefined').
   */
  @JsonProperty(required = false)
  private String rowETag;

  /**
   * identifies the service-level
   * interaction during which this
   * revision was made. Useful for
   * finding coincident changes
   * and prior/next changes.
   */
  @JsonProperty(required = false)
  private String dataETagAtModification;

  /**
   * deletion is itself a revision.
   */
  @JsonProperty(required = false)
```

```
private boolean deleted;

/**
 * audit field returned for
 * archive/recovery tools.
 */
@JsonProperty(required = false)
private String createUser;

/**
 * audit field returned for
 * archive/recovery tools
 */
@JsonProperty(required = false)
private String lastUpdateUser;

/**
 * OdkTables metadata column.
 *
 * The ODK Survey form that
 * was used when revising this
 * row.
 *
 * This can be useful for
 * implementing workflows.
 * I.e., if savepointTyp is
 * COMPLETE with this formId,
 * then enable editing with
 * this other formId.
 */
@JsonProperty(required = false)
private String formId;

/**
 * OdkTables metadata column.
 *
 * The locale of the device
 * that last revised this row.
 */
@JsonProperty(required = false)
private String locale;

/**
 * OdkTables metadata column.
 *
 * One of either COMPLETE
 * or INCOMPLETE. COMPLETE
```

```
   * indicates that the formId
   * used to fill out the row
   * has validated the entered
   * values.
   */
  @JsonProperty(required = false)
  private String savepointType;

  /**
   * OdkTables metadata column.
   *
   * For Mezuri, the timestamp
   * of this data value.
   *
   * For ODK Survey, the last
   * save time of the survey.
   *
   * For sensor data,
   * the timestamp for the
   * reading in this row.
   */
  @JsonProperty(required = false)
  private String savepointTimestamp;

  /**
   * OdkTables metadata column.
   *
   * For ODK Survey, the user
   * that filled out the survey.
   *
   * Unclear what this would be
   * for sensors.
   *
   * For Mezuri, this would be
   * the task execution ID that
   * created the row.
   */
 @JsonProperty(required = false)
  private String savepointCreator;

  /**
   * RowFilterScope is passed down to device.
   *
   * Implements DEFAULT, MODIFY, READ_ONLY, HIDDEN
   * with rowOwner being the "owner" of the row.
   *
   * It is passed down to the
```

```
   * device so that the
   * device can do best-effort
   * enforcement of access control
   * (trusted executor)
   */
  @JacksonXmlProperty(localName = "filterScope")
  @JsonProperty(value = "filterScope", required = false)
  private RowFilterScope rowFilterScope;

  /**
   * Array of user-defined column name to
   * the string representation of its value.
   * Sorted by ascending column name.
   */
  @JsonProperty(required = false)
  @JacksonXmlElementWrapper(localName="orderedColumns")
  @JacksonXmlProperty(localName="value")
  private ArrayList<DataKeyValue> orderedColumns;
}
```

where *RowFilterScope* is:

```
public class RowFilterScope {

  /**
   * Type of Filter.
   *
   * Limited to 10 characters
   */
  public enum Access {
    FULL, MODIFY, READ_ONLY, HIDDEN,
  }

  @JsonProperty(required = false)
  private Access defaultAccess;

  @JsonProperty(required = false)
  private String rowOwner;

  @JsonProperty(required = false)
  private String groupReadOnly;

  @JsonProperty(required = false)
  private String groupModify;

  @JsonProperty(required = false)
  private String groupPrivileged;
```

```
}
```

and *DataKeyValue* is:

```
public class DataKeyValue {
  @JacksonXmlProperty(isAttribute=true)
  public String column;

  @JacksonXmlText
  public String value;
}
```

e.g., for JSON


```
{
  "rows": [
    {
      "rowETag": "uuid:e818c096-c3c6-4ec6-ac40-015ddfbef303",
      "dataETagAtModification": "uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96",
      "deleted": false,
      "createUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "lastUpdateUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "formId": "geoweather_conditions",
      "locale": "en_US",
      "savepointType": "COMPLETE",
      "savepointTimestamp": "2017-07-21T19:13:52.594000000",
      "savepointCreator": "username:msundt",
      "orderedColumns": [
        {
          "column": "Code",
          "value": "clear"
        },
        {
          "column": "Description",
          "value": "Clear skies on 5.0"
        },
        {
          "column": "Language",
          "value": "en"
        }
      ],
      "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1\/
↪rows\/uuid:50caa4ef-4f7f-4229-80b6-8e2d44026b90",
      "id": "uuid:50caa4ef-4f7f-4229-80b6-8e2d44026b90",
      "filterScope": {
        "defaultAccess": "FULL",
```

```
        "rowOwner": null,
        "groupReadOnly": null,
        "groupModify": null,
        "groupPrivileged": null
      }
    },
    {
      "rowETag": "uuid:a3a8e4b8-295c-410e-a9ec-7577e386799f",
      "dataETagAtModification": "uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96",
      "deleted": false,
      "createUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "lastUpdateUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "formId": "geoweather_conditions",
      "locale": "en_US",
      "savepointType": "COMPLETE",
      "savepointTimestamp": "2017-07-21T19:13:02.633000000",
      "savepointCreator": "username:msundt",
      "orderedColumns": [
        {
          "column": "Code",
          "value": "rain"
        },
        {
          "column": "Description",
          "value": "Raining on 5.0"
        },
        {
          "column": "Language",
          "value": "en"
        }
      ],
      "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1\/
↪rows\/uuid:7fba9aa0-df29-4e3b-a390-e07b4ee48fe8",
      "id": "uuid:7fba9aa0-df29-4e3b-a390-e07b4ee48fe8",
      "filterScope": {
        "defaultAccess": "READ_ONLY",
        "rowOwner": null,
        "groupReadOnly": null,
        "groupModify": null,
        "groupPrivileged": null
      }
    },
    {
      "rowETag": "uuid:34847487-3f5d-4f66-814c-602e2dc4d6d2",
      "dataETagAtModification": "uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96",
      "deleted": false,
```

```
      "createUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "lastUpdateUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "formId": "geoweather_conditions",
      "locale": "en_US",
      "savepointType": "COMPLETE",
      "savepointTimestamp": "2017-07-21T19:14:32.127000000",
      "savepointCreator": "username:msundt",
      "orderedColumns": [
        {
          "column": "Code",
          "value": "thunderstorm"
        },
        {
          "column": "Description",
          "value": "Thunderstorm on 5.0"
        },
        {
          "column": "Language",
          "value": "en"
        }
      ],
      "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1\/
↪rows\/uuid:7fba9aa0-df29-4e3b-a390-e08b4ee48fe8",
      "id": "uuid:7fba9aa0-df29-4e3b-a390-e08b4ee48fe8",
      "filterScope": {
        "defaultAccess": "READ_ONLY",
        "rowOwner": null,
        "groupReadOnly": null,
        "groupModify": null,
        "groupPrivileged": null
      }
    },
    {
      "rowETag": "uuid:9c13fa4c-62c0-4a53-9038-34514c9b17f0",
      "dataETagAtModification": "uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96",
      "deleted": false,
      "createUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "lastUpdateUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "formId": "geoweather_conditions",
      "locale": "en_US",
      "savepointType": "COMPLETE",
      "savepointTimestamp": "2017-07-21T19:12:36.747000000",
      "savepointCreator": "username:msundt",
      "orderedColumns": [
        {
          "column": "Code",
```

```
          "value": "drizzle"
        },
        {
          "column": "Description",
          "value": "Light rain (drizzle) on 5.0"
        },
        {
          "column": "Language",
          "value": "en"
        }
      ],
      "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1\/
↪rows\/uuid:88b2edbc-092a-44c2-9736-8d50f6e44704",
      "id": "uuid:88b2edbc-092a-44c2-9736-8d50f6e44704",
      "filterScope": {
        "defaultAccess": "HIDDEN",
        "rowOwner": null,
        "groupReadOnly": null,
        "groupModify": null,
        "groupPrivileged": null
      }
    },
    {
      "rowETag": "uuid:82d61608-a870-4976-baa8-2c7af974f74e",
      "dataETagAtModification": "uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96",
      "deleted": false,
      "createUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "lastUpdateUser": "uid:msundt|2014-10-03T16:48:04.320+0000",
      "formId": "geoweather_conditions",
      "locale": "en_US",
      "savepointType": "COMPLETE",
      "savepointTimestamp": "2017-07-21T19:15:04.655000000",
      "savepointCreator": "username:msundt",
      "orderedColumns": [
        {
          "column": "Code",
          "value": "partly_cloudy"
        },
        {
          "column": "Description",
          "value": "Partly cloudy on 5.0"
        },
        {
          "column": "Language",
          "value": "en"
        }
```

```
      ],
      "selfUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions\/ref\/uuid:b48be1ae-d861-4453-97a2-ac6cd8bf98b1\/
↪rows\/uuid:999f57ec-d866-45bc-ad54-52c57489d54b",
      "id": "uuid:999f57ec-d866-45bc-ad54-52c57489d54b",
      "filterScope": {
        "defaultAccess": "MODIFY",
        "rowOwner": null,
        "groupReadOnly": null,
        "groupModify": null,
        "groupPrivileged": null
      }
    }
  ],
  "dataETag": "uuid:e93ead34-8ee1-4c5c-9d25-7732a5ec9c96",
  "tableUri": "https:\/\/msundt-test.appspot.com:443\/odktables\/default\/
↪tables\/geoweather_conditions",
  "webSafeRefetchCursor": null,
  "webSafeBackwardCursor": "H4sIAAAAAAAAAG2PW4vCMBCF_
↪4r4KmnSUO2FGBB1YUFckLKvMjXT3WBtJZlQf_6W7YIX9jzMw5nvHDjqFJzv3OR2aVq_nH4TXQvO-
↪76Puiu2BgjOlqLOffERnGoFRM5WgXAPF9TH9WG7Kt8_9sfNqtwq_vy9w5_
↪QBNRSxCkTKZOilHEh00JkUT6PZ2LQQ3aEVXB2B54OSMG1aEY3BGuKuTgBJFizpE6HI2XOMlEtWIbSJImQiyof-
↪v7NK-vfOteDM-vfRbqGxqPir7b6W6x_ACeKKe0jAQAA",
  "webSafeResumeCursor":
↪"H4sIAAAAAAAAAG2Py2rDMBREf6VkW2QpimVFRhWYPCBQUgimWyNbaiOS2OH6Cvfza-
↪pC0pJZDTNnFqObCHOHT1-Xc9u_
↪zI6I15zSYRiS7upbZ9GeAiYdfNIJnBltESHUEf3eXrypVodNUe7e9tW6KDea_
↪m1v8Ls9R284m0vCJOGs5POcy5yphPHFMxt1t51gHSG82h4PHiO03k1pjMHlSqkPIX1D3DLLSCrqhlgnUiJ4I2S6VKOvN
↪p_r3sPkG0rFWryIBAAA",
  "hasMoreResults": false,
  "hasPriorResults": false
}
```


The *dataETagAtModification* field tracks the change entry that can be used with the **Get All Data Changes Since... API** to return the changes in the data table from this row's last data change (as indicated by the *rowETag*).

The *createUser* and *lastUpdateUser* fields may be set and returned by the server. These are intended for data-dump and data-restore functionality and are not normally provided by a client.

The *formId* field identifies the ODK Survey form that last modified this record. This is useful for implementing multi-stage client workflows.

The *locale* field tracks the last ODK Survey locale in which the form was opened and perhaps modified.

The *savepointType* is one of *INCOMPLETE* or *COMPLETE*; it indicates whether the data is considered to be in a possibly-incomplete state or if it is complete (i.e., in ODK Survey, if it has been validated and marked as finalized). Together with the *formId*, this can indicate whether the client processing can advance from one workflow stage (*formId*) to another (i.e., when the record is 'COMPLETE' in the current stage) or whether to stall within the current workflow stage (*formId*). For autonomous data publishing (e.g., ODK Sensors Framework), this should be set to *COMPLETE*.

The *savepointTimestamp* is the timestamp of the last save of this data record, as reported on the client (whose time clock may be inaccurate).

The *savepointCreator* is the entity modifying/writing this data row. For ODK Survey, this is the user as identified by the Android device.

The *filterScope* should default to *{type: 'Default', value: null}*. It is used to control access to the data record. Future updates to this protocol will likely make this unmodifiable on the server unless the requesting user has appropriate permissions. The contents, interpretation and use of this field is evolving at this time.

The *values* map holds the data values that the user has defined.

Get Manifest of Attachments API

This returns all attachments (both current and historical) for the given *rowId* on the server.

This uses the same return structure as the Table-level and App-level manifest, but the path is relative to the directory in which the *rowId* attachments are stored on the client.

There is both a multipart file download/upload API and an individual-file download/upload API. The Android client uses the multipart file API.

Multipart Get Attachment API

```
@POST
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/attachments/{rowId}/download")
@Consumes({"application/json",
           "text/xml;charset=UTF-8",
           "application/xml;charset=UTF-8"})
@Produces({"multipart/form-data"})
public Response getFiles(OdkTablesFileManifest manifest) throws IOException,␣
↪ODKTaskLockException, PermissionDeniedException;
```

Returns a multipart form containing the files.

To Do: Verify that a part's name is the filename relative to the folder holding attachments for the *rowId*.

Multipart Put Attachment API

To Do: verify that a part's name is the filename relative to the folder holding attachments for the *rowId*.

Returns a string describing error on failure, otherwise empty and Status.CREATED.

Get Attachment API

```
@GET
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/attachments/{rowId}/file/
↪{filePath:.*}")
@Produces({"*"})
public Response getFile(@QueryParam("as_attachment") String asAttachment)
      throws IOException, ODKTaskLockException, PermissionDeniedException;
```

The *filePath* is relative to the folder holding attachments for the *rowId*.

Put Attachment API

```
@POST
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/attachments/{rowId}/file/
↪{filePath:.*}")
@Consumes({"*"})
public Response putFile(byte[] content)
      throws IOException, ODKTaskLockException, PermissionDeniedException,␣
↪ODKDatastoreException;
```

Report table status metrics

```
@POST
@Path("{appId}/tables/{tableId}/ref/{schemaETag}/installationStatus")
@Consumes({"application/json"})
public Response /*OK*/ postInstallationStatus(Object body)
    throws AppNameMismatchException,
               PermissionDeniedException,
                       ODKDatastoreException,
                       ODKTaskLockException;
```

This takes a generic JSON object and stores it on the server.

The JSON object (serialization) should be less than 4000 characters in length.

This API is used to report the outcome of the synchronization of this table on the client. In particular, it can be used to determine which devices are up-to-date with respect to the server's table contents (i.e., have no conflicts). That information is useful for determining when rows on the server can be permanently removed after having been marked as deleted.

Report device info and overall sync state

```
@POST
@Path("{appId}/installationInfo")
@Consumes({"application/json"})
public Response /*OK*/ postInstallationInfo(Object body)
    throws AppNameMismatchException,
               PermissionDeniedException,
                       ODKDatastoreException,
                       ODKTaskLockException;
```

This API is invoked after the sync has completed on the client.

This takes a generic JSON object and stores it on the server.

The JSON object (serialization) should be less than 4000 characters in length.

It can be used to determine whether a client successfully synced and provides information mapping the client's *X-OpenDataKit-Installation-Id* back to a physical device (info on the type of device and the reported Android ID for the device are in the Android implementation's object).

4.16.4 Build Scripts

Prerequisites

The Android tools, like most Android projects, use Gradle for compilation scripting. Before reading this document, you should be familiar with Gradle for Android.

Directory Structure

The build scripts for the Android tools expect a particular directory structure. They expect a parent directory that contains each of them at the same level. This is **optional**.

If you had all of the Android tools checked out your directory structure would look like this:

```
/opendatakit/
    /androidlibrary/
    /androidcommon/
```

```
/gradle-config/
/scan/
/sensorsframework/
/sensorsinterface/
/services/
/survey/
/tables/
```

There are two cases where this directory structure makes a difference:

- **Library Projects**:
  - If androidlibrary and androidcommon are present in the same directory, according to the above structure, as the Android tools, then they will build against the local copy. If you want to make changes to Services and androidlibrary simultaneously, for example, this structure would be necessary.
  - If the library projects are not present in the above configuration then a prebuilt binary will be downloaded according to the flavor you are building. For example, new binaries are posted on Snapshot for each commit, or on Master for each release.
- **Gradle Config**: If the gradle-config project is present in the above configuration, the gradle files in that folder will be used. Otherwise the release version specifies in `settings.gradle` will be used.

Building the Android Tools

The simplest way to build the tools is often to press the build button in Android Studio. However, the command line can also be used. To invoke the gradle wrapper, enter the root level of the project to be built and run a command that looks like this:

```
./gradlew clean assembleSnapshotBasic
```

If you are on **`Windows`** use `gradlew.bat` instead.

Note: If you are building with Android Studio, you will need to select the correct build variant. This is important when you don't have androidlibrary or androidcommon in your *Directory Structure*. These are discussed more in the *next section*.

Flavors

The Android tools use two dimensions of product flavors. The first dimension determines the version of the dependencies to pull. Each of the Android tools depends on the androidlibrary

library project, and some depend on androidcommon as well. Binary versions of these are posted to **Maven** and **Ivy** repositories corresponding to the latest version of each of the three branches:

- **Snapshot** is used if you are running the *development* branch. A new version of the libraries is automatically posted with each new commit that is merged.
- **Demo** is used if you are running the *demo* branch.
- **Master** is used if you are running the *master* branch. These are release versions that have been tested and posted by hand.

> **Warning:** The ODK 2 tools prefers pull requests to *development*. In unusual circumstances when *development* is undergoing heavy change we may accept pull requests to *demo* or *master* depending on the level of incompatibility that might exist.

The other dimension determines whether to apply changes necessary to run the UI tests. The two options are:

- **Basic** is used for normal builds
- **Uitest** is used for builds that will run the UI tests.

Therefore, if you wanted to build the normal version of the *master* branch, you would run:

```
./gradlew clean assembleMasterBasic
```

See *UI Testing* for an example of the UI testing flavor.

Running Lint

To run Lint:

```
./gradlew clean lintSnapshotBasicRelease
```

Unit Testing

To run unit tests:

```
./gradlew clean testSnapshotBasicDebug
```

Connected Testing

To run the connected device tests:

```
./gradlew clean connectedSnapshotBasicDebugAndroidTest
```

UI Testing

To run the UI tests:

```
./gradlew clean connectedSnapshotUitestDebugAndroidTest
```

Note: The previous commands can be run together. For example, to run the two unit test commands you would run:

```
./gradlew clean testSnapshotBasicDebug connectedSnapshotBasicDebugAndroidTest
```

Internal Build Files

This section covers the files that are stored inside each of the Android projects. These paths follow the same pattern for each Android project, just the project name differs. For clarity, the root level of the project will be referred to as `root` and the app/lib level of the project will be referred to as `app`. So, for example, the path `services/services_app/build.gradle` becomes `project/app/build.gradle`.

`root/settings.gradle`

This file determines where to look for the *External Build Files*.

The `gradleConfigVersion` corresponds to a tag in the Gradle Config repository. If the local gradle files are not found, the versions of those files committed under that tag will be downloaded and used.

Before downloading those files, this file checks the local *Directory Structure* for gradle-config. If it is found, that is used. Whichever path is chosen, this linkage is established here and made available to all the rest of the gradle files.

This file also looks for library projects in the local directory structure. If they are found, they are built as dependencies. If not, their prebuilt binaries are downloaded.

`root/build.gradle`

This file establishes URLs to use for resolving dependencies. Links to each of the prebuilt binary repositories are included (demo, master, snapshot).

The dependency versions are also managed here.

root/app/build.gradle

The file contains the specific build configuration for this project. The ODK 2 projects do not differ greatly from established norms in this configuration. However, many of the constants and version numbers are stored in *variables.gradle* and variables are used here. This allows the tools to be upgraded and maintained in unison, and they can be forced to stay in sync.

This file also establishes the product flavors, signing configs, build types, and other standard options found in many Android projects. The unique aspect comes in the `dependencies` block. The different flavors have different dependencies (they will download different prebuilt binaries for their library projects). The demo and snapshot flavors build against the latest from their repositories, while the master flavor is hard coded to a specific version.

External Build Files

These build files are centralized in the Gradle Config repository. They included shared configuration, versions, and tasks.

variables.gradle

This file contains all the versions and variables strings shared among the projects. Most notably this includes the release code version, the compile targets, the **Java** version, and the composed project build and variant names.

runnables.gradle

This file contains miscellaneous Gradle tasks necessary to the ODK 2 tools. Mostly these exist to make Jenkins or Artifactory work.

uitests.gradle

This file contains tasks to make the UI tests work on a build server. In particular, they disable animations and grant external storage permissions.

remote.gradle

This file contains the paths to the remote versions of these files stored on Github or in the directory structure. This is used by *root/settings.gradle* to fetch the appropriate files.

`publish.gradle`

This file contains parameters related to the different binary publishing versions the tools use.

`jacoco.gradle`

This file contains definitions and versions for the Jacoco code coverage tool.

4.17 Contributing to ODK Docs

4.17.1 Docs Contributor Technical Guide

This document explains how to set up your computer and work locally as an ODK Docs contributor. Local set up includes installing some software, and working locally involves:

- writing documentation text or code in a code editor
- using the Terminal (the "Shell" or "Command Line")

We encourage all potential contributors to try to work locally, following this guide.

Before you begin

Learn a little about Open Data Kit

Read about the project and the community at Open Data Kit's website.

Set up collaboration accounts

Open Data Kit is a collaborative community. Before diving in as a contributor, set up accounts on our three main collaboration platforms, *GitHub*, the *ODK Developer Slack*, and the *ODK Forum*

Tip: As you are setting up your accounts, keep in mind that it is very helpful (but not required) to use the same (or similar) username on *GitHub*, the *ODK Developer Slack*, and the *ODK Forum*.

This makes it easy for other people to keep track of conversations which sometimes span multiple online platforms.

If you are willing and able to do so, a profile picture in each place is also very helpful. (But it is okay if you are unable or uncomfortable adding a picture.)

1. Set up a GitHub account.

   GitHub is a popular code storage and collaboration platform. You will need a GitHub account to contribute to Open Date Kit documentation, or any other Open Date Kit projects.

   - Open Data Kit on GitHub
   - ODK Docs on GitHub

2. Join the ODK Developer Slack.

   Slack is a popular chat platform. The Open Data Kit contributor community uses Slack to discuss development, plan work, and generally keep in touch. If you have a question about how to contribute to ODK Docs, or any other ODK project, the ODK Slack is the best place to ask it.

   Conversations related to documentation are held in the `#docs-code` channel. You may also want to check in with `#general` and `#random`.

   (a) Get an automated invitation from http://slack.opendatakit.org

   (b) Check your email for the invitation.

   (c) Follow the link in your email and set up your account.

3. Join the ODK Forum

   The Open Data Kit Forum is the main place for support questions and conversations that affect the whole ODK community (users and other stake holders, as well as contributors).

   If you have a question about how to use any ODK software, or want to get connected with the larger ODK community, the forum is the best venue for that.

   Tip: The forum has a search feature, and a long history of archived support posts. When writing new documentation about an existing feature, old forum posts are an excellent source for figuring out what people need to know: If someone has asked a question about it, it should probably be in the documentation.

Should I ask in the Forum, the Slack, or a GitHub issue?

The ODK community talks a lot, in a lot of places. Sometimes it's hard to know where to ask a question.

Contribution-related questions and problems should be asked in Slack. This includes things like:

- How do I set up my local editing environment?
- How do I use git?
- I'm having a merge conflict.
- I got an error at the terminal which I don't understand.
- How do I add a picture to a document?
- What issue should I work on?

Work-specific questions and discussion should take place on the GitHub issue defining the work. This includes things like:

- I'm writing a piece of content, but I'm not sure where it should be organized.
- I'd like to work on this feature, but I don't know how to implement it.
- Here's my idea for solving this problem. Is that a good idea?
- I'm going to be working on this for the next few days. No one else should also work on it at the same time.
- I said I was working on this, but I didn't finish and I'm no longer working on it.

User-related questions and problems should be asked in the Forum. (You should use the search feature first, since someone else may have already asked the same question.) This includes things like:

- How do I install an ODK application?
- How do I create a form?
- How do I add a specific feature to a form?
- My ODK application crashed.

But don't worry about posting a question in the wrong place.

It is better to ask a question in the "wrong" venue than to not ask the question at all. Many of the same people are present in all three places, and we will help you wherever you happen to show up.

Initial Setup

Note: Developer and authoring tools have a lot of options and alternatives. Local tools and workflows presented in this guide are what the authors feel would be easiest for newcomers and those unfamiliar with open source.

You should feel free to use your preferred tools.

Before you begin working the first time you will need to install a few tools on your computer.

You should only need to do this one time on any computer.

1. Find and open a terminal or command line.

   Windows

   Mac

   Linux

   #### Windows versions prior to Windows 10

   Use Windows PowerShell. (Not the DOS Prompt.)

   We recommend using the **`Windows PowerShell ISE`**.

   During initial setup (this section of the guide) you will need to Run as Administrator.

   Throughout the rest of the instructions in this guide, follow the instructions labeled **PowerShell** or **Windows**.

   #### Windows 10

   In Windows 10, you have a choice:

   - Use the Powershell (as described above)
   - Use the Windows Subsystem for Linux.

   If you decide to use the Powershell, follow the **Powershell** or **Windows** instructions throughout the contributor guides.

   If you decide to use the Linux subsystem, follow the **Bash** or **Windows** instructions throughout the contributor guide.

   Use the **`Terminal`** app, or another Bash-like shell.

If you've never used it before, the Terminal is probably in the *Other* directory in your App collection.

Follow the **Bash** or **Mac** instructions throughout the contributor guide.

Optional: Install Homebrew

Homebrew is a package manager for Mac OS. It makes it easier to install other apps and tools from the command line.

Follow the installation instructions.

Use a Bash-like shell of your choosing, and follow the **Bash** or **Linux** instructions throughout the contributor guide.

You will also need to be familiar with the relevant package manager for your system.

2. Install git.

   Git is a version control system. It helps us keep track of changes to the documentation. (Similar to the undo history in a document editing program.)

   Linux

   Mac

   Windows

   Use your distribution's package management system to install git on Linux.

Option 1: Download an installer

(a) Download the git installer for Mac.

(b) Open the installer package.

(c) Follow the prompts.

(d) Accept any default settings.

Option 2: Use Homebrew to install git

```
$ brew install git
```

(a) Download the git installer for Windows.

(b) Open the installer package.

(c) Follow the prompts.

(d) Accept any default settings.

3. Install Git LFS

   Git Large File Storage (Git LFS) is a tool that helps us manage images, videos, and other files which are neither text nor code.

   Linux

   Mac

   Windows

   Use your distribution's package management system to install Git LFS on Linux.

   After initial installation by the package manager, complete the install by running:

```
$ git lfs install
```

   (a) Download Git LFS from the Git LFS website.

   (b) Open the downloaded installer.

   (c) Follow the prompts.

   (d) Accept any default settings.

   (e) Open the Terminal and add LFS to git:

```
$ git lfs install
```

Option 2: Use Homebrew to install Git LFS.

```
$ brew install git-lfs
$ git lfs install
```

(a) Download Git LFS from the Git LFS website.

(b) Open the downloaded installer.

(c) Follow the prompts.

(d) Accept any default settings.

(e) Open Powershell and add LFS to git:

```
> git lfs install
```

4. Install Python 3

   Python is a programming language.

   Most of the ODK Docs tools are written in Python, so you need it installed on your computer in order to use those tools. (Don't worry. You don't need to know how to program in Python.)

   We require Python 3, version 3.6 or later.

   Linux

   Mac

   Windows

   Use your distribution's package management system to install Python 3.6+ on Linux.

   (For more help, see Installing Python on Linux.)

Tip: Mac OS includes a legacy (outdated) version of Python. It's best to just ignore it.

Option 1: Use the Python Installer for Mac

(a) Download the latest Python installer for Mac.

64-bit or 32-bit?

Python provides 64-bit and 32-bit installers. You probably need the 64-bit installer.

If you are running a relatively recent Mac OS update (Mountain Lion or later — any Mac from the last several years) the 64-bit installer is for you.

If you have an older Mac, and are unsure if it can run a 64-bit installer, check the processor details in -> *About This Mac*.

(b) Open the Installer.

(c) Follow the prompts.

(d) Accept the default settings.

(e) Open the Terminal to see if Python installed properly.

```
$ python3 --version
Python 3.7.0
```

The output from **`python3 --version`** might be a little different, but it should be higher than `3.6`.

If you get an error here, something went wrong. Try running the installer again. If the problem persists, and you can't debug it yourself, asks us about it on ODK Slack.

Option 2: Use Homebrew to install Python 3.6+

```
$ brew install python
.
.
.
$ python3 --version
Python 3.7.0
```

The output from **`python3 --version`** might be a little different, but it should be higher than `3.6`.

If you get an error here, something went wrong. Try running **`brew install python`** again. If the problem persists, and you can't debug it yourself, asks us about it on ODK Slack.

(a) Go to the Python Releases for Windows page.

(b) Under the latest numbered release for Python 3, find and download the **`Windows x86-64 web-based installer`** (for a 64-bit system) or the **`Windows x86 web-based installer`** (for a 32-bit system).

64-bit or 32-bit?

Well over 90% of computers running Windows are 64-bit. So you probably need the 64-bit version.

If you are running a very old or low-powered computer, and you are unsure if it is 64-bit or 32-bit, you can use *this guide from HP* (which will work for other computer brands) to find that information.

(c) Open the downloaded installer.

(d) Follow the prompts.

(e) Accept all default settings.

(f) Open Powershell and make sure the installation completed.

```
> python --version
Python 3.7.0
```

The output from **`python --version`** might be a little different, but it should be whatever numbered version you downloaded.

If you get an error here, something went wrong. Try running the installer again. If the problem persists, and you can't debug it yourself, asks us about it on ODK Slack.

5. Set up your working directory

In whatever directory (folder) on your computer where you organize projects, create a new directory for Open Data Kit, and then navigate to that directory. (We recommend calling this directory `odk`, and the rest of the guide will assume that's what you called it.)

Bash

PowerShell

```
$ mkdir odk
$ cd odk
/odk/ $
```

```
> mkdir odk
> cd odk
/odk/ >
```

For the rest of this guide, we assume you are in the `/odk/` directory, or a subdirectory of it.

6. Set up a virtual environment

A virtual environment is a Python construct that lets you download and install tools for a specific project without installing them for your entire computer.

(a) Create the virtual environment.

Bash

PowerShell

```
/odk/ $ python3 -m venv odkenv
```

```
/odk/ > python -m venv odkenv
```

(b) Activate the virtual environment.

Bash

PowerShell

```
/odk/ $ source odkenv/bin/activate
(odkenv) /odk/ $
```

```
/odk/ > source odkenv/bin/activate
(odkenv) /odk/ >
```

The `(odkenv)` before the prompt shows that the virtual environment is active. You will need to have this active any time you are working on the docs.

Later, to deactivate the virtual environment:

Bash

PowerShell

```
(odkenv) /odk/ $ deactivate
/odk/ $
```

```
(odkenv) /odk/ > deactivate
/odk/ >
```

7. Fork the ODK Docs repository to your own GitHub account.

A *repository* (*repo*) is a store of all the code and text for a project. The ODK Docs repo is kept at GitHub.

On GitHub, a *fork* is a copy of a repo, cloned from one user to another. In order to work on ODK Docs, you will create your own fork.

(a) Go to the ODK Docs repo on GitHub.

(b) Use the *Fork* button (top right) to create your own copy.

(c) After the process completes, you'll be looking at your own fork on GitHub.

8. Clone down your copy to your local computer

   (a) From your own fork of the repo on GitHub, select the *Clone or download* button.

   (b) Copy the URI from the text box that opens. It will be something like: `https://github.com/your-gh-username/docs.git`

   (c) Use your terminal to clone the repository.

   You should already be in the `odk` directory, with the virtual environment active.

   Bash

   Powershell

```
(odkenv) /odk/ $ git clone https://github.com/your-github-username/docs.
↪git
.
.
.
(odkenv) /odk/ $ cd docs
(odkenv) /odk/docs/ $
```

```
(odkenv) /odk/ > git clone https://github.com/your-github-username/docs.
↪git
.
.
.
(odkenv) /odk/ > cd docs
(odkenv) /odk/docs/ >
```

Warning: Some of the git commands produce meaningless errors in PowerShell. If you get an error when using git, but everything seems to work otherwise, ignore the error.

Note: This will cause your computer to download the entire ODK Docs repository, including a large number of images. It will take several minutes to complete.

Your local directory

If you followed the instructions, you should now have the following directory structure:

- `odk`

- `docs`
- `odkenv`

The `odkenv` directory stores your virtual environment, and you should not need to open it or directly view its content. Just ignore it.

The `docs` directory is your copy of the ODK Docs repo. You will do most of your work in this directory.

If you need to download or create additional files which are not actually a part of the ODK Docs repository, keep them out of the `docs` directory.

You can use the main `odk` directory for any other files you need to work on. (For example, you may want to create a directory called `odk/forms` to hold XLSForm and XForm files.)

9. Set the upstream remote

   In git, a *remote* is a copy of a repo somewhere else. From your local computer's point of view, your online copy at GitHub is a remote.

   When you cloned down a repo, your local copy gives your GitHub copy the name `origin`.

   You also need to give the primary ODK Docs repo a name, and our convention is to name it `upstream`.

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ git remote add upstream https://github.com/
↪opendatakit/docs.git
(odkenv) /odk/docs/ $ git remote -v
origin https://github.com/your-github-username/docs.git (fetch)
origin https://github.com/your-github-username/docs.git (push)
upstream https://github.com/opendatakit/docs.git (fetch)
upstream https://github.com/opendatakit/docs.git (push)
```

```
(odkenv) /odk/docs/ > git remote add upstream https://github.com/
↪opendatakit/docs.git
(odkenv) /odk/docs/ > git remote -v
origin https://github.com/your-github-username/docs.git (fetch)
origin https://github.com/your-github-username/docs.git (push)
upstream https://github.com/opendatakit/docs.git (fetch)
upstream https://github.com/opendatakit/docs.git (push)
```

   If everything went right, you should see output similar to what is shown above.

10. Install Python tools with pip

    Pip is a package management tool that comes with Python. We use it to download and install our documentation tools. These Python tools are listed in `requirements.txt`.

    Bash

    PowerShell

```
(odkenv) /odk/docs/ $ pip install --upgrade pip
(odkenv) /odk/docs/ $ pip install -r requirements.txt
```

```
(odkenv) /odk/docs/ > pip install --upgrade pip
(odkenv) /odk/docs/ > pip install -r requirements.txt
```

    The first command upgrades pip itself to the latest version. Then second checks `requirements.txt` and installs everything listed in it. This will take several moments.

    Note: If you are ever running one of the build commands shown below and it fails with a message that includes `ModuleNotFoundError`, there might be changes to `requirements.txt` since you originally ran **`pip install -r requirement.txt`**. Run the installation again and then retry your build.

11. Choose a text/code editor

    The documentation source files are written in a plain text format called reStructuredText. This means special formatting (bullets, headers, bold text) is represented by visible characters, not hidden behind a graphical display. When working on a documentation file, you see and write something that looks like:

```
#. Choose a text/code editor

   The documentation source files
   are written in a plain text format called `reStructuredText`_.

   .. _reStructuredText: http://docutils.sourceforge.net/docs/user/rst/
↪quickref.html
```

    You cannot write and edit these files in a typical document preparation program like **`MS Word`** or **`Google Docs`**. Instead, you need a coding editor.

    There are a lot of editors, and people who use them often have very strong opinions about them. You are free to choose any editor you like.

    If you've never used an editor before, you might want to start with one of the easier and more popular ones:

    - Atom

- Sublime
- VS Code
- Notebook++ (Windows only)

Most of these have plugins that will make writing reStructuredText easier by color-coding the markup.

This completes the setup of your local working environment. Take a break before diving into how you actually work.

Working on the docs

1. Find an issue to work on.

   Work on ODK Docs is planned using the GitHub repository's issue tracker.

   (a) Browse the issue tracker and find one you may want to work on.

   (b) Make sure you understand the goal of the project. If the goal isn't clear, ask. If there is anything in the issue that doesn't make sense, ask about it. Feel free to make suggestions about how something could be accomplished.

   (c) If you decide to work on an issue, assign yourself to it by writing **@opendatakit-bot claim** in a comment.

   (d) If the issue requires a novel or creative solution not defined in the issue already (we've stated a problem and you think you know a way to fix it) write a comment describing your plan. It is a good idea to get feedback on an idea before working on it. Often, other contributors can provide additional context about why a particular solution may or may not work.

   Your first issue

   The very first issue you should work on as a new ODK Docs contributor is Issue 96 — Line Edits. The issue is very simple:

   (a) Find a typo.

   (b) Fix the typo.

   This will help you get used to working with the documentation tools, and helps us get rid of the inevitable errors that creep in to our writing.

2. Make sure you are on the master branch

   A branch is a named sequence of changes representing work on the repo. For example, if you were going to work on Issue 96 — Line Edits, you would create a new branch

called `line-edits` to hold that work. When you were done, you would merge those changes back to the main branch, which we call `master`.

The first time you clone the docs repo and start working, you will be on the *master* branch.

Each time you come back to starting work on a new issue, make sure you are on the `master` branch before continuing.

(a) Check the current branch with **`git branch`**. This will output a list of branches, with a star next to the current one.

Bash

PowerShell

```
(odkenv) /odk/docs/ $ git branch
   branch-name
   branch-name
   branch-name
 * master
   branch-name
```

```
(odkenv) /odk/docs/ > git branch
   branch-name
   branch-name
   branch-name
 * master
   branch-name
```

(b) If you are not on master, switch to master with **`git checkout`**.

Bash

PowerShell

```
(odkenv) /odk/docs/ $  git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

```
(odkenv) /odk/docs/ >  git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

3. Pull in changes from upstream

Other people are constantly making changes to the docs, so you need to keep your local copy up to date.

Before you start working, use **`git pull`** to pull in the changes from the upstream

repository's master branch. Then, just to be sure, you can use **`git status`** to make sure everything is up to date.

Bash

PowerShell

```
(odkenv) /odk/docs/ $ git pull upstream master
(odkenv) /odk/docs/ $ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

```
(odkenv) /odk/docs/ > git pull upstream master
(odkenv) /odk/docs/ > git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

> **Warning:** Some git commands (including **`git pull`** and **`git checkout`**) send error messages to PowerShell even when they work correctly. If everything seems to be working, you can ignore these.

4. Create a new branch for your work.

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ git checkout -b branch-name
Switched to a new branch 'branch-name'
```

```
(odkenv) /odk/docs/ > git checkout -b branch-name
Switched to a new branch 'branch-name'
```

   Branch names should be short, lowercase, and use hyphens as separators. They do not need to carry a lot of information (like your name or the date).

   Good branch names:

   - `getting-started-guide`
   - `contributing`
   - `fix-issue-13`

Bad branch names:

- `getting started guide`
- `Getting started guide`
- `Getting_started_guide`
- `writing-the-getting-started-guide-adammichaelwood-july-2017-draft`

5. Work on the documentation

   Finally, you can open an *editor of your choice* and work on the documentation.

   The source files for documentation text are in these directories:

   `odk1-src` Files for the pages at http://docs.opendatakit.com

   `odk2-src` Files for the pages at http://docs.opendatakit.com/odk2

   `shared-src` Files for pages shared by both ODK1 and ODK2 docs. (This page and the other contributor guide pages.)

   If you're going to write or edit documentation text, please read:

   - *Docs Markup and Syntax Guide*
   - *Docs Style Guide*

   If you're working on code or deployment, please read:

   - *Docs Developer Guide*

6. Local checks

   Once you have worked on the documentation, we want to make sure your contribution will get accepted and published right away.

   To ensure your changes will pass all the deployment tests, you should run the tests locally first and correct any problems.

   (a) Spell check

   If you've been working on files in `odk1-src` or `shared-src`:

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ make odk1-spell-check
```

```
(odkenv) /odk/docs/ > rm -r -fo tmp1-src
(odkenv) /odk/docs/ > rm -r -fo odk1-build
(odkenv) /odk/docs/ > Copy-Item odk1-src -Destination tmp1-src -Recurse
(odkenv) /odk/docs/ > Copy-Item shared-src -Destination tmp1-src -
↪Recurse
```

```
(odkenv) /odk/docs/ > sphinx-build -b spelling tmp1-src odk1-build/
↪spelling
(odkenv) /odk/docs/ > python util/check-spelling-output.py odk1-build
```

If you've been working on files in `odk2-src`:

Bash

PowerShell

```
(odkenv) /odk/docs/ $ make odk2-spell-check
```

```
(odkenv) /odk/docs/ > rm -r -fo tmp2-src
(odkenv) /odk/docs/ > rm -r -fo odk2-build
(odkenv) /odk/docs/ > Copy-Item odk2-src -Destination tmp2-src -Recurse
(odkenv) /odk/docs/ > Copy-Item shared-src -Destination tmp2-src -
↪Recurse
(odkenv) /odk/docs/ > sphinx-build -b spelling tmp1-src odk2-build/
↪spelling
(odkenv) /odk/docs/ > python util/check-spelling-output.py odk2-build
```

This will send some output to the terminal, which will include mentions of any words not in the dictionary.

- If the flagged words are really misspellings, correct them.
- If the flagged words are not misspelled, and *should* be in the dictionary add them to `spelling_wordlist.txt`.
- If the flagged words are not misspelled, but *should not* be in the dictionary (for example, they are non-words that make sense on a single page for a specific reason) add them at the top of the file in which they are being used, before the title heading:

```
.. spelling::

   abc
   def
   exe
   functool

This Is The Page Title
======================
```

When adding new words to `spelling_wordlist.txt` or the top of a document file, please keep the words in alphabetical order.

(b) Style check

7. Build and check

   We use a Python tool called Sphinx to compile all the `.rst` files into a working website.

   If you've been working on files in `odk1-src` or `shared-src`:

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ make odk1
```

```
(odkenv) /odk/docs/ > rm -r -fo tmp1-src
(odkenv) /odk/docs/ > rm -r -fo odk1-build
(odkenv) /odk/docs/ > Copy-Item odk1-src -Destination tmp1-src -Recurse
(odkenv) /odk/docs/ > Copy-Item shared-src -Destination tmp1-src -Recurse
(odkenv) /odk/docs/ > sphinx-build -b dirhtml tmp1-src odk1-build
```

   If you've been working on files in `odk2-src`:

   Bash

   PowerShell

```
make odk2
```

```
(odkenv) /odk/docs/ > rm -r -fo tmp2-src
(odkenv) /odk/docs/ > rm -r -fo odk2-build
(odkenv) /odk/docs/ > Copy-Item odk2-src -Destination tmp2-src -Recurse
(odkenv) /odk/docs/ > Copy-Item shared-src -Destination tmp2-src -Recurse
(odkenv) /odk/docs/ > sphinx-build -b dirhtml tmp2-src odk2-build
```

   This generates a lot of output. Near the end of the output you may see a statement like:

```
build succeeded, 18 warnings.
```

   Those warnings are problems with the text which you need to fix before submitting your changes. Scroll up in the terminal to find each warning, so that you can address it in the source files.

   A Sphinx warning looks like this:

```
/path/to/file-name.rst:LINENUMBER: WARNING: warning message

short excerpt from the file
```

   This tells you what file the problem is in, the approximate line number, and the nature of the problem. Usually that is enough to fix it. If you can not figure out the meaning of a particular warning, you can always ask about it on the ODK Slack.

Note: Because of a bug in Sphinx the line numbers in error and warning messages will be off by about 15 lines (the length of `rst_prolog` in `conf.py`).

As you fix each warning, run the build again to see if it disappears from the output.

Note: The warning messages will refer to the file name using the temporary directory path `tmp1-src` or `tmp2-src`. You need to correct the problems in the real source directory (`odk1-src`, `odk2-src`, or `shared-src`).

When you just can't fix the error…

If you've done your best and asked on the ODK Slack, and you still cannot correct the warning, stop worrying about it and skip to the next step. When you submit your changes on GitHub, include a note about the warning. Other contributors will help solve the problem before merging.

Once you've corrected all the warnings that can be corrected…

8. Serve the documentation website locally and view it.

   If you've been working on files in `odk1-src` or `shared-src`:

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ python -m http.server -d odk-build 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

```
(odkenv) /odk/docs/ > python -m http.server -d odk1-build 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

   If you've been working on files in `odk2-src`:

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ python -m http.server -d odk2-build 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

```
(odkenv) /odk/docs/ > python -m http.server -d odk2-build 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

   (a) Open your browser and go to http://localhost:8000.

(b) Read through your doc edits in the browser.

(c) Go back to the source files to correct any errors you find.

(d) Go to your terminal, and press `CTRL C` to shut down the local web server.

(e) Re-run the build and serve steps.

(f) Continue proofreading.

Once you are reasonably sure your changes are ready…

9. Commit your changes to your local repository.

   A *commit* is snapshot of your working files in a particular state, along with a record of all the changes that led up to that state. That snapshot is what you will submit to the main repository.

Note: We explain how to do a commit at this step because you need to do it before you can submit your changes. However, you don't have to wait until you are done to commit. You can commit as many times as you like while working.

This can be especially helpful if you are working on a complicated set of changes, over several working sessions.

(a) Stage the files for commit with **`git add`**.

To stage all changes for commit:

Bash

PowerShell

```
(odkenv) /odk/docs/ $ git add -A
```

```
(odkenv) /odk/docs/ > git add -A
```

(b) Commit the staged files with **`git commit`**.

Bash

PowerShell

```
(odkenv) /odk/docs/ $ git commit -m "Write a commit message here."
```

```
(odkenv) /odk/docs/ > git commit -m "Write a commit message here."
```

Your commit message needs to be wrapped in quote marks. It should, in a sentence or less, explain your work.

10. Push your committed changes to your GitHub repo with **`git push`**.

    Bash

    PowerShell

```
(odkenv) /odk/docs/ $ git push origin branch-name
```

```
(odkenv) /odk/docs/ > git push origin branch-name
```

> **Warning:** The **`git push`** command produces meaningless errors in PowerShell. If you get an error when using **`git push`**, but everything seems to work otherwise, ignore the error.

Tip: You may be prompted to enter your GitHub username and password. When entering your password, the cursor won't move — it will look like you aren't entering anything, even though you are.

To avoid having to retype these every time, you can store your GitHub credentials locally.

---

11. Issue a pull request from your GitHub repo to the main ODK Docs repo.

    A *pull request* (or PR) is a request from you to the ODK Docs maintainers to pull in your changes to the main repo.

    (a) Go to the ODK Docs repo on GitHub. (Make sure you are logged in.)

    (b) Find the message near the top of the page that mentions your recent pushed branches. Select *Compare & pull request* to start a pull request.

    (c) Follow GitHub's instructions to start the pull request.

    These details should fill-in automatically, but be sure to double-check them:

    - *Base fork* should be the main repo (`opendatakit/docs`).
    - *base* should be `master`.
    - Your repo and working branch name should be listed beside them.

    You will see either a green **Able to be merged** message or a message informing that the branch can not be merged. You can proceed in either case. If the branch cannot be merged, the maintainers will work with you to resolve the problem.

    (d) Write a PR message explaining your work.

The PR message field includes a template to remind you of what to include. Fill in the template and delete any sections which are not applicable.

A good PR message includes:

- The issue number you are working on. (Write `closes #123` if the PR completes the work for the issue. If there's still work to do, write `addresses #123`.)
- A summary of what you did.
- Details of work that still needs to be done.
- Details of new work created or implied by this PR.
- Details of any unresolved errors or warnings, including details of what you tried.
- Justification for any changes to `requirements.txt`.
- Details of any difficulties, questions, or concerns that came up while working on this issue.

(e) Submit your pull request.

The maintainers and other contributors will review your PR as quickly as possible. They may request changes to your work. If changes are needed:

(a) **Don't worry.** Revision is a normal part of technical writing, and everyone (even the project's founders and leaders) has their work reviewed and are frequently asked to revise it.

(b) Work on the files again locally. (Use **`git branch`** to make sure you are still in the same working branch.)

(c) *Stage and commit* your changes locally again (**`git add -A`**; **`git commit -m "Commit message"`**).

(d) *Push your commit* (**`git push origin branch-name`**).

(e) Your new commits will automatically update the PR. Do not start a new PR.

Once everything has been approved, the changes will be merged in and will appear on *this website*. At that point… congratulations! You are now a contributor to Open Data Kit.

The next time you work

We hope that contributing to ODK Docs is a rewarding experience and that you'll want to keep going. Each time you start work on a new issue the process is the same as outline above.

Here are a few things to keep in mind when you start your next contribution.

1. Return to `master` with **`git checkout master`**.

   New work is done on new branches which are started from master. So, before you start a new branch, return to the master branch.

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ git checkout master
```

```
(odkenv) /odk/docs/ > git checkout master
```

2. Pull in changes with **`git pull upstream master`**.

   You need to start your new work from the latest version of everyone else's work.

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ git pull upstream master
```

```
(odkenv) /odk/docs/ > git pull upstream master
```

3. Update the master branch of your online GitHub repository.

   Bash

   PowerShell

```
(odkenv) /odk/docs/ $ git push origin master
```

```
(odkenv) /odk/docs/ > git push origin master
```

4. Find a new issue to work on.
5. Start a new branch for your work with **`git checkout -b branch-name`**.

Keep improving

As you are getting comfortable with the contribution process, take a few minutes to read our *Tips for Making Good Contributions*. You may also want to dig deeper into the *Docs Style Guide* and the *Docs Markup and Syntax Guide*. (And if you are writing code, check out the *Docs Developer Guide*.)

And don't forget to join us on the ODK Slack.

Open Data Kit is a community, and we depend on each other's work. Thank you for your contribution to ODK Docs and your presence in this community.

4.17.2 Docs Markup and Syntax Guide

The ODK documentation is built using Sphinx, a static-site generator designed to create structured, semantic, and internally consistent documentation. Source documents are written in reStructuredText, a semantic, extensible markup syntax similar to Markdown.

- reStructuredText Primer — Introduction to reStructuredText
  - reStructuredText Quick Reference
  - reStructuredText 1-page cheat sheet
- Sphinx Markup — Detailed guide to Sphinx's markup concepts and reStructuredText extensions

Note: Sphinx and reStructuredText can be very flexible. For the sake of consistency and maintainability, this guide is *highly opinionated* about how documentation source files are organized and marked up.

Indentation

Indentation is meaningful in Sphinx and reStructuredText text. Usually, indenting a section means that is "belongs to" the line it is indented under. For example:

```
.. figure:: path-to-image.*

  This is the caption of the figure. Notice that it is indented under the line␣
↪defining the figure.
```

The rules for indentation are:

- Use **spaces, not tabs**.
- Generally, indent **two spaces**.

The exception to the two spaces rule is *Ordered (numbered) lists*, where indentation follows the content of the list item.

```
1. This is a list item.

   This is some additional content related to Item 1. Notice that it is indented␣
↪to the same column as the first line of content. In this case, that's three␣
↪(3) spaces.
```

```
.
.
.

10. The tenth item in a list.

    This related content will be indented four spaces.
```

Documentation Files

Sphinx document files have the `.rst` extension. File names should be all lowercase and use hyphens (not underscores or spaces) as word separators.

Normally, the title of the page should be the first line of the file, underlined with equal-signs.

```
Title of Page
================

Page content is here...
```

You can alternatively wrap the title in two lines of asterisks, in some cases. (This should not be your default choice.)

```
*******************
Title of Page
*******************

Page content here.
```

The asterisks style is useful when you are combining several existing documents and don't want to change every subsection headline. Or, you can use it when you are working on a document that you have reason to think might be split into separate documents in the future.

Important: If you use the double-asterisks style, your major section headlines (*<h2>*) should use the equal-signs underline style. This allows major sections to be easily promoted to individual pages.

See *Sections and Titles* for more details.

Tables of Content

The `toctree` directive defines a table of content. The content of a `toctree` is a list of page file names, without the `.rst` extension. When rendered, the `toctree` becomes an unordered list of page links, including links to sections and subsections of the included pages.

```
.. toctree::

  page-name
  another-page
  this-other-page
```

The depth of section and subsection links to display in the output can be controlled using the `maxdepth` attribute. We typically use a depth of `2`, but you should use your judgment if you feel it should be more or less in any given context.

```
.. toctree::
  :maxdepth: 2

  this-page
  that-page
  thick-page
  flat-page
```

See also:

The TOC Tree

> The Sphinx documentation includes information about a number of other `toctree` attributes.

Sidebar navigation menu

The `index.rst` file serves as a front-page to the documentation and contains the main tables of content, defined using `toctree` directives.

These `toctree` directives control the sidebar navigation menu. To add a new document to a table of content, add the file name (without the `.rst` extension) to the relevant list of file names in `index.rst`.

Secondary tables of content

Collections of documents are sometimes given their own table of content on an individual page. (See, for example, Setting Up ODK Collect and Using ODK Collect.)

In these cases, the page containing the `toctree` serves as a sort of intro page for the collection. That intro must, itself, be included in the *Sidebar navigation menu*.

The contents of a `toctree` appear as section links in another `toctree` it is included in. That is, if a `toctree` in `index.rst` lists `collect-using`, and `collect-using.rst` has a `toctree`, then the contents of that second `toctree` will appear in the *Sidebar navigation menu*, as sub-items to Using ODK Collect. (Indeed, this is precisely the case in the docs currently.)

How ODK Docs uses main and secondary tables of content

- Major topics get a `toctree` in `index.rst`

  Major topics include things like:

  - Each major product (Collect, Aggregate, Briefcase)
  - Large, general categories like Contributing

  Major topic tables of content include both sub-collection intro pages and also individual pages that don't fit into a sub-collection.

  The `caption` attribute of the `toctree` directive defines the section label in the *Sidebar navigation menu*.

- Within a large topic, documents are grouped into collections of related pages, defined by a `toctree` on a topic intro page.

  Intro pages (pages that contain secondary `toctree` directives) may include additional content, introducing the collection or providing contextual wayfinding. However, this is not always necessary or desirable. Use your judgment, and avoid stating things just for the sake of having some text. ("Here are the pages in this collection.")

  We also (very occasionally) include `toctree` directives in sub-collection pages.

Tip: If it not obvious where a new document should appear in the navigation, the best practice is to simply ask about it in the GitHub issue driving the new page.

Note: For wayfinding purposes, we sometimes create an *Unordered (bullet) lists* of page links rather than a `toctree` directive. (For example, see `collect-intro`. We do this when using a `toctree` would create redundant links in the *Sidebar navigation menu*.

Why are the docs files not grouped into folders in the source?

We use `toctree` directives as our primary way of organizing the documentation for readers. We do not organize the source `rst` files into subfolders.

The reason is that if we put them into topic-related subfolders, it would affect the URI of the document. Keeping all of our document files in a single flat directory results in a flat URI structure. Every page's URI looks like `docs.opendatakit.org/page-name`.

If we used subdirectories, then our URIs would look like `docs.opendatakit.org/subdirectory-name/page-name`. This would mean that our URIs would change every time we moved a document from one folder to another, greatly increasing the time cost and broken-link risk of reorganizing the docs.

Sections and Titles

Headlines require two lines:

- the text of the headline, followed by
- a line filled with a single character.

Each level in a headline hierarchy uses a different character:

```
Title of the Page - <h1> - Equal Signs
======================================


Major Section - <h2> - Hyphens
-------------------------------


Subsection - <h3> - Tildes
~~~~~~~~~~~~~~~~~~~~~~~~~~~


Sub-subsection - <h4> - Double Quotes
"""""""""""""""""""""""""""""""""""""


Sub-sub-subsection - <h5> - Single Quotes
'''''''''''''''''''''''''''''''''''''''''''
```

If you need to combine several existing pages together, or want to start a single-page doc that you think might be split into individual pages later on, you can add a top-level title, demoting the other headline types by one.

```
***********************************************
Page Title - <h1> - Asterisks above and below
***********************************************
```

```
Major Section - <h2> - Equal Signs
==========================================


Subsection - <h3> - Hyphens
---------------------------------


Sub-subsection - <h4> - Tildes
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


Sub-sub-subsection - <h5> - Double Quotes
"""""""""""""""""""""""""""""""""""""""""""""""

Sub-sub-sub-subsection - <h6> - Single Quotes
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''
```

In either case, the underline of characters needs to be *longer than* the line of text. In the case of the asterisks, the two lines of asterisks need to be the same length.

Note: The exact order of underline characters is flexible in reStructuredText. However, this specific ordering should be used throughout the ODK documentation.

Section labels

In order to facilitate efficient *Cross referencing*, sections should be labeled. This is done on the line above the section title. The format is:

- two dots
- underscore
- section label
  - lowercase
  - hyphen separators
- a single colon

```
.. _section-label:

Section Title
----------------
```

```
Lorem ipsum content of section blah blah.
```

The section label is usually a slugified version of the section title.

Section titles must be unique throughout the entire documentation set. Therefore, if you write a common title that might appear in more than one document (*Learn More* or *Getting Started*, for example), you'll need to include additional words to make the label unique. The best way to do this is to add a meaningful work from the document title.

```
ODK Aggregate
===============

ODK Aggregate is a server application...

.. _aggregate-getting-started:

Getting Started
-----------------
```

Basic Markup

Escaping characters

Markup characters can be escaped using the \ character.

```
*Italic.*

\*Not italic, surrounded by asterisks.\*
```

Italic.

Not italic, surrounded by asterisks.

Emphasis and Inline Literal

```
Single asterisks for *italic text* (``<em>``).

Double asterisks for **bold text** (``<strong>``).

Double back-ticks for ``inline literal text`` (``<code>``).
```

Single asterisks for *italic text* (`<em>`).

Double asterisks for **bold text** (`<strong>`).

Double back-ticks for `inline literal text` (`<code>`).

Note: The **bold**, *italic*, and `inline literal` styles do not carry semantic meaning. They should not be used when a more semantically appropriate markup construct is available; for example, when *writing about GUI text*.

Hyperlinks

External hyperlinks — that is, links to resources *outside* the documentation — look like this:

```
This is a link to `example <http://example.com>`_.
```

This is a link to example.

You can also use "reference style" links:

```
This is a link to `example`_.

.. _example: http://example.com
```

This may help make paragraphs with *a lot* of links more readable. In general, the inline style is preferable. If you use the reference style, be sure to keep the link references below the paragraph where they appear.

```
You can also simply place an unadorned URI in the text: http://example.com
```

You can also simply place an unadorned URI in the text: http://example.com

Lists

Unordered (bullet) lists

```
Bulleted lists ( ``<ul>`` ):

- use hyphens
- are unindented at the first level
- must have a blank line before and after

  - the blank line requirement means that nested list items will have a blank␣
↪line before and after as well

  - you may *optionally* put a blank line *between* list items
```

Bulleted lists (`<ul>`):

- use hyphens
- are unindented at the first level
- must have a blank line before and after
  - the blank line requirement means that nested list items will have a blank line before and after as well
  - you may *optionally* put a blank line *between* list items

Ordered (numbered) lists

```
Numbered lists ( ``<ol>`` ):

1. Start each line with a number and period
2. Can begin on any number
3. Must have a blank line before and after
4. Can have nested sub-lists

   a. nested lists are numbered separately
   b. nested lists need a blank line before and after

#. Can have automatic number with the ``#`` character.
```

Numbered lists (`<ol>`):

1. Start each line with a number and period
2. Can begin on any number
3. Must have a blank line before and after
4. Can have nested sub-lists
   (a) nested lists are numbered separately
   (b) nested lists need a blank line before and after
5. Can have an automatic number with the `#` character.

Note: See *Ordered and unordered lists* in the *Docs Style Guide* for details on when to use ordered and unordered lists.

Definition Lists

```
Definition list ( ``<dl>`` )
  a list with several term-definition pairs

Terms
  should not be indented

Definitions
  should be indented under the term

Line spacing
  there should be a blank line between term-definition pairs
```

Definition list (`<dl>`) a list with several term-definition pairs

Terms should not be indented

Definitions should be indented under the term

Line spacing there should be a blank line between term-definition pairs

Paragraph-level Markup

```
Paragraphs are separated by blank lines. Line breaks in the source code do not␣
↪create line breaks in the output.

This means that you *could*, in theory,
include a lot of arbitrary line breaks
in your source document files.
These line breaks would not appear in the output.
Some people like to do this because they have been trained
to not exceed 80 column lines, and they like
to write .txt files this way.
Please do not do this.

There is **no reason** to put a limit on line length in source files for␣
↪documentation, since this is prose and not code. Therefore, please do not put␣
↪arbitrary line breaks in your files.
```

Paragraphs are separated by blank lines. Line breaks in the source code do not create line breaks in the output.

This means that you *could*, in theory, include a lot of arbitrary line breaks in your source document files. These line breaks would not appear in the output. Some people like to do this because they have been trained to not exceed 80 column lines, and they like to write .txt files this way. Please do not do this.

There is **no reason** to put a limit on line length in source files for documentation, since this is prose and not code. Therefore, please do not put arbitrary line breaks in your files.

Block Quotes

```
This is not a block quote. Block quotes are indented, and otherwise unadorned.

  This is a block quote.
  - Adam Michael Wood
```

This is not a block quote. Block quotes are indented, and otherwise unadorned.

> This is a block quote. — Adam Michael Wood

Line Blocks

```
| Line blocks are useful for addresses,
| verse, and adornment-free lists.
|
| Each new line begins with a
| vertical bar ("|").
|     Line breaks and initial indents
|     are preserved.
```

Line blocks are useful for addresses,
verse, and adornment-free lists.

Each new line begins with a
vertical bar (”|”).
    Line breaks and initial indents
    are preserved.

Tables

Grid style

```
+------------+------------+-----------+
| Header 1   | Header 2   | Header 3  |
+============+============+===========+
```

```
| body row 1 | column 2   | column 3  |
+------------+------------+-----------+
| body row 2 | Cells may span columns.|
+------------+------------+-----------+
| body row 3 | Cells may  | - Cells   |
+------------+ span rows. | - contain |
| body row 4 |            | - blocks. |
+------------+------------+-----------+
```

Header 1	Header 2	Header 3
body row 1	column 2	column 3
body row 2	Cells may span columns.	
body row 3	Cells may span rows.	• Cells • contain • blocks.
body row 4		

Simple style

```
=====  =====  ======
   Inputs     Output
------------  ------
  A      B    A or B
=====  =====  ======
False  False  False
True   False  True
False  True   True
True   True   True
=====  =====  ======
```

Inputs		Output
A	B	A or B
False	False	False
True	False	True
False	True	True
True	True	True

CSV Table

The `csv-table` role is used to create a table from CSV (comma-separated values) data. CSV is a common data format generated by spreadsheet applications and commercial databases. The data may be internal (an integral part of the document) or external (a separate file).

```
.. csv-table:: Example Table
 :header: "Treat", "Quantity", "Description"
 :widths: 15, 10, 30

 "Albatross", 2.99, "On a stick!"
 "Crunchy Frog", 1.49, "If we took the bones out, it wouldn't be
 crunchy, now would it?"
 "Gannet Ripple", 1.99, "On a stick!"
```

Table 4.55: Example Table

Treat	Quantity	Description
Albatross	2.99	On a stick!
Crunchy Frog	1.49	If we took the bones out, it wouldn't be crunchy, now would it?
Gannet Ripple	1.99	On a stick!

Some of the options recognized are:

:widths:

Contains a comma or space-separated list of relative column widths. The default is equal-width columns.

The special value `auto` may be used by writers to decide whether to delegate the determination of column widths to the backend.

In most cases, the best result is either the default or `auto`. If you're unsure, try it both ways and see which looks better to you.

:header:

Contains column titles. It must use the same CSV format as the main CSV data.

:delim:

Contains a one character string used to separate fields. Default value is comma. It must be a single character or Unicode code.

The only reason to use something other than a comma is when copying large blocks of content from another source that uses a different style. If you are creating new table content yourself, use the comma.

```
.. csv-table:: Table using # as delimiter
  :header: "Name", "Grade"
  :widths: auto
  :delim: #

  "Peter"#"A"
  "Paul"#"B"
```

```
.. csv-table:: Table using | as delimiter
  :header: "Name", "Grade"
  :widths: auto
  :delim: |

  "Peter"|"A"
  "Paul"|"B"
```

:align:

It specifies the horizontal alignment of the table. It can be *left*, *right* or *center*.

```
.. csv-table:: Table aligned to right
  :header: "Name", "Grade"
  :align: right

  "Peter", "A"
  "Paul", "B"
```

Table 4.56: Table aligned to right

Name	Grade
Peter	A
Paul	B

:file:

Contains the local file system path to a CSV data file.

:url:

Contains an Internet URL reference to a CSV data file.

Note:

- There is no support for checking that the number of columns in each row is the same. However, this directive supports CSV generators that do not insert "empty" entries at the end of short rows, by automatically adding empty entries.

```
.. csv-table:: Table with different number of columns in each row
   :header: "Name", "Grade"

   "Peter"
   "Paul", "B"
```

Table 4.57: Table with different number of columns in each row

Name	Grade
Peter	
Paul	B

- Whitespace delimiters are supported only for external CSV files.

For more details, refer to this guide on CSV Tables.

Note: In almost all cases, `csv-table` is the easiest and most maintainable way to insert a table into a document. It should be preferred unless there is a compelling reason to use one of the other styles.

Sphinx-specific Markup

Roles and directives

A *role* is an inline markup construct that wraps some text, similar to an HTML or XML tag. They look like this:

```
:rolename:`some text`
```

A directive is a block-level markup construct. They look like this:

```
.. directivename:: additional info or options here
  :option: optional-value
  :option: optional-value

  Content of block here, indented.

This is no longer part of the block controlled by the directive.
```

Most of the Sphinx-specific and ODK-specific markup will use one or both of these constructs.

Cross referencing

Cross referencing is linking internally, from one place in the documentation to another. This is **not** done using the *Hyperlinks* syntax, but with one of the several roles:

```
:role:`target`
  becomes...
    <a href="target">reference title</a>

:role:`anchor text <target>`
  becomes...
    <a href="target">anchor text</a>
```

:doc:

- Links to documents (pages)
- *target* is the file name, without the `.rst` extension
- *title* is the first *headline* (`<h1>`) of the page

:ref:

- Links to *sections*
- *target* is the *Section labels*
- *title* is the *section title (headline)*

:term:

- Links to items in the Glossary
- *target* is the term, in the glossary
- *title* is the term itself

To recap: If you do not include an explicit `target`, the text inside the role will be understood as the target, and the anchor text for the link in the output will be the title of the target.

For example:

```
- Link to this document:

  - :doc:`contributing`
  - :doc:`anchor text <contributing>`

- Link to this section:

  - :ref:`cross-referencing`
  - :ref:`anchor text <cross-referencing>`

- Link to a term:

  - :term:`participant`
  - :term:`anchor text <participant>`
```

- Link to this document:
  - *Contributing to ODK Docs*
  - *anchor text*
- Link to this section:
  - *Cross referencing*
  - *anchor text*
- Link to a term:
  - participant
  - anchor text

Writing about User Interface

Several roles are used when describing user interactions.

:guilabel:
Marks up *actual UI text* of form labels or buttons.

```
Press the :guilabel:`Submit` button.
```

:menuselection:
Marks up the *actual UI text* of a navigation menu or form select element.

```
Select :menuselection:`Help` from menu.
```

When writing about multi-level menus, use a single `:menuselection:` role, and separate menu choices with `-->`.

```
To save your file, go to :menuselection:`File --> Save` in the Main Menu.
```

Note: In some situations you might not be clear about which option (*`menuselection`* or *`guilabel`*) to use. GUIs in real life can sometimes be ambiguous. The general rule is:

- Actual UI text will always receive *`guilabel`* role unless the text could reasonably be understood to be part of a menu.
- If the actual UI text could be understood as a menu, *`menuselection`* should be used.

These both render the same on output, so don't worry too much if you get it wrong. Just use your judgment and take your best guess.

:kbd:
Marks up a sequence of literal keyboard strokes.

```
To stop the local server, type :kbd:`CTRL C`.
```

:command:
Marks up a terminal command.

```
To build the documentation, use :command:`sphinx-build`.
```

:option:
Marks up a terminal command option.

```
The :option:`-b html` option specifies the HTML builder.
```

:gesture:
Describes a touch screen gesture.

```
:gesture:`Swipe Left`
```

Writing about forms

We have added several custom text roles for writing about forms and the XForms and XLSForm formats.

:th:
Used to refer to a table header cell.

:tc:
Used to refer to a table cell.

```
External App String Widget
~~~~~~~~~~~~~~~~~~~~~~~~~~~
The external app widget is displayed when the :th:`appearance` attribute␣
↪begins with :tc:`ex:`.
```

:formstate:
Specifies the state of the form in ODK Collect, which could be one of the following:

- Blank
- Finalized
- Saved
- Sent
- Deleted

```
:formstate:`Sent`
```

Other Semantic Markup

:abbr:

Marks up an abbreviation. If the role content contains a parenthesized explanation, it will be treated specially: it will be shown in a tool-tip in HTML.

```
:abbr:`ODK (Open Data Kit)`
```

:dfn:

Marks the defining instance of a term outside the glossary.

```
:dfn:`Open Data Kit` (ODK) is a suite of open source applications that help␣
↪organizations engaged in enumerator-mediated data collection.
```

:file:

Marks the name of a file or directory. Within the contents, you can use curly braces to indicate a "variable" part.

```
is installed in :file:`/usr/lib/python2.{x}/site-packages`
```

In the built documentation, the `x` will be displayed differently to indicate that it is variable.

:program:

Marks the name of an executable program.

```
launch the :program:`ODK Aggregate Installer`
```

Images and Figures

PNGs only

All still images used in ODK Docs should be PNG files. This helps us keep our image compression tooling simple, and generally results in higher-quality screenshots.

Whenever possible, you should generate your images as PNGs rather than converting to PNGs from another format. If you have to start in another format, use lossless formats whenever possible. These include BMP, GIF, and TIFF. (Avoid JPG/JPEG if possible, as this is a lossy format that does not replicate screenshots very well.)

Where to put image files

Image files should be put in the `/src/img/` directory in the source, and they should be in a subdirectory with the same name as the document in which they appear. (That is, the filename without the `.rst` extension.)

Image compression

Before committing images locally, run lossless compression on them using one of the following tools:

- ImageOptim
- Pngout

Inserting images in a document

To place an image in a document, use the `image` directive.

```
.. image:: /img/{document-subdirectory}/{file}.*
  :alt: Alt text. Every image should have descriptive alt text.
```

Note the literal asterisk (`*`) at the end, in place of a file extension. Use the asterisk, and omit the file extension.

Inserting images with captions (figures)

Use `figure` to markup an image with a caption.

```
.. figure:: /img/{document-subdirectory}/{file}.*
  :alt: Alt text. Every image should have descriptive alt text.

  The rest of the indented content will be the caption. This can be a short␣
↪sentence or several paragraphs. Captions can contain any other rst markup.
```

Inline images

To information on creating inline images, see *Substitutions*.

Image File Names

Image file names should:

- be short yet descriptive
- contain only lower case characters and (in *Sequentially numbered images* only) numbers
- have no spaces
- use hyphens as the separator

Good image file names:

- `collect-home-screen.png`
- `build-data-export-menu.png`

Bad image file names:

- `Collect home screen.png`
- `collect_home_screen.png`
- `3987948p2983768ohl84692p094.jpg-large`

Sequentially numbered images

In the case of sequentially numbered images, the numbers should:

- be zero-indexed
- have two digits with leading zeroes
- be separated from the rest of the file name with a hyphen
- be placed at the end of the file name

Good sequentially numbered image file names:

- `map-widget-00`, `map-widget-01`, `map-widget-02`

Bad sequentially numbered image file names:

- `1-map-widget`, `2-map-widget`
- `map-widget_00`, `map-widget_01`
- `map-widget-1`, `map-widget-2`

Screenshots from ODK Collect

If you have set up Android Debug Bridge, you can connect your Android device to your computer and take screenshots from the command line.

- Connect your device via USB
- Enable Developer Settings
  - *Settings* → *About phone*
  - Tap *Build number* seven (7) times
- Turn on USB Debugging
  - *Settings* → *Developer options* → *USB debugging*

Now, at the command line, from the root directory of the `odk-docs` repo:

```
python ss.py {document-name}/{image-name}
```

- `{document-name}` is the filename (without extension) where the image will be used.
- `{image-name}` is the name (without extension) given to the image. - follow the *Image File Names* guidelines

Warning: Make sure you do not overwrite an existing image.

Tip: If you have a problem running ss.py, check to make sure your Python 3 virtual environment is activated.

Tip: Be sure to obscure any personally-identifiable information from screen shots. Crop to the smallest relevant screen area. Annotate screen shots with arrows or circles to indicate relevant information.

Videos

Video files should be put in the `/src/vid/` directory in the source, and they should be in a subdirectory with the same name as the document in which they appear. (That is, the filename without the `.rst` extension.)

The purpose of on page videos is to illustrate complicated user interactions that might be difficult to describe otherwise. Longer tutorial videos should be hosted elsewhere and, if

appropriate, linked to from the docs. Therefore:

- The length of the videos must be less than a minute.
- Videos should have no audio.

To insert a video, use the custom *video* directive.

`.. video:: path/to/video`

Specify the source path of the video and a descriptive alt content in the video directive. Alternate content is displayed when the video cannot be played. It can contain long texts as well as any other rst content.

```
.. video:: /vid/{document-subdirectory}/{file}.ext

  Alt content. Every video should have descriptive alt content.
```

The following optional attributes are supported:

`:autoplay:`

Specifies whether the video should start playing as soon as it is ready. Can take boolean value: true, false, yes or no. Default is **no**.

It is almost never a good idea to turn `autoplay` on.

`:controls:`

Specifies whether the video controls should be displayed. Can take boolean value: true, false, yes or no. Default is **yes**.

`:muted:`

Specifies whether the audio output of the video should be muted. Can take boolean value: true, false, yes or no. Default is **yes**.

`:loop:`

Specifies whether the video should start over again, every time it is finished. Can take boolean value: true, false, yes or no. Default is **no**.

`:preload:`

Specifies if and how the author thinks the video should be loaded when the page loads. Can take one of the following three values: **auto**, **metadata** or **none**.

`:poster:`

Contains the source address for an image to be shown while the video is downloading, or until the user hits the play button.

Note: Images to be used as poster for a video should be in the same directory as the video and should have a filename like `[same-file-name-as-video]-poster.ext`.

`:class:`
Specifies a class for the video element.

For more details on these attributes, see this guide.

To add a video in a document with the above options, you can do the following:

```
.. video:: /vid/{document-subdirectory}/{file}.ext
  :autoplay: yes/no
  :controls: yes/no
  :muted: yes/no
  :loop: yes/no
  :class: class-name
  :preload: auto/metadata/none
  :poster:: /vid/{document-subdirectory}/{file}.ext

  Alt content. Every video should have descriptive alt content.
```

Capturing video from Android

Android Debug Bridge (ADB) can be used to capture a screen recording from an Android app.

```
$ adb shell screenrecord /sdcard/example.mp4
```

On pressing the enter key the video recording starts. Recording stops automatically after 3 minutes. Since videos have to be less than a minute, press `CTRL C` to stop the recording.

The video file is saved in your Android device to a file at `/sdcard/example.mp4` file.

To pull the video locally:

```
$ adb pull /sdcard/example.mp4 local/path/to/save/to
```

Downloadable files

Downloadable files should be put in the `/src/downloads/` directory in the source, and they should be in a subdirectory with the same name as the document in which they appear. (That is, the filename without the `.rst` extension.)

To place a downloadable file in a document, use the `download` role.

```
See this :download:`example script </downloads/contributing/example_script.py>`␣
↪to understand the procedure better.
```

Code Samples

Use the `code-block` directive to insert code samples. Specify the language on the same line as the directive for syntax highlighting.

```
.. code-block:: rst

  Use the ``code-block`` directive to markup code samples.

.. code-block:: python

  print("Hello ODK!")

.. code-block:: console

  $ python --version

.. code-block:: java

  public class HelloWorld {

      public static void main(String[] args) {
          // Prints "Hello, World" to the terminal window.
          System.out.println("Hello, World");
      }

  }
```

Note: **rst** code-blocks wrap overflow lines by default. To unwrap overflow lines, use **unwrap** class with **rst** code-blocks.

```
.. code-block:: rst
  :class: unwrap
```

Code-blocks for other languages don't wrap overflow lines. Instead of wrapping, you need to scroll side-ways. To wrap overflow lines with other code-blocks, use **wrap** class with them.

```
.. code-block:: python
  :class: wrap
```

Substitutions

Substitutions are a useful way to define a value which is needed in many places.

Substitution definitions are indicated by an explicit markup start (".. ") followed by a vertical bar, the substitution text (which gets substituted), another vertical bar, whitespace, and the definition block.

A substitution definition block may contain inline-compatible directives such as *image* or replace. For more information, refer this guide.

You can define the value once like this:

```
.. |RST| replace:: reStructuredText
```

and then reuse it like this:

```
We use |RST| to write documentation source files.
```

Here, `|RST|` will be replaced by reStructuredText

You can also create a reference with styled text:

```
.. |slack| replace:: **ODK Slack**
.. slack: https://opendatakit.slack.com
```

You can use the hyperlink reference by appending a "_" at the end of the vertical bars, for example:

```
You can ask about your problem in |slack|_.
```

You can ask about your problem in **ODK Slack**.

The `rst_epilog` in `conf.py` contains a list of global substitutions that can be used from any file. The list is given below:

- If you want to create a hyperlink reference for ODK Slack, you can use `|odk-slack|_`.

  ```
  You can use |odk-slack|_ to ask your questions.
  ```

  You can use ODK Slack to ask your questions.

- To create a hyperlink reference for docs related issues, use `|docs-issue|_`.

  ```
  If you find a problem, file an |docs-issue|_.
  ```

  If you find a problem, file an issue.

- To create a hyperlink reference for ODK Forum, use `|forum|_`.

```
You can ask support questions in |forum|_.
```

  You can ask support questions in ODK Forum.

- To create a hyperlink reference for contributors guide, use `|contrib-guide|_`.

```
Be sure to read the |contrib-guide|_.
```

  Be sure to read the contributors guide.

You can add inline images in the document using substitutions. The following block of code substitutes arrow in the text with the image specified.

```
The |arrow| icon opens the jump menu.

.. |arrow| image:: /img/{document-subdirectory}/{file}.*
           :alt: Alt text.
```

4.17.3 Docs Style Guide

Spelling and grammar

American spelling and grammar

Whenever U.S. English and British (or other) English spelling or usage disagree, standard U.S. spelling and usage is preferred.

Wrong

```
The colour of the button is grey.
```

Right

```
The color of the button is gray.
```

```
@memoize
def check_ukus(text):
    """UK vs. US spelling usage."""
    err = "style-guide.uk-us"
    msg = "uk-vs-us-spell-check. '{}' is the preferred spelling."

    preferences = [
        ["gray",                ["grey"]],
        ["color",               ["colour"]],
        ["accessorizing",       ["accessorising"]],
        ["acclimatization",     ["acclimatisation"]],
        ["acclimatize",         ["acclimatise"]],
        ["acclimatized",        ["acclimatised"]],
    ]

    return preferred_forms_check(text, preferences, err, msg)

    # This is a sample code. The complete code can be found in the file:
    # proselint-extra.py.
```

Quote marks

- Quote marks should generally be avoided if possible.
- *Smart* quotes (also known as *curly quotes* or *directional quotes*) are not permitted in source files.

Avoid quote marks

Quote marks are used in prose writing to indicate verbatim text. This is rarely useful in technical writing, as verbatim text usually requires a more specific semantic markup.

Wrong

```
Click the button that says, "Save."
```

Right

```
Click :guilabel:`Save`.
```

Wrong

```
You may see an error message that says, "Something went wrong."
```

Right

```
You may get an error: ``Something went wrong.``
```

```
def check_quotes(text):
    """Avoid using straight quotes."""
    err = "style-guide.check-quote"
    msg = "Avoid using quote marks."
    regex = r"\"[a-zA-z0-9 ]{1,15}\""

    errors = []

    for matchobj in re.finditer(regex, text):
        start = matchobj.start()+1
        end = matchobj.end()
        (row, col) = line_and_column(text, start)
        extent = matchobj.end()-matchobj.start()
        errors += [(err, msg, row, col, start, end,
                        extent, "warning", "None")]

    return errors
```

Straight quotes

Any time that you *do* need to use quotation marks, use straight (or *plain*) quotes. Sphinx and Docutils will output the typographically correct quote style.

```
def check_curlyquotes(text):
    """Do not use curly quotes."""
    err = "style-guide.check-curlyquote"
    msg = "Do not use curly quotes. If needed use straight quotes."
    regex = r"\"[a-zA-z0-9 ]{1,15}\""

    errors = []

    for matchobj in re.finditer(regex, text):
        start = matchobj.start()+1
        end = matchobj.end()
        (row, col) = line_and_column(text, start)
```

```
        extent = matchobj.end()-matchobj.start()
        errors += [(err, msg, row, col, start, end,
                        extent, "warning", "None")]

    return errors
```

Serial comma

In a comma-delineated list of items, the penultimate item should be followed by a comma.

Wrong

```
Apples, oranges and pears.
```

Right

```
Apples, oranges, and pears.
```

```
@memoize
def check_comma(text):
    """Use serial comma after penultimate item."""
    err = "style-guide.serial-comma"
    msg = "Use serial comma after penultimate item."
    regex = "\,\s[a-zA-Z0-9]+\sand\s"

    return existence_check(text, [regex], err, msg, require_padding=False)
```

A bulleted list is often more clear than an inline list.

Correct

```
You will need to be familiar with git, GitHub, and Python.
```

Possibly Better

```
You will need to be familiar with:
```

```
- git
- GitHub
- Python
```

There's no hard rule about which to use in any situation. Use your judgement: try it both ways and see which is more clear.

Direct Address

Direct address — speaking directly to the reader using the second person "you" — is preferred over passive voice ("it can be done"), first-person plural ("we can do it"), or other constructions.

First person plural ("we") should only be used when speaking of the ODK project team ("We recommend…").

Ordered and unordered lists

An ordered list is numbered. It should be used when the order of the list is essential. For example, when enumerating a series of steps in a procedure.

Wrong

```
- First we do this.
- And then we do this.
- And then we do this.
```

Right

```
1. Do this.
2. Do this.
3. Do this.
```

An unordered list is bulleted. It should be used for a collection of items in which order is not essential.

Wrong

```
1. apples
2. oranges
3. bananas
```

Right

```
- apples
- oranges
- bananas
```

Avoid Latin

Several Latin abbreviations are common in written English:

At best, these present a minor barrier to understanding. This is often made worse by unintentional misuse.

Avoid Latin abbreviations.

Wrong

```
If you are writing about a specific process (e.g., installing an application)...
```

Right

```
If you are writing about a specific process (for example, installing an␣
↪application)...
```

```
@memoize
def check_latin(text):
    """Avoid using Latin abbreviations."""
    err = "style-guide.latin-abbr"
    msg = "Avoid using Latin abbreviations like \"etc.\", \"i.e.\"."

    list = [
        "etc\.", "etc", "\*etc\.\*", "\*etc\*",
        "i\.e\.", "ie", "\*ie\.\*", "\*ie\*",
        "e\.g\.", "eg", "\*eg\.\*", "\*eg\*",
```

```
        "viz\.", "viz", "\*viz\.\*", "\*viz\*",
        "c\.f\.", "cf", "\*cf\.\*", "\*cf\*",
        "n\.b\.", "nb", "\*nb\.\*", "\*nb\*",
        "q\.v\.", "qv", "\*qv\.\*", "\*qv\*",
        "ibid\.", "ibid", "\*ibid\.\*", "\*ibid\*",
      ]

    return existence_check(text, list, err, msg, ignore_case=True)
```

Etc.

Et cetera (or *etc.*) deserves a special mention.

Et cetera means "and all the rest," and is often used to indicate that there is more that could or should be said, but which is being omitted.

Writers often use *etc.* to gloss over details of the subject which they are not fully aware of. If you find yourself tempted use *etc.*, ask yourself if you really understand the thing you are writing about.

Avoid unneeded words

Adverbs

Adverbs often contribute nothing. Common offenders include:

- simply
- easily
- just
- very
- really
- basically
- extremely
- actually

Wrong

```
To open the file, simply click the button.
```

Right

```
To open the file, click the button.
```

Wrong

```
You can easily edit the form by...
```

Right

```
To edit the form...
```

```python
@memoize
def check_adverb(text):
    """Avoid using unneeded adverbs."""
    err = "style-guide.unneed-adverb"
    msg = "Avoid using unneeded adverbs like \"just\", \"simply\"."

    list = [
        "simply",
        "easily",
        "just",
        "very",
        "really",
        "basically",
        "extremely",
        "actually",
    ]

    return existence_check(text, list, err, msg, ignore_case=True)
```

Filler words and phrases

Many words and phrases provide no direct meaning. They are often inserted to make a sentence seem more formal, or to simulate a perceived style of business communication. These should be removed.

Common filler phrases and words include:

- to the extent that
- for all intents and purposes
- when all is said and done
- from the perspective of
- point in time

This list is not exhaustive. These "canned phrases" are pervasive in technical writing. Remove them whenever they occur.

```
@memoize
def check_filler(text):
    """Avoid using filler phrases."""
    err = "style-guide.filler-phrase"
    msg = "Avoid using filler phrases like \"to the extent that\"."

    list = [
        "to the extent that",
        "when all is said and done",
        "from the perspective of",
        "point in time",
    ]

    return existence_check(text, list, err, msg, ignore_case=True)
```

Semicolons

Semicolons are used to separate two independent clauses which could stand as individual sentences but which the writer feels would benefit by close proximity.

Semicolons can almost always be replaced with periods (full stops). This rarely diminishes correctness and often improves readability.

Correct

```
These "canned phrases" are pervasive in technical writing; remove them whenever␣
↪they occur.
```

Better

```
These "canned phrases" are pervasive in technical writing. Remove them whenever␣
↪they occur.
```

```python
@memoize
def check_semicolon(text):
    """Avoid using semicolon."""
    err = "style-guide.check-semicolon"
    msg = "Avoid using semicolon."
    regex = ";"

    return existence_check(text, [regex], err, msg, require_padding=False)
```

Pronouns

Third-person personal pronouns

Third-person personal pronouns are:

- he/him/his
- she/her/her(s)
- they/them/their(s)

Note: While some people consider *they/them/their* to be non-standard (or "incorrect") as third-person singular, it has gained wide use as a gender-neutral or gender-ambiguous alternative to *he* or *she*.

There are two issues with personal pronouns:

- gender bias
- clarity

To avoid gender bias, the third person gender-neutral *they/then/their(s)* is preferred over *he* or *she* pronouns when writing about abstract individuals.

Wrong

```
The enumerator uses his device.
```

Right

```
The enumerator uses their device.
```

Unfortunately, *they/them/their* is not a perfect solution. Since it is conventionally used as a plural pronoun, it can cause confusion.

Therefore, avoid the use of personal pronouns whenever possible. They are often out of place in technical writing anyway. Rewriting passages to avoid personal pronouns often makes the writing more clear.

Correct

```
When using Collect, first the enumerator opens the app on their device. Then␣
↪they complete the survey.
```

Better

```
To use Collect:

- open the app
- complete the survey
```

```
@memoize
def check_pronoun(text):
    """Avoid using third-person personal pronouns."""
    err = "style-guide.personal-pronoun"
    msg = "Avoid using third-person personal pronouns like \"he\", \"she\". In␣
↪case of absolute need, prefer using \"they\"."

    list = [
        "he",
        "him",
        "his",
        "she",
        "her",
        "hers",
    ]

    return existence_check(text, list, err, msg, ignore_case=True)
```

"Same"

Same, when used as an impersonal pronoun, is non-standard in Modern American English. It should be avoided.

Wrong

```
ODK Collect is an Android app. The same can be used for...
```

Right

```
ODK Collect is an Android app. It can be used for...
```

Right

```
ODK Collect is an Android app that is used to...
```

```
@memoize
def check_same(text):
    """Avoid using impersonal pronoun same."""
    err = "style-guide.check-same"
    msg = "Avoid using \"The same\"."
    regex = "\. The same"

    return existence_check(text, [regex], err, msg, ignore_case=False,
                           require_padding=False)
```

Titles

Title case and sentence case

Document titles should be in `Title Case` – that is, all meaningful words are to be capitalized.

Section titles should use `Sentence case` – that is, only the first word should be capitalized, along with any proper nouns or other words usually capitalized in a sentence.

Verb forms

If a document or section describes a procedure that someone might do, use a verb ending in *-ing*. (That is, a gerund.) Do not use the "How to…" construction.

Wrong

```
How to install ODK Collect
--------------------------
```

Right

```
Installing ODK Collect
----------------------
```

If section title is a directive to do something (for example, as a step in a procedure), use an imperative.

```
Installing ODK Aggregate
------------------------

Download ODK Aggregate
~~~~~~~~~~~~~~~~~~~~~~

Section content here.
```

```
@memoize
def check_howto(text):
    """Avoid using how to construct."""
    err = "style-guide.check-howto"
    msg = "Avoid using \"How to\" construction."
    regex = "(How to.*)(\n)([=~\-\"\*]+)"

    return existence_check(text, [regex], err, msg, require_padding=False)
```

Section labels

Section titles should almost always be preceded by labels.

The only exception is very short subsections that repeat — like the **Right** and **Wrong** titles in this document or the **XLSForm Rows** and **XForm XML** sections in the Question Types document.

In these cases, you may want to use the `rubric` directive.

```
def check_label(text):
    """Prefer giving a section label."""
    err = "style-guide.check-label"
    msg = "Add a section label if required."
    regex = r"(.*\n)(( )*\n)(.+\n)(([=\-~\"\']){3,})"

    errors = []
    sym_list = ['===','---','~~~','"""','\'\'\'']
    is_doc_title = True

    for matchobj in re.finditer(regex, text):
        if is_doc_title:
            is_doc_title = False
            continue
        label = matchobj.group(1)
        start = matchobj.start()+1
        end = matchobj.end()
        (row, col) = line_and_column(text, start)
        row = row + 2
        if any(word in text.splitlines(True)[row] for word in sym_list):
            row = row - 1
        col = 0
        extent = matchobj.end()-matchobj.start()
        catches = tuple(re.finditer(r"\.\. _", label))
        if not len(catches):
            errors += [(err, msg, row, col, start, end,
                        extent, "warning", "None")]

    return errors
```

Other titling considerations

- Do not put step numbers in section titles.
- Readers skim. Section titles should be clear and provide information.

Writing code and writing about code

ODK Documentation includes code samples in a number of languages. Make sure to follow generally accepted coding style for each language.

Indenting

In code samples:

- Use spaces, not tabs.
- Two spaces for logical indents in most languages.
  - Python samples must use four spaces per indent level.
- Strive for clarity. Sometimes nonstandard indentation, especially when combined with non-syntactic line breaks, makes things easier to read.
  - Make sure that line breaks and indentation stay within the valid syntax of the language.

Using two spaces keeps code sample lines shorter, which makes them easier to view.

Example of indenting for clarity

```
HTTP/1.0 401 Unauthorized
Server: HTTPd/0.9
Date: Sun, 10 Apr 2005 20:26:47 GMT
WWW-Authenticate: Digest realm="testrealm@host.com",
                         qop="auth,auth-int",
                         nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                         opaque="5ccc069c403ebaf9f0171e9517f40e41"
Content-Type: text/html
Content-Length: 311
```

Meaningful names

When writing sample code, avoid meaningless names.

Wrong

```
def myFunction(foo):

  for bar in foo:
     bar[foo] = foo[spam] + spam[foo]

  return foobar
```

XML and HTML

Some of the terms often used to describe XML and HTML code structures are imprecise or confusing. For clarity, we restrict certain terms and uses.

Likewise, coding practices and styles for XML and HTML vary widely. For the sake of clarity and consistency, samples should follow the guidelines set forth here.

Element

The following piece of code represents an **element**:

```
<element>
  Some content.
</element>
```

Note: An element is **not** a *block* or a *tag*.

- *Tag* is defined below.
- *Block* has a specific meaning in HTML and XML templates, and should generally be avoided outside those contexts.

Tag

A **tag** is the token that begins or ends an element.

```
<element>  <!-- The opening tag of this element. -->
  Some content.
</element> <!-- The closing tag. -->
```

The word *tag* has often been used to refer to the entire element. For clarity, we will avoid that here.

Node

The word *node* is often used interchangeably with *element*.

For clarity, we make the following distinction:

- An HTML or XML document has *elements*, not *nodes*.
- A *node* is part of a "live" DOM tree or other dynamic representation.

  - An XML or HTML element becomes an *element node* in a DOM tree.
  - There are also other types of nodes in a DOM tree.

Attributes and values

An element may have attributes. Attributes have values. Values are wrapped in straight double-quotes.

```
<element attribute="value">
  Content.
</element>
```

Other names for attributes, such as *variables* or *properties*, should be avoided.

Element content

The code between the opening and closing tags of an element is the content. Content can include other elements, which are called *child elements*.

```
<element>
  Content.
  <child-element>
    More content.
  </child-element>
</element>
```

When an element is empty, it can be called a *null element*.

```
<null-element attribute="value" />
```

In XML, null element tags always self-close. This is not the case in HTML.

- HTML elements that are always null (for example, *<img>*) do not need to be self-closed.
- Empty HTML elements that normally accept content have a separate closing tag.

```
<img src="awesome-picture.jpeg">

<script src="some-javascript.js"></script>
```

Capitalization

For all HTML samples, tag names and attribute names should be `all lowercase`.

Newly-written XML examples should also be `all lowercase`.

XML examples that show actual code generated by tools in the ODK ecosystem should replicate that code exactly, regardless of its capitalization practice.

ODK jargon

ODK and ODK Docs

Wrong

- Odk
- odk
- Open data kit
- OpenDataKit
- the Open Data Kit
- ODK docs
- ODK documentation

Right

- ODK
- Open Data Kit
- ODK Docs
- ODK Documentation

Probably want to avoid...

- Open Data Kit Documentation

```
@memoize
def check_odkspell(text):
    """ODK spelling usage."""
    err = "style-guide.spelling-odk"
    msg = "ODK spell check. '{}' is the preferred usage."

    preferences = [
```

```
        ["Open Data Kit",         ["Open data kit"]],
        ["Open Data Kit",         ["OpenDataKit"]],
        ["ODK",                   ["Odk"]],
        ["ODK",                   ["{0} odk"]],
        ["ODK Docs",              ["ODK docs"]],
        ["ODK Documentation",     ["ODK documentation"]]
    ]

    return preferred_forms_check(text, preferences, err, msg, ignore_case=False)
```

ODK app and project names

ODK includes a number of components, including:

- Collect
- Aggregate
- Briefcase

These should always be capitalized.

The **ODK** prefix (as in, *ODK Collect*) should be used the first time a document mentions the app or project, or any other time it would be unclear.

A few projects should *always* use the **ODK** prefix:

- ODK XForm
- ODK Javarosa
- ODK Docs

```
@memoize
def check_appspell(text):
    """ODK spelling usage."""
    err = "style-guide.spelling-odk"
    msg = "ODK spell check. '{}' is the preferred usage."

    preferences = [
        ["Aggregate",             ["{0} aggregate"]],
        ["Briefcase",             ["{0} briefcase"]]
    ]

    return preferred_forms_check(text, preferences, err, msg, ignore_case=False)
```

XForms and XLSForm

- *XForms* refers to XML-encoded forms.
- *XLSForm* refers to a spreadsheet format used to define forms.

Wrong

- Xforms
- X-Forms
- xforms
- XFORMS
- XForm (no *s*, when referring to the specification)
- xlsform
- XLSform
- Xlsform

Right

- XForms
- an Xform (when referring to a single form)
- XLSForm

```
@memoize
def check_formspell(text):
    """ODK spelling usage."""
    err = "style-guide.spelling-odk"
    msg = "ODK spell check. '{}' is the preferred usage."

    preferences = [
        ["XForms",                ["Xforms"]],
        ["XForms",                ["X-Forms"]],
        ["XForms",                ["{0} xforms"]],
        ["XForms",                ["XFORMS"]],
        ["an XForm",              ["a XForm"]],
        ["an XLSForm",            ["a XLSForm"]],
        ["XLSForm",               ["{0} xlsform"]],
        ["XLSForm",               ["XLSform"]],
        ["XLSForm",               ["Xlsform"]]
```

```
    ]

    return preferred_forms_check(text, preferences, err, msg, ignore_case=False)
```

XForms Spec, XForms Tools, XForms

XForms can refer to:

- The XML-based form format
- The official XForms specification from the W3C
- The ODK XForms Specification, which is a subset of the full W3C recommendation.
- The general idea of an XML-based form.

XForm (without an *s*) refers to:

- A specific XML document that encodes a form.

When writing about any of these things, make sure you are clear — in your mind as well as in your writing — which one you are talking about.

XLSForm

XLSForm can refer to:

- The XLSForm format for describing form in an Excel spreadsheet
- A spreadsheet file that describes a form using the format.
- A tool for converting `*.xls(x)` files to XForm documents.

When writing about any of these things, make sure you are clear — in your mind as well as in your writing — which one you are talking about.

4.17.4 Docs Developer Guide

This document is for contributors working on the design, templating, deployment, or development of the ODK Docs website.

Tech Overview

ODK Docs uses:

- Sphinx, a static-site generator written in Python

  Sphinx uses:

  - Docutils for parsing reStructuredText
  - Jinja for templating

- sphinx_rtd_theme, a Sphinx theme/template

  sphinx_rtd_theme uses:

  - JQuery, a JavaScript library

- Proselint for style testing
- git and GitHub for version control
- CircleCI for testing and deployment
- Amazon S3 for hosting

Custom HTML templating

ODK Docs uses the sphinx_rtd_theme, with some minor customizations.

ODK-specific versions of HTML/Jinja templates are in `_templates`. Any file in that directory will override the file of the same name in the sphinx_rtd_theme source.

So, to customize a portion of the HTML template, copy the source file from sphinx_rtd_theme and then edit it.

Please commit the copied file unchanged before editing, so that it is easy to track what you have changed.

Custom JavaScript

Custom JavaScript should be added in `src/_static/js/custom.js`. Comment your code with an explanation of what the JS accomplishes, and a reference to the issue number you are working on.

The ODK Docs template includes JQuery, so you can use it in your custom JS.

Custom CSS

Custom CSS should be added in `src/_static/css/custom.css`. Comment your code with an explanation of what the CSS accomplishes and a reference to the issue number you are working on.

For example:

```
/* Example CSS PR #xyx */

div[class^='example'] {
  color: black;
}
```

It is helpful to keep the CSS file organized. There are several sections in the `custom.css` file:

- Styling for rst roles and directives
- Responsive CSS
- Styling for JS implementation
- Utility classes

Each of these sections are enclosed in start and end comments. Add your code to the relevant section. If you don't find any section relevant, add a new section and add your code there.

For example:

```
/* New section starts */

/* Example CSS PR #xyx */

div[class^='example'] {
  color: black;
}

/* New section ends */
```

Style Guide checks

Proselint is used for style testing the docs. Apart from the built-in tests in proselint, custom checks are added for style guide testing. Following a literate programming. model, style checks are defined in *docs-style-guide.rst*. After each style rule, you can define a python code-block containing the code for style testing. When the style-test script is run, these python code-blocks are parsed to generate a testing script.

Proselint dependent checks

In most of the custom checks, a new function is written that calls one of the built-in proselint functions as a return value.

All the checks use a decorator *`memoize()`* to cache the check for faster execution.

memoize()

Use `@memoize` above function definition to cache the result.

Proselint provides several functions for defining style tests:

existence_check(*text, list, err, msg, ignore_case=True, str=False, max_errors=float("inf"), offset=0, require_padding=True, dotall=False, excluded_topics=None, join=False*)

To check for existence of a regex pattern(s) in the text. The parameters `offset`, `excluded_topics` and `join` are not needed for style guide testing.

Parameters

- **text** (*str*) – Text to be checked
- **list** (*list*) – List of regex expressions
- **err** (*str*) – Name of the test
- **msg** (*str*) – Error or warning message
- **ignore_case** (*bool*) – For using `re.IGNORECASE`
- **str** (*bool*) – For using `re.UNICODE`
- **max_errors** (*float*) – Maximum number of errors to be generated
- **require_padding** (*bool*) – To use padding with the specified regex (It is better to set it as **False** and specify the regex accordingly)
- **dotall** (*bool*) – For using `re.DOTALL`

Returns The error list consisting of error tuples: `[(start, end, err, msg, replacement)]`.

Return type list

preferred_forms_check(*text, list, err, msg, ignore_case=True, offset=0, max_errors=float("inf")*)

To suggest a preferred form of the word used. The parameter `offset` is not needed for style guide testing.

Parameters

- **text** (*str*) – Text to be checked
- **list** (*list*) – list of comparison (words or regex): `[correct form , incorrect form]`
- **err** (*str*) – Name of the test
- **msg** (*str*) – Error or warning message
- **ignore_case** (*bool*) – For using `re.IGNORECASE`
- **max_errors** (*float*) – Maximum number of errors to be generated

Returns The error list consisting of error tuples: `[(start, end, err, msg, replacement)]`.

Return type list

`consistency_check`(*text*, *word_pairs*, *err*, *msg*, *offset=0*)

To check for consistency for the given word pairs. The parameters `offset` is not needed for style guide testing.

Parameters

- **text** (*str*) – Text to be checked
- **word_pairs** (*list*) – Word pairs to be checked for consistency
- **err** (*str*) – Name of the test
- **msg** (*str*) – Error or warning message

Returns The error list consisting of error tuples: `[(start, end, err, msg, replacement)]`.

Return type list

Note: The checker functions are used by the built-in proselint function `lint()` to generate an error list of different format. The returned list finally is: `[(check, message, line, column, start, end, end - start, "warning", replacements)]`

See also:

Proselint source code

Example Usage

```
@memoize
def example(text):
    """Example check."""
    err = "style-guide.example"
    msg = "A demonstration for writing checks."
    regex = "[\.\?!](example)"

    return existence_check(text, [regex], err, msg, ignore_case=False,
                           require_padding=False)
```

When you define code-blocks which use built-in proselint testing, specify the class **style-checks**.

```
.. code-block:: python
  :class: style-checks
```

The generated file after parsing code for style checks is `style-checks.py`.

If the test is too large to be defined in the file `docs-style-guide.rst`, you can use a snippet from the test (as *here*). The code-blocks for such snippets should specify the class **proselint-extra-checks**. Define the complete test in the file `/style-guide/proselint-extra-checks.py`.

Independent checks

Apart from the checks, which are to be run through proselint, you can add extra checks to be run independently. They are not enabled in `proselintrc` as well. For example, the checks for finding quote marks and section labels do not use any built-in functions to obtain an error list.

Example Usage

```
def check_quotes(text):
    """Avoid using straight quotes."""
    err = "style-guide.check-quote"
    msg = "Avoid using quote marks."
    regex = r"\"[a-zA-z0-9 ]{1,15}\""

    errors = []

    for matchobj in re.finditer(regex, text):
        start = matchobj.start()+1
        end = matchobj.end()
        (row, col) = line_and_column(text, start)
        extent = matchobj.end()-matchobj.start()
        errors += [(err, msg, row, col, start, end,
                        extent, "warning", "None")]

    return errors
```

The code-blocks for extra checks should specify the class **extra-checks**. The generated file after parsing code for extra checks is `extra-checks.py`.

Note: Built-in proselint function *`line_and_column()`* is used with extra checks to obtain the row and column of the matched text.

line_and_column(*text*, *start*)

To find the line number and column of a position in a string.

Parameters

- **text** (*str*) – Text to be searched for
- **start** (*int*) – Starting position of matched pattern

Returns Tuple containing row and column number

Return type tuple

Error vs warning

- Warnings are intended to provide guidance to authors.
- Errors enforce "hard" rules, and raising an error will stop the build.

You can classify the result of a check as an error if you are sure that no false positives would be produced. The checks classified as errors should return a replacement for fixing the errors. Proselint dependent checks which use the function *preferred_forms_check()* or *consistency_check()* always return a preferred form. If you create an independent check which generates an error make sure to return a replacement in the error list.

To generate an error from a check, specify the check name in the list of errors in the function `get_errlist()` in the file `style-test.py`.

Excluding built-in proselint checks

To exclude an built-in proselint check, specify the check name in the check list in the function `exclude_checks()` in the file `style-test.py`.

4.17.5 Tips for Making Good Contributions

Smallest meaningful PR

A PR should normally address one issue. This makes it easier to review, easier to deploy, and easier to roll back in case of a problem. Additionally, the smaller the PR, the less likely it is to create a merge issue.

The exception is when several issues are closely related or can reasonably be worked on together. In this case, it should be clear by looking at the conversation on the Github issues that the items are related and will be worked on together. Your PR message should also

make it clear which issues are being worked on, and whether the PR closes the issues or not. Mention the PR by number:

> addresses #123
>
> closes #123

Descriptive PR names

A PR title should answer the question, "What does this Pull Request do?"

Good PR titles:

- adds a video directive
- makes navigation buttons responsive
- swaps placement of nav buttons and file an issue note

Bad PR titles:

- fix issue
- fix #123
- collect

Small, atomic commits

When working locally, commit often. Don't wait until you have 100 lines of changes across multiple files.

- If you need to copy or move a large section or file, commit that change before also editing it.
- If you have to create a new template file based on an existing template file, copy the file in one commit and then work on the changes. This makes it easier to know what you actually did.
- If writing a new doc, commits after each new section are a good idea.

Commit messages should answer the question, "What does this commit do?"

Small, well-named commits will help you keep track of your own work and make rollbacks and other changes easier to deal with.

Discuss issues before working

Take the time to clarify the needs and scope of an issue before committing to work on it. Especially for coding tasks, make sure you state your understanding and your plan before

working.

If you have a question, ask. Don't guess.

Note: Many new contributors don't ask questions because they are worried about appearing under informed. Please set this worry aside.

You will never be judged harshly for asking clarifying questions or for seeking more information.

Claim issues

If you decide to work on an issue, let the community know you are working on it by *claiming* the issue.

> @opendatakit-bot claim

Once you've claimed an issue, other people won't work on it. So make sure you're actually going to work on it before claiming it.

Don't claim more than one or two open issues at a time.

Share work in progress

It can be helpful to share your in-progress work. To mark a PR as a work in progress, append **WIP:** to the beginning of the PR title. We will not merge **WIP** PRs, and we won't do a review on them unless you ask.

If you want a review, comment, opinion, or help on a **WIP** PR, please tag the relevant person in the PR comments.

If you finish the work and want the PR to be merged, you do not need to open a new one. Just edit the PR title.

If you get stuck while working

- Ask for help in the issue comments. Maybe you can get back on track and complete the issue.
  - Asking questions is always better than guessing.
- Submit a **WIP** (work in progress) pull request.
  - If we can see what progress you have made, it is easier to offer help.
  - Even if you don't complete the task, perhaps someone else can pull in your in-progress work and build on it.

  - Sometimes your in-progress work is an improvement over not having it, and so we'll merge in something even if it isn't complete.

- If you really cannot move forward, it is okay to abandon an issue.

It is okay to abandon an issue

Sometimes you simply cannot complete work you have said you were going to complete. This could happen because you don't have all the required skills or knowledge to complete the work, or because the issue cannot actually be completed as scoped, or because you don't have the time.

Please let the community know in the issue discussion.

> @opendatakit-bot unclaim

This way, everyone knows that someone else can take up the project (or that we need to rethink it).

If you did significant work on a project before abandoning it, consider filing a **WIP** (work in progress) PR, so that others can see what you did and potentially build off of it. (Be sure to mention the issue, so the work is easy to find later.)

If an issue takes a long time to complete

For our purposes, a "long time" is a week or more, from when you first announce your intention to work on something until submitting a merge-ready PR.

An issue might take a long time because:

- it is complex and requires lots of hours
- you only have a short period of time each day to work on it
- you are new to the project and are having to learn as you go

The thing that matters is: **Are you actively working on the issue, and making progress, at least a little bit?**

If you are actively working on it, we do not want someone else to jump on and try to work on it at the same time. So please keep the community informed of your work by filing a **WIP** (work in progress) PR and committing to it as you work.

Issues only

All PRs must be directly connected to open issues. PRs should not represent suggestions, good ideas, or independent initiative.

If you have a good idea, file an issue. If you are curious about whether something should be an issue, chat with one of the core team in the *#docs-code* channel on the Slack.

Once you have filed an issue, wait for comment and approval before diving into the work. We do not want surprise PRs.

Actually install and use Open Data Kit or other tools

You cannot write effectively about tools you have not used. If you're going to write or edit documentation about any of the apps in the ODK ecosystem, you need to spend some time actually using it.

Before diving into writing documentation, try out the core tools *here https://opendatakit.org/software* and become familiar with them.

This is also true of writing about Sphinx or any of our documentation build tools. Reading existing documentation is not enough to write about something.

And actually do the thing

If you are writing about a specific process (installing an application, for example), you need to actually complete the process yourself. If possible, follow your own instructions *after* writing them to make sure they make sense.

Always build locally

Before submitting a PR, run the build locally to make sure you do not produce any errors or warnings. **We do not accept PRs that produce errors or warnings.**

It is best to run the build frequently as you work. You'll often catch simple mistakes that are harder to track down later.

You are not an impostor

Impostor syndrome is the feeling that you are not good enough or accomplished enough to do the work you are doing.

We all feel this way sometimes, and that's okay. But it is important to realize that **you are not an impostor.**

You can contribute to this community, no matter your background or skills.

- If there is something you don't know how to do, you can ask.
  - If it is issue related, ask on the issue.

  - If it is more general, try the #docs-code channel in the ODK Developer Slack.

- If you want to try something even though you aren't sure you can do it, go ahead and try.

Another worry you may have is that something will take you a long time when an "expert" might be able to do it quickly. You may feel, then, you aren't the "right person" for the job. But if you are the only one with the time or desire to work on something, **you are the right person to work on it.**

4.17.6 Working with Docs in Windows using Cygwin

The main contribution guide :doc: *contributing* was built for *nix systems, and the same commands may not work in Windows. Cygwin is a Windows tool, equivalent to *nix bash terminal. This guide helps to set up ODK Docs platform from the default Windows command prompt (CMD).

The prerequisites

The following are software tools that you need in the first place. If it is already installed. Just follow the steps and apply what is missing.

- Cygwin
- Python 3
- Virtualenv
- Virtualenvwrapper-win
- Git and GLFS

The requirements

These are a set of the main packages. ODK team combined them in a file such that all will be installed at once.

- alabaster==0.7.10
- Babel==2.4.0
- docutils==0.13.1
- imagesize==0.7.1
- Jinja2==2.9.6
- MarkupSafe==1.0

- Pygments==2.2.0
- pytz==2017.2
- requests==2.14.2
- six==1.10.0
- snowballstemmer==1.2.1
- Sphinx==1.6.1
- sphinx-rtd-theme==0.2.4
- sphinxcontrib-websupport==1.0.1
- typing==3.6.1
- update==0.4.4

Install Cygwin

Cygwin tool lets Windows users execute many *nix commands. Install Cygwin and add its path to Windows to work completely from the default command prompt. For instructions.

> **Warning:** If you encounter any downloading problems, make sure to select a mirror site near you. The entire list is available on the Cygwin website.

Python 3

You need to install Python 3. Select Python installation that fits your system (32 or 64 bit). For instructions, see.

Make sure to select the option "Add Python to the Path", as shown below.

Next, make sure to select "pip", which is *a package manager < https://pypi.python.org/pypi/pip>_* written in Python. We will use it to install packages. See the following image:

Alternatively, if you forgot to add Python 3 to the PATH, add it manually using the following

command:

```
> set PATH=%PATH%;C:\Users\your username\python3
```

Tip: `C:/Users/your username/python3` is the default Python 3 installation PATH. If you change it, please substitute the above path by the new one.

Virtual Environment

A virtual environment tool creates multiple Pythons environments, each has its packages and dependencies.

For easy installation, `pip` command can be used, which comes with Python 3 (as shown in Python installation).

```
> pip install virtualenv
```

Create a new directory for your odkdocs work:

```
> mkdir odk
```

To work with virtualenv, you have two options:

- Use the native virtualenv.
- Use virtualenvwrapper on the top of virtualenv.

Native Virtual Environment

Create a new Python 3 virtual environment, "odkenv" is the name of the virtualenv, you can choose any name.

```
> virtualenv -p <python path/python.exe> odkenv
```

After creating the virtualenv, multiple files are copied into the folder odkenv in your working directory.

```
> ls odkenv
```

The folder Scripts contains all virtualenv controls as ".bat" files.

To activate the odkenv:

```
> cd odkenv
.
.
.
> cd Scripts

> odk/odkenv/Scripts/activate.bat
```

To deactivate the odkenv:

```
> odk/odkenv/Scripts/deactivate.bat
```

Virtual Environment Wrapper

Tip: This step is not an alternative to virtualenv. You must install virtualenv first.

The *Virtualenvwrapper <https://pypi.python.org/pypi/virtualenvwrapper-win>* mediates between user CMD and virtualenv to ease management and working with multiple virtual environments. To install virtualenvwrapper, use the following:

```
> pip install virtualenvwrapper-win
```

Create a new virtualenv:

```
> mkvirtualenv odkenv
```

Once the odkenv is created, it is automatically activated:

```
(odkenv) /odk/docs
```

To deactivate the odkenv, write:

```
> deactivate
```

To activate the odkenv:

```
> workon odkenv
```

Git and GLFS

- Install Git for windows. Make sure that git is installed properly by typing (git) in the CMD.

- Install GLFS.

Android Tools

Install *Android Debug Bridge <collect-adb>* to learn more about ADB. ADB is part of Android studio, To download

To use ADB, you must run Android Studio once. The default location of ADB is `C:/Users/your username/AppData/Local/Android/sdk/platform-tools`. Add it to `Windows PATH` by using the following command:

```
set PATH=%PATH%;C:\Users\your user␣
↪name\AppData\Local\Android\sdk\platform-tools
```

Fork and Clone the ODK Docs repo

From Github, fork the ODK Docs. This will create a copy of the docs in your Github account called `origin`. Move to the ODk working directory, and clone the ODk Docs into your local machine.

```
> git clone https://github.com/your-github-username/docs.git
```

Set the Upstream Remote

```
> git remote add upstream https://github.com/opendatakit/docs.git
```

Install the Requirements

This step will install a bunch of packages that are listed in the :file: *requirement.txt* file. ODK team prepared this file for you to ease the installation.

First, you need to activate your virtual environment (odkenv):

```
> workon odkenv
```

Make sure you are inside the docs folder, then run:

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

You completed the installation and you can start change and build ODK Docs.