ODK Tables: Improving the Custom Views

by

Heidi So

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering
University of Washington
June 2013

Presentation of work given on ________________________

Thesis and presentation approved by ________________________

Date ________________________
Abstract

ODK Tables is an Android application that allows users to enter, collect and explore tabular data. There are multiple views, built-in or custom made by users, that users can view data through. The customizable views are made using HTML/JavaScript which allows users to choose what information to present, display data from a table in a way that best fits their use case, and incorporate the views into the application without the overhead of recompilation. One focus of my research has been on improving the interface for creating custom views so that building interesting and effective displays is a simpler task. Through this process, I have produced various examples of views customized for different datasets. In consideration for users with minimal experience in using HTML/JavaScript, I created code templates that all users can use, edit, and build upon for their custom views. Surveyed users say that the added features make custom views “very clear”; the written portion of the surveys further demonstrate that building these custom views are “very easy” for individuals with wide-ranging HTML/JavaScript backgrounds. This paper will further define the enhancements made to the interface and report performance and user evaluation results.

1 Introduction

There are two types of custom views that users can create: the List view and Detailed view. The List View allows for easy exploration of a table’s rows and the Detailed View serves as a more comprehensive display of each row’s data. Each list item represents a row in the table and can open up the Detailed view of that row when clicked. Specifics such as the exact appearance and content of the views can be established by the user in the HTML/CSS/JavaScript files. For each table, users can assign a file for each of the views. When a table is opened in the application and the user chooses to view the table in List view, the assigned files will be rendered by Android WebViews, which displays web pages within the application.

The goal of ODK Tables and its custom views is to help lower barriers to developing customized information applications [1]. By utilizing the objects and tools provided, users can create custom views that join tables by common properties and provide links to open other tables, results to queries, and more. Displaying table data in a more dynamic format helps create more coherent use case applications out of the seemingly separated tables. For that reason, these views must be highly customizable and easily produced by users for all use cases. Custom views are made using standard web technologies like HTML/JavaScript which gives users great flexibility in producing varying designs and displays. In an effort to provide users a useful and convenient interface for working with HTML/JavaScript for the custom views, I created my own for different datasets. I addressed potential issues users may have encountered with the early interface and added features that make building more effective displays easier and accessible.

This paper describes the additions made to the interface, shows examples of customized views and code templates, summarizes evaluations of the enhanced interface, and presents performance results for the loading of List views.
2 Implementation

2.1 Creating customized views

There two JavaScript objects that are created in the Java environment of the application and then sent into the JavaScript environment to be used in the code. The first object, data, is an object that refers to the current table you are viewing. This object gives users access to cell values for any row under any column, the number of rows in a table, and other information about the table's state. The second object, control, can query and open any table on the application. The JavaScript objects provide valuable functions so that users do not have to write substantial code to complete the desired tasks.

I made List and Detailed views for various datasets with these two objects. One of the first datasets I worked with was for a cold chain management use case. The set consists of a table with facility information and a table listing all the equipment at each facility. For its detailed view, I organized relevant details into sections for easy reading. I also added a list of checkboxes to visually represent whether or not the facility has certain features (specified by the values of the columns). To show all the refrigerators that are located at the facility, I queried another table with refrigerator data to search for all the refrigerators with the same facility code. I created a link to the results so that when clicked, the table of refrigerators (at the facility) opens.

For the List view, I tried to display the needed details for each row without overwhelming the list. Since each list item represents a facility (a row entry in the facilities table), the name of the facility was determined as the piece of information that would distinguish each list item from the others; therefore, the name is displayed in the main header. In order to incorporate the other details without crowding the List view, I decided to put the other information hidden under each list item header, shown only when the items header is clicked on. This way, details are accessible with one click and easily hidden (by clicking the header again) when unneeded. I used jQuery [2], a JavaScript library, to create sections that expand when you click on the item header (clicking on a selected item header will close the expanded section). When an items header is selected to expand the details, the items header ap-
pears red to signify that this item has been selected to show more information. If you click on the expanded area, it will open up the Detailed view of that row.

I also worked with other use cases that had columns for image or audio file names. For a Geotagging use case, it was particularly important to incorporate the images into the views since the purpose of the geographical data is to identify and describe the image. In general, it’s helpful to see a preview of an image that a row is associated with, so I displayed the image from each row as a thumbnail in the item headers. See Figure 6. Figure 5 is another design of the List view where I placed the images in the expanding sections along with the rest of the hidden details. For Detailed views, one way I incorporated the images was by using them as the backdrop. For datasets with audio files, I integrated the audio by placing a play button in the Detailed views (see Figure 8) which plays the file when clicked on.

Figure 5: Using images as thumbnails for list items.

Figure 6: Adding images to the expanding sections.

Figure 7: Using images as backgrounds.

Figure 8: Playing audio in Detailed views.

2.2 Addressing ambiguity

When a column of a table is indexed, the table displays only distinct entries under that column. Multiple rows with identical values under the column are grouped into a collection and the collection is represented by just one row. That representative row is simply the first entry of that group. The hidden rows can be viewed by long-pressing on the representative row and choosing to view that rows Collection view.

While experimenting with the custom views, I found that the context of the List views were evidently ambiguous when tables were indexed. The custom views do not clearly portray the state of the table or highlight the important features of an indexed table. For example, when the column Department from the facilities table is indexed, we only see rows with different department names and each row has hundreds of rows with the same department hidden under it. Essentially, we are grouping facilities by department. The problem with ambiguity occurs when we switch to the List view of this indexed table.

In the code, a list item is created for each of the rows, which represents entire collections of rows. Without indicating that the table is indexed and that there are hidden rows to be seen, creating the same list items as when the table is not indexed can be misleading. Figure 9 shows us what the List view of the facilities looks like when it is indexed by the department field. If we look carefully enough at each list item and notice that each of the items department is unique, we may make the assumption that the table is grouped by department, but it remains hard to tell. Also, using the first row of a collection to represent the entire group also contributes to the ambiguous display of collections.
There are a few things I thought were important to make explicit when the table is indexed. First, it should be clear that the rows are grouped in some way. It should also be easy to access collections. Taking these into account, I created the List view seen in Figure [10]. To implement this, I added methods to the data object that would determine if we are viewing an indexed table. JavaScript code can now call that method to test if the table is indexed; if it is, it displays an explicit message that says what the rows are grouped by. Furthermore, each list item doesn’t display the first row like it did before, but it displays the actual department that the collection of rows have in common. I also added a method that returns the number of rows in a specific collection. I displayed the size of each collection on a button which opens a table of all the underlying rows when clicked on. The buttons on the list items allow viewers to conveniently access collections without having to long-press and search for the option to view the hidden rows.

Figure 9: Facilities table, ambiguously indexed by department.

The same ambiguity arises when we view a collection in List view versus a normal table’s List view. If we compare Figure [1], the List view of the full facilities table, to Figure [11], the List view of a collection, the views appear similar at a glance. We can look closely and see that all the list items have the same department value. This could indicate that we are viewing a collection of facilities with that department value, but it is uncertain. These two tables are distinct representations of the tables data; one shows all the rows and the other shows all the rows with a common property. To distinguish the collection view from the view of a full, normal table, I added another method to the data object that returns whether or not the current table we are viewing is a collection. The code calls on this added feature and if the table is a collection, there is a message at the top of the List view stating the department all the rows have in common. This is shown in Figure [12]. This, in combination with the numbered list items more clearly shows that we are viewing a collection of rows with the same department.

Adding these new features just provides users a way to determine the state of the table so that they can customize their views appropriately. The messages, buttons, and numbering shown in the figures are how I customized the views for the cold chain use case. The exact way users decide to handle the cases in which the table is indexed or a collection view is completely up to them.

2.3 Code templates and documentation

Initially, there was one set of example HTML/JavaScript files, but there was no documentation on how to start building these views.
I created a readme file that explicitly defines the process for users. This file describes what the two custom views are, includes an outline of the objects and their methods, and introduces the code templates.

Users who are familiar with HTML/JavaScript may feel comfortable creating List and Detailed views from scratch, but having some outline of a List view and Detailed view may help users with less HTML/JavaScript background. I created code templates with important features implemented for users not only to make the process more convenient for them, but also to provide clear examples of how to use the JavaScript objects that are given to them. Although the partially completed code creates the general structure of the List and Detailed views, users need to edit the code with specific column names from the table they are assigning the files to. Figure 13 shows a small portion of the code template for a Detailed view. There are directions that help identify parts of the code that needs to be edited. The code shown in the figure models how to use the data object to retrieve data from a particular column. It also describes how to query a table with the control object. Code examples for other features that are useful for these views are sectioned off in Note sections of the template. If users want to implement the extra features, they can edit the example code and place it appropriately in their file. The templates for the Detailed view generally provides a lot of tips on how to query, link to other tables, include images, and accomplish tasks appropriate for a Detailed view.

The provided code in the List view templates creates a list item for each row of the table. The content of each item is up to the user, but it’s common to display a few fields from the table for each item, so there are examples of how to do so. This template also includes the appropriate tests to determine whether the table is indexed or in a collection view so that the task of making the views unambiguous is done for the users. Users may have their own way to interpret and present indexed tables and collections, so I included the explicit messages and buttons as suggestions.

These templates were designed to be a helpful guidance in creating custom views. They incorporate elements of the new design (implemented with jQuery) and the added functionality to the objects to provide users a foundation to build more effective and interesting views.

3 Related Work

Microsoft Access [3] is a database management application that provides the similar feature of allowing its users to create visuals of the data from its tables. One of its tools, Reports, formats and displays table data for viewing and easy printing. As a desktop application, Access provides an elaborate graphic interface for creating Reports which contrasts from the basic text editors that can be used to create custom views for this mobile application. On the other hand, Access offers its users tools that are comparable to the code templates and JavaScript objects that are provided to create the custom views for ODK Tables.

There are three ways to create Reports in Access. The AutoReport Tool is a button that displays all the fields from the table in a polished table and the Blank Report Tool allows users to start from scratch. The Report Wizard generates a report based on the users answers to various questions and offers extra options to apply different styles (e.g., background color, font, font size). The code templates I designed for the custom views are similar to the Wizard tool; the partially completed files generate List and Detailed views after the user replaces the placeholders in the code with specific column names. The templates also provide directions and example code to add extra features they may want in their displays such as images and links to other tables. Users can start from an empty file and create the views from scratch like the Blank Report Tool allows. But like the Wizard tool, these templates are provided to make the process convenient for the users.

Access users can use objects called Bound controls to display values from the table in the Reports. After identifying the field from which the control will get its data, the value from that column will be displayed where the object is placed in the Report. Data from other tables can be incorporated by creating queries and relationships to connect tables by common columns. Users of ODK Tables have similar objects at their disposal; they can use the JavaScript object data which provides a method called getData that will retrieve values under the
specified column for any row of the table. The control object can also join tables, open other tables on the application, and allow access to those tables data.

Access Reports are meant to suit many purposes. There are multiple ways to create Reports (e.g., AutoReport Tool, Blank Report Tool, Report Wizard), various environments in which to edit those Reports (e.g., Layout View, Design View) and many ways to group and display data in decorative arrangements. This interface makes Reports highly customizable in terms of format and design. While Reports can have various formats, the custom views of ODK Tables are designed to be list and detailed representations of the tables rows. Therefore, with fewer formats to support, the provided tools can be simpler and more focused on providing tools that help customize the content and appearance within each view (e.g., data, placement of elements, links, and images) rather than the overall structure of the display. While preserving the list and detailed format, I've focused on making the interface simple, convenient, and more equipped with useful customizing features.

4 Performance

The load time for List views were a particular concern compared to the load time for Detailed views because the size of the list changes depending on the size of the table. Before the new functionality was added to the data object, it was first implemented in the HTML/JavaScript code. The load time for the List views was noticeably long since the JavaScript code was making expensive calls on the objects to determine if tables were indexed or collections. To reduce the overhead of making those calls, I moved the functionality onto the Java side. All the logic to determine what state the table is in is now completed when the objects are created. The results can be accessed through new methods of the data object.

Figure 14 summarizes the load time of the List views with the new features (e.g., checking if the table is indexed or a collection and displaying the list accordingly). The List - 1 column refers to how long loading takes with the new functionality implemented in the HTML/JavaScript code. List - 2 refers to how long it takes when the code accesses
<table>
<thead>
<tr>
<th>Rows in table</th>
<th>Rows displayed</th>
<th>Load time (ms): List - 1</th>
<th>Load time (ms): List - 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>414.8</td>
<td>121.2</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>669</td>
<td>200.6</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>1,818</td>
<td>1,299.6</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>21,88.8</td>
<td>16,642</td>
</tr>
</tbody>
</table>

Figure 14: Performance Results. List - 1 refers to the code that implements new functionality. List - 2 refers to the code that calls on the objects for new functionality.

the new functionality through the objects from the Java side. In summary, load time did improve after moving the functionality and implementing it on the Java side. Load time particularly improved in the case where tables are indexed. When a table is indexed, each list item represents a collection and displays the size of that collection on a button. Previously, the code was calling the objects’ methods several times for each list item to determine the size of the collection each of them represented; moving this calculation to the Java side and having the object return the size of a specified collection proves to be much faster.

The improvement in loading time may not always be noticeable when switching views, but moving the implementation of these tests to address performance concerns clearly simplified the creating of List views for users. The code that calls the objects new methods has approximately thirty lines less code than the code that implements the logic in the JavaScript. This makes creating custom views more convenient for users since they can simply call the objects methods without having to know or write the logic behind the tests. These additions will help save users some time and effort.

5 Evaluations

I conducted surveys to evaluate how effective the enhanced interface and designs are in equipping users to create their custom views. I first gave a demo of the cold chain use case. I presented the facilities table in three views: the spreadsheet, List, and Detailed view. I also repeated these steps after indexing the tables to show users how the views change with the state of the table. I presented these in two sets; the first set was with the old custom view designs and the second set was with the new designs and added functionality.

The first set of questions were based on the demos that were given. The majority of individuals said the newer designs were very clear in indicating tables were indexed whereas they found the older set of views unclear. Some individuals mentioned that seeing the list items represent the general collection rather than the first row of the collection made more sense. When they were asked to specify what aspect of the new designs helped them determine that the tables were indexed, they said the messages at the top of the lists were most helpful since they clearly implied whether the table was indexed or in collection mode. By the end of the two demos, the general agreement was that the visual clarifications were very important to include.

All the surveyed individuals have more than a year of programming experience, but they separate into two general groups in terms of their background in HTML/JavaScript; members in the first group have less than a year of experience writing HTML/JavaScript and the members of the second group have more than a year of experience. Another portion of the survey asked volunteers to create a simple Detailed view. They were asked to complete a few basic coding tasks such as adding in a field from a row and querying a table. To
help them complete the tasks, they were given the readme file in addition to the code templates I created.

After the surveys, volunteers answered a series of questions about the tasks and the tools they were given to complete them. They gave ratings on a scale from 0 to 5 where 0 represents really hard or not helpful and 5 represents really easy or really helpful. Both groups rated the tasks level of difficulty between a 4 and 5 on average. Also, reviewing the code they wrote shows that all the necessary method calls and variables were used. Besides a few syntax errors, the tasks were all completed correctly. When they were asked how helpful the code templates were in completing the tasks, the group with less background gave a higher rating than the second group. This result was expected since the templates were particularly designed to be examples for those who are unfamiliar with HTML/JavaScript. The majority of each group also gave a rating of 5 for how easy they feel it would be to create their own custom views with the given tools.

The objects are used substantially in the views to perform tasks and retrieve data, so I asked the volunteers how easy it was to understand and use their methods. They rated each method on a scale of 0 to 5, 5 implying that the description, input, output and purpose of the method were clear. All of the methods were rated between 4 and 5 by the second group. The first group gave a few of the control objects methods a rating of 3 and rated the rest of the methods between 4 and 5. In their reflections, they mentioned that the methods of the control object required query strings which they were unfamiliar with. Some added that seeing examples of the methods being used in the code templates helped them figure it out. The average results are presented in Figure [15](#).

Feedback from the users suggests that the provided tools are easy to understand and use. All the users called the appropriate methods and used them correctly (besides a few syntax errors). Overall, the surveyed users demonstrated that the task of using the objects to retrieve and present data in the custom views is feasible for users with even basic HTML/JavaScript knowledge. Moreover, users expressed confidence in being able to build their own designs with the given examples, templates, and objects.

### 6 Conclusion

ODK Tables gives its users the ability to customize the mobile application for their unique use case. To better assist the users in creating their own information applications, I focused on implementing my own views to address obstacles and expose areas that can be improved. This paper has described some of the ways I recreated and improved the views to represent tables of different use cases. It shows examples of new List and Detailed views and also describes the additions made to the set of tools that are presented to users. Surveyed individuals express confidence in being able to create their own views with the resources given. With the additions and enhancements made, future users have a more complete interface that makes creating custom views simpler and intuitive.
References

