The FAIMS Mobile User to Developer Guide: How to Build and Deploy Field Data Collection Systems for Arbitrary Methodologies

Version 1.0.20180613

Dr Brian Ballsun-Stanton
Russell Alleen-Willems
Adam DeCamp

Funded by the NeCTAR V005 Documentation grant for RT043 (FAIMS Project).

Typeset in ConTEXt.

Code for this project and the latest version of this document can be found at http://github.com/FAIMS/UserToDev.

1 Forward

This document will teach individuals familiar with the FAIMS system of efficient, comprehensive data collection how to create their own modules. No programming experience is required; all you need is a computer, some idea how to use it, and patience.

Because this guide starts from simple explanations and builds up to more complex applications, it's important to understand a section completely before you move on. Keep an eye out for our Test Your Knowledge questions. If at any point you can't confidently answer one, it might be a good idea to back up and re-read.

Quiz 1.1 Test your knowledge

1. What does this guide teach you how to create?
2. What should you do if you don't know the answer to a Test Your Knowledge Question?

2 What is FAIMS?

As far as your team's day-to-day usage is concerned, FAIMS is a replacement for less precise or efficient data-gathering tools such as paper forms, notebooks, photo logs, and spreadsheets. In a more technical sense, it's a system that lets humans intuitively collect data and computers meaningfully organize it.

The basic feature the FAIMS system revolves around is the **module**. The module is the part you interact with and the part that manages the data. These are what you're going to learn how to make, and it's important to understand them fully.

2.1 What does the "module" do?

From the perspective of users (which in this case will include all members of your team who aren't responsible for the module's technical aspects), the module appears to be a digital form that can be accessed from an app and used to submit field data. What the user sees, what data the user are asked to provide, and even what kinds of data the user are allowed to submit are all functions of the module's design.

> **When you design your module, you will have to think about what you do and do not need from your team.**

More importantly, FAIMS modules act as databases that collect data submitted by your team while remembering when it was originally collected, by whom, and how the information is related. Regardless of how or when your team's data are collected, relevant context will be preserved and nothing will be accidentally discarded "saved over" by later findings. Not only does this provide a convenient and comprehensive centralized repository of your team's data, it means that team

members who collect data far outside the range of internet or mobile service can be confident that when they're able to upload their work won't impact how the data are ultimately presented.

2.2 Why should you use a FAIMS database?

Why not stick with older analog methods?

When projects rely on combinations of multiple data-gathering methods, ranging from basic paper forms to supplemental map notes, annotations, GPS readouts, and photographs, it eventually becomes necessary to organize the data into physical files, folders, and electronic databases. Even when the data can be organized centrally, so that everything can be found in the same place and format, important considerations of context–where something was found, when it was found, who recorded it, how it relates to other pieces of collected data–are frequently lost in the transition. It's all too easy to scan a photograph and fail to capture the date penciled in on the border, or enter a paper form's submissions into a spreadsheet and have nowhere to insert a margin note, or digitize a stack of records and lose track of which folders and archaeological contexts they came from.

The goal of FAIMS is to replace those diverse tools with one kind of tool, Android tablets, running modules through which all data are collected efficiently and compiled automatically and completely. There's no longer as much need for integrating separate cameras, notebooks, and forms; instead you can collect all those kinds of data on one device. In addition to simple preselected or text inputs, you can can record and directly attach video, audio, and other files such as PDFs and electronic sketches collected with your tablet.

Because your data are being recorded digitally from the start, you'll be able to mostly skip transcription once you get back from the field. In fact, a key advantage of FAIMS is that you can synchronize your Android devices with your main database at any time you have a

network connection, and then use the updated data to plan where to focus your next day's field activities - while still out in the field!

FAIMS modules can be exhaustively customized to suit your team's needs, and to an extent may even be altered and automatically updated to all devices even after your team's project has begun. As you test out your new digital modules, you'll naturally find places where you'll want to add features like taking photographs on the tablet, tracking GPS locations of each record, and including additional files such as videos in your digital documentation. Once FAIMS becomes embedded in your field recording, you'll also discover ways, usually through the feedback of your field crews, that you can optimize your modules to allow recording to proceed faster and more efficiently. For example, you may include tracking for individual FAIMS user accounts so that field technicians no longer have to worry about including their initials, automating time and date fields, and organizing fields so that they appear in the order of your recording workflow.

2.3 What are the benefits of using the FAIMS database over other databases?

While any database can be used to collect your team's data, FAIMS is designed with the specific needs and tools of archaeologists in mind.

For one thing, FAIMS treats the records you collect with an eye towards preservation. If you and another researcher enter in conflicting records, one of you doesn't "save over" the other; both of your contributions are preserved. Similarly, if a researcher alters a record, the project manager can still go back and see what it used to say. This can be particularly important when your team is collecting data in remote conditions and may not always be able to submit their findings in a timely manner.

Furthermore, unlike simple databases which only allow for text input, FAIMS accounts for all the diverse tools the modern archaeologist has at their disposal, including video, photographs, audio logs, and GPS

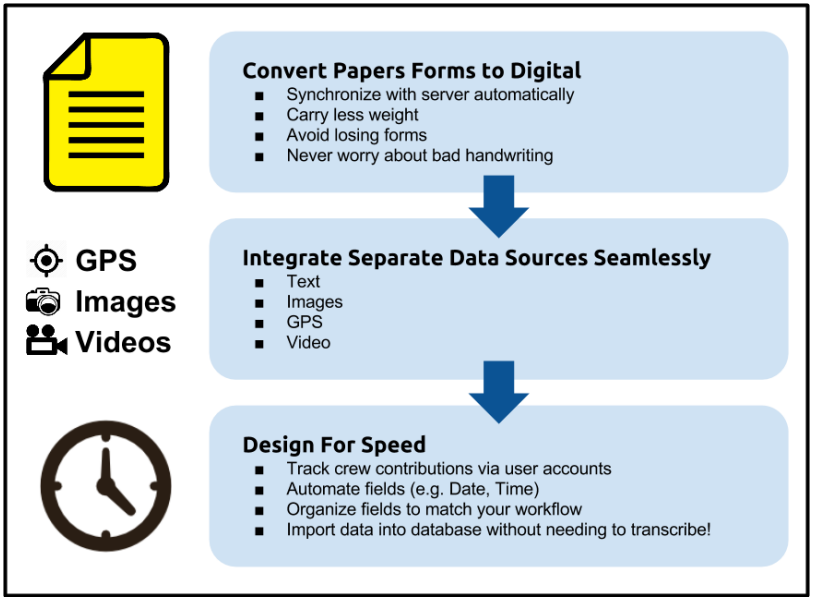

Figure 2.1 A prototypical workflow for a FAIMS Mobile module.

tags. You can attach these directly to your digital forms and store them seamlessly along with the rest of your collected data.

2.4 What are the benefits of using the FAIMS database over a spreadsheet?

The obvious reason is that spreadsheets are clunky and not really designed for records-keeping. Even if you put a painstaking amount of effort into creating a custom form for you and your team to use that clearly outlines where data should go and what form it should take, it's very easy for someone to accidentally undo that work by hitting the wrong key or saving at the wrong time. Furthermore, if you ever want to look at an earlier version of your record, you'll need to have been saving multiple redundant versions.

But there's a more important reason, and it has to do with the difference between how humans like to format data and how computers have to interpret it. This reason is technical, but may be useful to understand once you begin formatting and designing your own modules.

Let's say a researcher turns in a typical data-collection spreadsheet like this:

SITE:	**SITE GROUND COVER:**	**ANIMAL:**	**COLOUR:**
Site A	Grasses	Bird 1	Red and White Wings
			Blue Plumage

Table 2.1 A sample spreadsheet table presenting complications with contextual whitespace.

We can immediately understand a lot about this spreadsheet from reading it. That's because we, as humans, can intuitively grasp things about it that a computer must be explicitly told.

We know that "Bird 1" is located somewhere called "Site A." We also know, without having to think about it, that a location like Site A can have multiple birds that belong to it or no birds at all. We can think of such birds we find there as instances of data (or a "record") which belong to another "record" called "Site A". Because there is a central piece of data to which zero or more pieces of data are hierarchically structured beneath, the relationship between Site A and Bird 1 is called a *parent-child relationship.*

We know also know from observing the spreadsheet that Bird 1 has red and white wings as well as blue plumage. Now, Bird 1 cannot have "zero or more" color in its wings or "zero or more" color to its plumage. It will obviously have a definable color for both, and the form expects this color to be described in straightforward terms. Similarly, the site will obviously have definable ground cover; the only question is what the ground cover looks like. We call these *attributes of an entity.*

Because of our unspoken knowledge of these factors, we are able to make clear intuitive sense of the spreadsheet. The problem is that forms that make sense to humans often don't make sense to computers. This form, for example, is not digitally parseable. All a computer knows looking at those six cells is that at Site A there is a Bird 1 with Red and White Wings, but the quality of Blue Plumage has no connection to any other data. The attribute has no defined entity. The computer knows that something is Blue, but has absolutely no way of inferring further information from context, as humans invariably do when reading *and compiling* data.

When demonstrating *parent-child relationships* and *attributes of an entity* in a spreadsheet, inevitably human users will format data in a way a computer won't be able to meaningfully parse or store. The structure of FAIMS prevents natural human tendencies from making data unclear or digitally unmanageable.

2.4.1 Some framing questions for FAIMS Mobile

- How does FAIMS handle multiple users collecting data at the same time?
  FAIMS has no problem handling simultaneous usages or submissions. When two users create, edit, or delete records at the same time, FAIMS automatically takes note of who uploaded it and when and preserves both in the database. (Though if the two users are working without talking to each other, the server will raise a "flag" for someone to review the submissions.)
- How does FAIMS handle international teams and multiple languages?
  We'll explain more in a section below, but when you design your module, you can include a translation file that will allow users to choose between differently-worded forms.
- How is data reviewed?
  If you just want to quickly review your data, and not export it to a format such as shapefile or json, you've got two options:

1. You can navigate to your FAIMS server and use the Record View feature to look through individual records your team has made; or,
2. using your device, you can navigate using your module to "table views."

2.5 Exporters: how to share data with others.

To get the data in a viewable, usable fashion, you'll need to find and download a type of program called an exporter. What form you want the data to be in decides which exporter you'll want to use. You can find exporters for formats like shapefile and json by visiting github.com/FAIMS/ and searching for the correct program. Look for something called "(x)Exporter or (x)Export," where (x) is the desired end format (e.g., "jsonExporter" or "shapefileExport").

On a PC, you can simply download the file from github. If you're using your UNIX virtual machine, you can do so by entering at the command prompt in Codeblock 2.1 (denoted by $. Don't copy the $).

```
$   git clone https://github.com/FAIMS/shapefileExport/
```

Code 2.1 A command to use the program "git" to make a copy of the remote code onto your computer

...with the name of whatever exporter you want, in this example shapefileExport, in the final position.

Once you've got the exporter program, you're going to put it in a usable form. To do that with a PC, create a tarball from the exporter using a program like 7zip; if you're using UNIX, enter something like Codeblock 2.2.

```
tar -czf shapefileExport.tar.gz shapefileExport/
```

Code 2.2 A command to: tell the `tar` command to compress, gzip, and save the folder "shapefileExport" to the file `shapefileExport.tar.gz`

Now, if you navigate on the server to your module, you'll see a tab at the top labeled "Plugin Management." Click that and you'll be brought to a page with the handy feature, "Upload Exporter." Choose the tarball you've just created and hit "upload." You now have an exporter permanently stored to your FAIMS server and may make use of it whenever you'd like.

From now on, whenever you'd like to use your uploaded exporter, navigate to your module from the main page on the server and click "export module." Select from the dropdown menu the exporter you'd like to use, review and select from any additional options, and click "export."

You'll be brought to your module's background jobs page while the server exports your data. After a few moments, you should be able to hit "refresh" and see a blue hyperlink to "export results." Clicking that will allow you to download your exported data in a compressed file.

Exporting data doesn't close down the project or prevent you from working any further on it, so feel free to export data whenever it's convenient.

Quiz 2.1 Test your knowledge of some of the fundamental concepts of FAIMS Mobile

1. Once you've set up your FAIMS module, will it be possible to alter it to suit the needs of your team?
2. Can you use FAIMS to collect audio or video files as part of your field data?
3. What's the difference between data as stored by FAIMS and data as stored by a spreadsheet?
4. If two researchers enter data at the same time, does one of them "save over" the other's work?
5. Can you view collected data via your tablet?

3 Making Your Own Modules

In the next chapter, you're going to make your module by following these steps:

- Learn what the components of a module are and what they do.
- Download tools that allow you to create module components (or "necessary files") based on a set of instructions.
- Learn how to write those instructions so that the necessary files you produce will fit together to assemble the module you need.
- Learn how to set up and operate the tools so that they'll follow your instructions and make the necessary files you need.
- Use software to hunt down and correct any mistakes you've made.
- Send the necessary files to the FAIMS servers and create a module you and your team can download and use.
- If you need to modify parts of your module, create new necessary files and send them to the FAIMS server. They'll replace the old ones and allow everyone on your team to update to the new, improved version.

3.1 The Parts of a Module

Modules are assembled by your FAIMS server from components called "necessary files"[1].

Speaking practically, most "necessary files" are text files. Unless noted, they tend to end with the file extension `.xml` and can be opened and even edited with simple text editors, such as Notepad (although you may want to find something with a few more features, as we'll explain in the next section). Each necessary file serves a definite and distinct function in the final operation of the module.

[1] Brian's note: In our academic papers about modules we will sometimes use the term "Definition packets." The module is the product of the processing of the "necessary files" which as a set comprise the "definition packet".

Your team members can use a module without ever learning exactly what necessary files are or what they do, but you'll need to become familiar with them if you plan to make a module or alter one already in use. Here are the kinds of necessary files you'll come across working with FAIMS.

3.1.1 Data Schema

This file, which should appear on your computer as `data_schema.xml`, defines what kinds of data you want to record and how they're related to each other. We go into a little more technical detail about what a Data Schema is and does in the section below, "Tour from the Data Schema".

The Data Schema is one of the most fundamental and important necessary files of a FAIMS module. Unlike other necessary files, which can be replaced and updated even after the module's in use by your team, the Data Schema cannot be replaced. If for some reason your Data Schema no longer provides satisfactory results, you'll need to create a whole new module and instruct your team to transition over to it. This is the one part you should be absolutely certain you're happy with before you proceed.

3.1.2 UI Schema

The User Interface (UI) Schema, or `ui_schema.xml`, defines what your module will actually look like and where your users will input their data.

3.1.3 Validation Schema

The Validation Schema, `validation.xml`, defines what kinds of data your team *should* be collecting. It allows the module to "validate," or give a thumbs-up or thumbs-down to, your team's submissions to make sure the data being collected is thorough enough or makes sense.

For example, let's say your team is submitting data on handaxes they've excavated from a particular context. To complete your research goals you need to make sure that every time someone records a handaxe, they report how much it weighs in grams.

When you're designing your module, you will utlimately create a Validation Schema that ensures users must:

a. put something in "Weight of Handaxe" instead of leaving it blank
b. enter a number, not a phrase
c. list the weight in grams, not pounds, tons, or catties.

If a module checks a submission and discovers it isn't valid, it alerts the user who made the error, flagging the incomplete or problematic field as "dirty" and giving the user some idea what the problem is. However, the data are still collected and can be viewed or modified by the project manager.

3.1.4 UI Logic

The UI Logic performs a few functions. It tells a module's user interface how to behave, governs operations on the database, and facilitates interactions between FAIMS and devices such as GPS receivers, cameras, and bluetooth-compatible peripherals.

For example, when a record is created, the user who created it and a record of when it was created (also called a "timestamp") are automatically stored in the database. A UI Logic program can be used to 1) query the database to retrieve either of these points of data, and; 2) update the UI to display the retrieved data to the user.

3.1.5 Arch16n

You won't necessarily need to mess with your module's Arch16n file. It's there to allow you to provide synonyms and translations for the "entities" in your module–useful if some of your team members speak

a different language or use different terminology, in which case they would have their version of the module translated automatically.

3.1.6 CSS

The UI Schema defines the basic layout of your module's user interface, but the details, like how the entry fields and controls appear, are defined by the Cascading Style Sheet (CSS). You set these styles using the `ui_styling.css` file.

3.1.7 Picture Gallery Images

This part you handle more directly. Simply sort your images into folders, then put them all in a tarball (see "How do I share data with others?" for instructions). You can upload the tarball to the module generator directly, same as any other necessary file.

Quiz 3.1 Test your knowledge of some FAIMS jargon.

1. If data entered by a user isn't valid, is it collected?
2. What *necessary file* cannot be altered once a module has been created?
3. Which *necessary file* do you need to worry about if some of your team speaks English and some only French?

3.2 Tour from the Data Schema

We've already explained that the Data Schema describes what data your module is going to store–in other words, what your team is going to record and what you'll be able to export and analyze later. We've also explained that it defines the basic structure of the module and can't be altered once the module's been uploaded to your FAIMS server.

Because the Data Schema and its structure are very important to FAIMS, it may be useful (and for advanced coding, will be necessary) to understand what exactly is contained within the Data Schema.

Unlike the rest of the document, you don't need to understand this next part fully before you continue on to the next section. Instead, revisit this section every so often as you master a new portion of the User to Developer document and see how well you can apply your understanding of modules and their structure to this information.

A Data Schema contains "elements," or individual components that define some part of how the module works. The two kinds of elements you can find in a FAIMS Data Schema are called "Archaeological Elements" and "Relationship Elements," and they each do very different things to allow computers and users to collect and organize data.

3.2.1 Archaeological Elements

Archaeological Elements[2] are exactly what they sound like: they define an individual piece of archaeological data which can be collected. Each Archaeological Element has "property types" which define exactly what it represents and what kind of data are collected with regards to it. They can contain multiple property elements, such as a name, value, associated file, picture, video, or audio file, as well as a description.

[2] We will also use the term "Archaeological Entities or ArchEnts to refer to these. The term "Element" refers to *this specific file's definition* while entitiy refers to the thing living and working inside the module. Don't worry about it, but know that we use both terms."

Codeblock 3.1 below is an example of an Archaeological Element for birds located within areas A, B, C, and D. Again: you don't really need to understand this yet, but it won't hurt to familiarize yourself with its structure.

3.2.2 Relationships

Archaeological Elements contain a lot of information about what is being collected, but they don't actually define how the data being collected is organized or related. They can be used to say "users enter their location" and "users identify what they found," but not "users identify what they found IF they are in a certain location." In other words, Archaeological Elements explain how data is collected, but not how it is organized or related.

This is why the Data Schema also has something called a Relationship Element, a kind of element that describes how archaeological elements relate to one another; whether they are in a hierarchy, one contains another, or they are bidirectional. Relationship element can also contain child elements with more information about the relationship. These child elements can include descriptions, definitions of the parent and child entities, and multiple properties, such as "lookup," which lists controlled vocabulary terms.

An example of a Relationship Element can be found in CodeBlock 3.2[3]:

[3] Brian's note: While I built relationships to hold data all the way back in FAIMS Mobile 1.0, no one ever used them for that purpose. For now, we use them link Archaeological Elements usually with a directed 1:1 relationship.

```
<ArchaeologicalElement name="small">
    <description>
        A bird located within the area.
    </description>
    <property type="string" name="entity" isIdentifier="true">
    </property>
    <property type="string" name="name">
    </property>
    <property type="integer" name="value">
    </property>
    <property type="file" name="filename">
    </property>
    <property type="file" name="picture">
    </property>
    <property type="file" name="video">
    </property>
    <property type="file" name="audio">
    </property>
    <property type="timestamp" name="timestamp">
    </property>
    <property type="dropdown" name="location">
        <lookup>
            <term>Location A
                <description>This is the first location.
                </description>
            </term>
            <term>Location B</term>
            <term>Location C</term>
        </lookup>
    </property>
</ArchaeologicalElement>
```

Code 3.1 A demonstration archaeological element or “Archent”.

```
<RelationshipElement name="AboveBelow" type="hierarchy">
    <description>
        Indicates that one element is above or below another element.
    </description>
    <parent>
        Above
    </parent>
    <child>
        Below
    </child>
</RelationshipElement>
```

Code 3.2 A demonstration Relationship element or "Reln" allowing for a spatial relationship to be defined between two Archaeological Elements, one "above" the other.

You'll understand more about how these elements are structured and what they mean after a thorough reading of "Coding Modules." In the meantime, as long as you understand in basic terms what's contained within a Data Schema, it's safe to proceed.

Quiz 3.2 Test your knowledge of element types

1. Which of the following are real types of element contained within the Data Schema: Archaeological Elements, Data Elements, Module Elements, Relationship Elements

4 Setting up the Development Environment

While it would be wonderfully exciting[4] to be able to jump right into module development, we must first get your computer ready to handle the "excitement".

Before you can code any new modules, you'll need to download some files and free programs. Some of those programs you'll need to configure to make coding new modules easier and more efficient. At the end of this chapter, you will be able to:

- Run the FAIMS Server, where your team members will download your module from and upload data to, on a virtual machine.
- Examine all the necessary files of a FAIMS module with a text editor.

4.1 Deploying the Virtual Machine

4.1.1 Hardware Assumptions

Every computer is different, and unless you're using exactly the machine we used when writing these instructions (an HP laptop with a Core i7 processor and 8GB of RAM, running Windows 7) you probably won't be able to follow each step exactly as it's written. Many of the differences will be very minor; an option might have a slightly different name or be located somewhere else on your computer, for example. When you find an option isn't present as we describe it, take a deep breath. Click around or use your computer's "search" feature to see if you can find the option or setting somewhere else. If necessary, web search the terms we use in our instructions along with the system you're using in

[4] Brian's note: Exciting for some folks.

order to find equivalents. A good rule of thumb for general computer configuration is: whatever you can't figure out right now, someone else has had the same problem and a third person has fixed it for them.

As a last resort, we offer technical support services. See the appendix for further information.

4.1.2 Before You Start

Before starting, you'll need to make sure you have enough room on your hard drive. We recommend about 25GB minimum for the server installation, as well as enough RAM to hold both your current operating system, the emulated machine, and any open programs in working memory. If you don't have that much space, delete files (especially large media files, like unnecessary videos) or uninstall programs until you're ready to begin. No good will come of trying to follow these next steps without enough room to work.

4.1.3 Installing the FAIMS Server and Virtual Machine (VM)

For various reasons, the FAIMS server is designed to run on a machine running an operating system called Ubuntu Linux 14.04 (as opposed to, say, Windows or OSX). You probably don't use Ubuntu, and you probably wouldn't like to use it all of the time. That's why our first task here is to set up a "virtual machine" on the computer you do use. Virtual machines let you emulate entirely different kinds of computer system without making any significant changes to your own–in this case, an Ubuntu Linux 14.04 machine. It's like you're installing an application that simulates an entirely different computer whenever you need it.

4.1.4 Enable VM extensions in your computer BIOS for better VirtualBox Performance (VT Hypervisor)

You'll have a much easier time running your virtual machine, as well as Android emulators you'll need later, if you first enable a feature specifically designed for this purpose called "VT Hypervisor".

To enable VT Hypervisor, enter your computer's BIOS or UEFI menu while booting your computer. This process differs a lot from computer to computer, but you should be able to find instructions here: http://www.howtogeek.com/213795/how-to-enable-intel-vt-x-in-your-computers-bios-or-uefi-firmware/

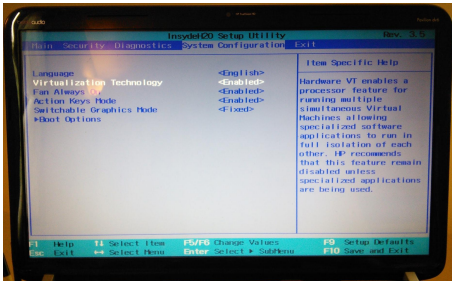

Figure 4.1 A rather literal picture of a computer's "BIOS" setup screen, showing the enabling of "VT-X extensions".

We found the VT setting under the "Virtualization Technology" option in the System Configuration menu. Again, unless you're using an HP dv6qt Laptop, yours will probably be somewhere else. This process can feel a little intimidating if you're not used to messing with your computer on this basic a level, but relax: this step is perfectly safe.

4.1.5 Installing VirtualBox

Now you can download and install the VirtualBox client from: https://www.virtualbox.org/wiki/Downloads. This is what we're going to use to

create a virtual machine that can run our Ubuntu Linux-based FAIMS server.

You probably won't have installed VirtualBox before, but on the off chance you have (say, if this isn't your first time trying this step) make sure you uninstall the old version **completely** before trying to install it again.

During installation, you may receive a couple Windows Security questions about Oracle drivers. Windows is naturally suspicious of this kind of installation, but nothing you're putting on your computer right now is dangerous. You can ignore each prompt individually or skip them all at once by selecting "always trust Oracle" in the popup window (Figure 4.2).

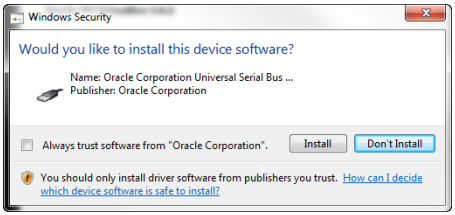

Figure 4.2 We are installing network device drivers here, so that your tablet can talk to this "Virtual Machine".

Once VirtualBox is installed, you can start the program. The default screen should look similar to Figure 4.3.

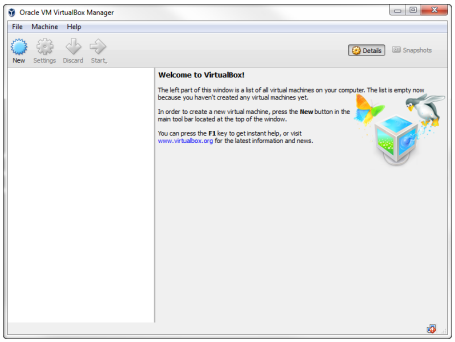

Figure 4.3 VirtualBox without any virtual...boxes. A box inside your computer which can hold boxes[5], but not cats.

If you need to take a break, this is a good stopping point[6]!

If you're ready to move on, download the FAIMS Server image (2.5GB) from: `FIXME`

The server image is packaged as an `.ova` file, which is a type of file that can be opened by VirtualBox. This file is the complete package: it represents an Ubuntu Linux 16.04 computer system that has the FAIMS server set to automatically configure and run. If you would like a server prepared for you, contact `support@fedarch.org`.

Before using the image, you'll need to unzip the compressed zip file using your favorite archival program (e.g. *7-ZIP*, *ICEOWs*).

[5] We call a computer a box because the old fashioned tower that older or powerful computers come with is usually referred to as the Box by tech types

[6] Time for coffee! It's *always* time for coffee.

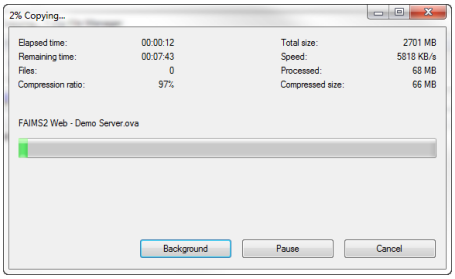

Figure 4.4 Unzipping the faims ova.

To set up the emulated machine, we'll use the preconfigured machine image we downloaded in the last step.

In VirtualBox, select the `File->Import Appliance` option.

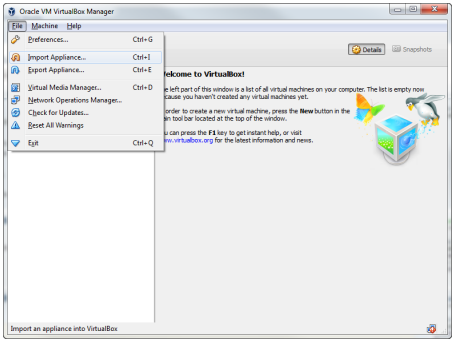

Figure 4.5 Or use the "ctrl-I" keyboard shortcut.

Using the file browser that appears, navigate to the directory where you downloaded the FAIMS server image and select to open it.

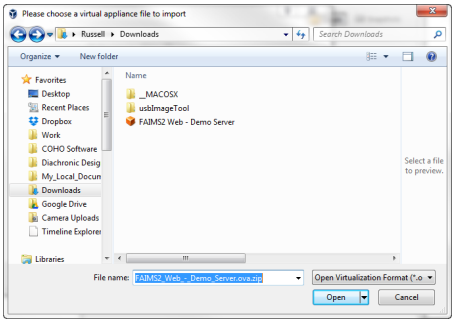

Figure 4.6 Opening "FAIMS2 Web - Demo Server".

The pre-configured settings will appear. Select the "Import" option.

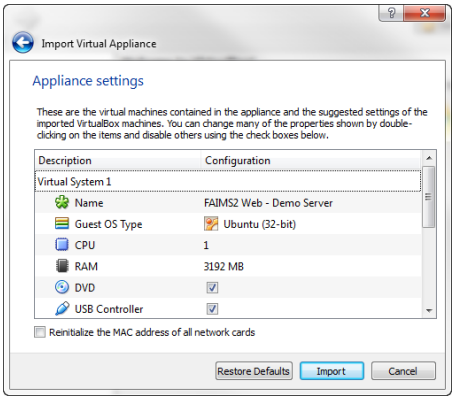

Figure 4.7 Import pre-configured settings.

VirtualBox will now set to work importing the image and setting everything up. Depending on your system, this may take a few minutes.

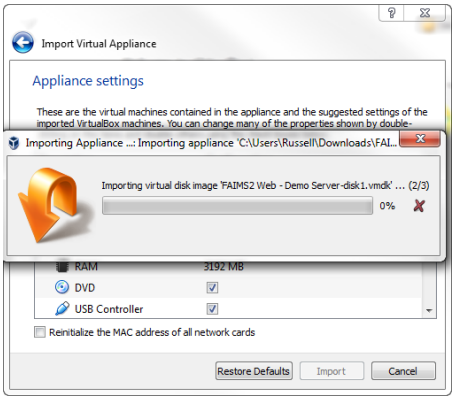

Figure 4.8 Waiting time!

Once the setup is complete, "FAIMS2 Web - Demo Server" will appear in the menu on the left in VirtualBox.

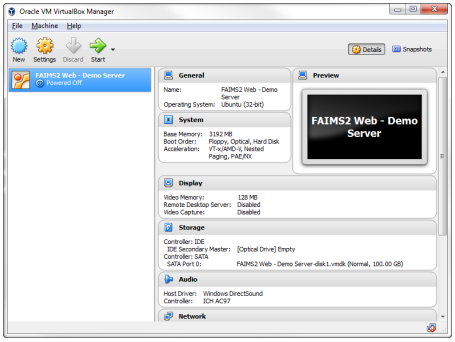

Figure 4.9 Click the green arrow to start.

Click on the system and then click the "Start" button in VirtualBox to start the server. You *will* encounter the following error (Figure 4.10).

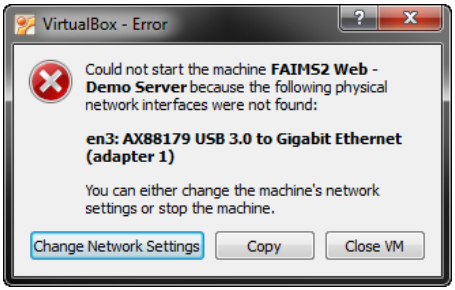

Figure 4.10 An expected error.

DON'T PANIC.

Simply, click the "Close VM" button. You're still on the right track; sometimes, things just have to be a little fussy.

Back on the main screen, click on the "Network" section.

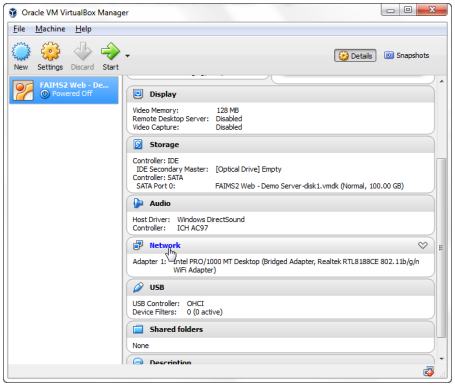

Figure 4.11 Scroll on the right hand panel and click "Network".

In the popup window that appears, check the "Enable Network Adapter" box on the "Adapter 1" tab, and select "Bridged Adapter" from the "Attached to:" menu.

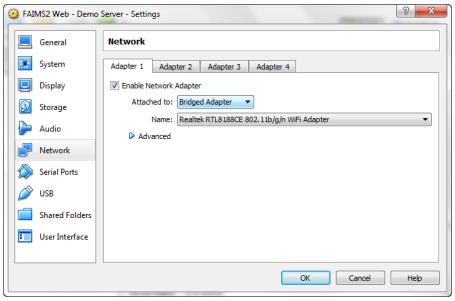

Figure 4.12 You may choose your machine's wifi adapter but, if you run into any issues, try using your hardline connection instead.

Click the "Start" button on the VirtualBox program. VirtualBox will pop up a new window in which your emulated machine will run. Give VirtualBox a few minutes to start your new emulated Ubuntu machine, and a few minutes longer for Ubuntu to start the FAIMS server. When they've finished, you will see the following message about the server having been set up (Figure 4.13).

Figure 4.13 Click "start" and wait for this screen.

If you like, you can skip ahead a little and, back on your real machine, you can open the FAIMS server by navigating to 192.168.1.121, or whatever address is listed by your FAIMS status page in your web browser (check your messages for the specific IP address on your machine). **Note:** The default FAIMS account is `faimsadmin@intersect.org.au`, password `Pass.123`.

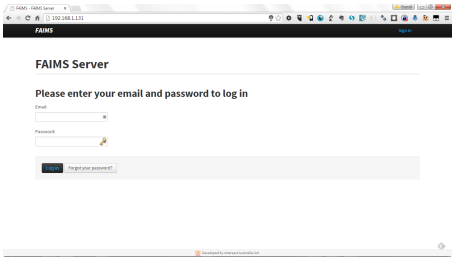

Figure 4.14 Sign in with the above credentials.

If you log out of the Ubuntu installation at any time, just use the password `Pass.123` to log back into the system.

Quiz 4.1 Test your knowledge about the VM

1. If the FAIMS server is supposed to be run on an Ubuntu machine, how are you going to run it on your non-Ubuntu computer?
2. What can you do if you can't find an option or setting we refer to in our instructions?
3. If you've installed VirtualBox before, what do you need to do before setting it up on your computer?

4.2 Finding the right Text Editor for your

The purpose of the FAIMS tools are to remove the need to individually craft all of your module's necessary files. Instead, you'll need to create

one file the tools can use as a blueprint for making everything else. In order to make that one file (and to make any fancy edits to the necessary files to add detail or functionality), you're going to need a simple text editor that's good for coding with.

If you're doing your coding within the virtual machine, we recommend using an editor called gEdit which came with your Ubuntu installation. It's simple, functional, and won't clog up your instructions with unnecessary formatting like a standard Word Processor[7].

If you decide to write the files outside your virtual machine, we don't recommend using the text editors (such as Notepad) that come with Windows or OSX. While they may not complicate your code like a word processor would, they also don't have little features like autocomplete (a feature that means you don't need to fully type out words you use often) and syntax highlighting (a feature that helps you keep track of how your code is structured with helpful visual cues). With a little searching around, you can find plenty of excellent free or commercial text editors.

We particularly recommend:

- *Notepad++* (Windows)
- *Atom* (Cross Platform)
- *TextWrangler* (OSX)

For each of these, download their installers and follow their installation wizards. If you're using a Linux device that isn't the virtual machine, you can install gEdit, Vim, Emacs, or another text editor using apt-get.

4.3 Access to local files via VirtualBox

[7] Note from your friends at FAIMS: One of the main reasons to use the text editors above is so that you can avoid "Smart Quotes," an automatic feature of many word processor programs that will insert differently styled or extra quotes that will cause the FAIMS server to reject your module. If you run into any issues with your module, ensure that "Smart Quotes" are disabled in your text editor.

4.4 Installing the Android SDK

Now we'll need to install the Android Software Development Kit (ADK). It's available at `https://developer.android.com/sdk/installing/index.html` Ignore the Android Studio link; we just need the SDK Tools.

Because you're going to use the tools for debugging an Android device plugged into your actual computer, you're going to want to download and install SDK Tools to your regular computer system, NOT your Ubuntu virtual machine.

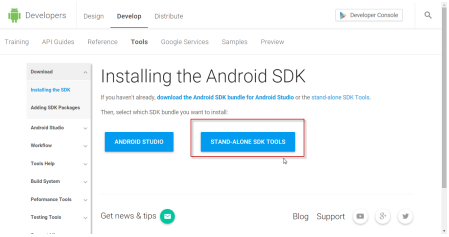

Figure 4.15 Click Stand-Alone SDK Tools.

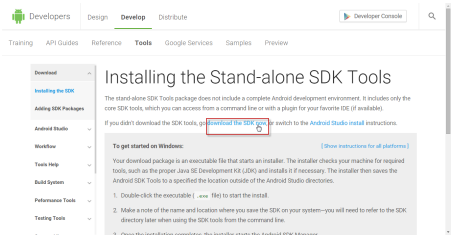

Figure 4.16 Choose to download the SDK Now.

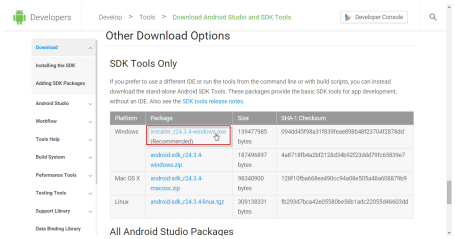

Figure 4.17 Make sure to get the SDK tools only. We won't need to compile a new Android application here.

Run the downloaded installer and follow its prompts to finish setting up the SDK tools.

4.4.1 Installing the FAIMS Debug APK on your Android Device

The final step to setting up your development environment is to install the FAIMS app on your Android tablet or phone. Instead of installing the app through the Google Play Store, navigate to `https://www.fedarch.org/apk/faims-debug-latest.apk` and download the special debug version.

Next, you'll need to place the FAIMS APK file on your phone, either by moving it over via USB connection or by placing the APK file in a cloud service like Dropbox and downloading it through the mobile app.

If you load the URL `https://www.fedarch.org/apk/faims-debug-latest.apk` on your Android devices, you'll be able to directly download the file.

To install the APK, you may need to change the security settings on your device to allow non-market apps. Go to the "Settings" app, find the Security settings, and check "Unknown sources" (device pictured in Figure 4.18 is a Samsung Galaxy s5. Your settings menu may look different).

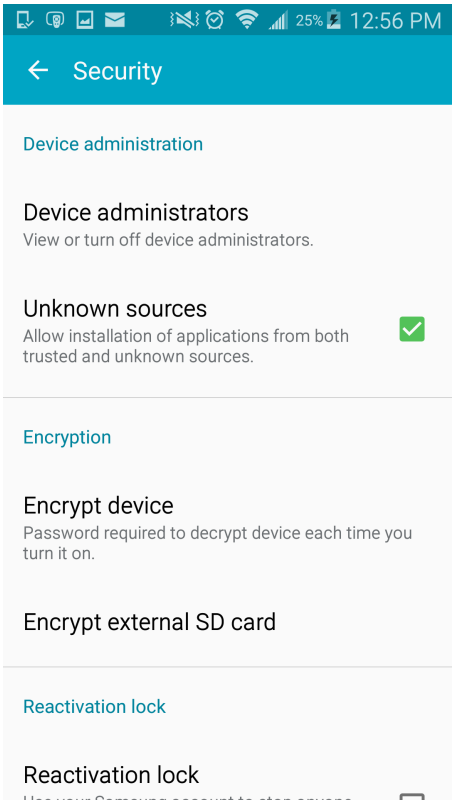

Figure 4.18 Change your security settings to allow "Unknown sources".

4.4.2 Making your Android Device Communicate with your Computer with USB Debugging

First, you'll need to enable the "Developer Options" on your device. This is another step where the instructions are different depending on what kind of device you're using.

On Android versions 6+ to enable the "Developer Options":

1. Open the App drawer.
2. Launch Settings menu.
3. Find the open the “About Device” menu.
4. Scroll down to “Build Number”.
5. Next, tap on the “Build Number” section seven times. (Yes, really.)
6. After the seventh tap you will be told that you are now a developer. (Again, this is really how it works.)
7. Go back to Settings menu and the Developer Options menu will now be displayed.

You will need to install (for some samsung models) specific debug drivers:

Note that if you want for USB Debugging to work when your Samsung device is connected with your computer, then you will first need to make sure that the Samsung USB Drivers are installed on your PC.

http://www.android.gs/download-samsung-usb-drivers-for-android/

For many non-Samsung devices, you can use Google’s ABD/USB driver instead: https://developer.android.com/studio/run/oem-usb.html#InstallingDriver

In order to enable the USB Debugging you will need to open Developer Options, scroll down and tick the box that says “USB Debugging”.

That’s it! Your FAIMS development environment should now be set up properly.

Quiz 4.2 Test your knowledge of all of FAIMS needed support tools.

1. What are two things you can do if you don't find an option or setting where we tell you it will be?
2. If the "Error: Network Device Not Found" error appears when you start VirtualBox, how do you fix it?
3. True or False: Microsoft Word is a good program to code your modules in.
4. Should you install your Android developer kit within the virtual machine? Why or why not?
5. Which of the following are true?
   - VirtualBox allows you to run a different operating system on your computer
   - VirtualBox connects to a computer at the FAIMS offices
   - You must install XSLT Proc and saxon-xslt before running the XML generator tool
   - FAIMS can use both Android and Apple devices
6. Exercise: Open up a command window. In this command window, write `adb --version` then enter. Some information about the Android Development Bridge (ADB) program should appear. Did you encounter any errors? If not, congratulations, ADB/Android SDK is successfully installed.

5 How to Code Modules

For this tutorial, we'll work together and create a simple FAIMS module. For illustrative purposes, we're going to recreate a simple module available from the FAIMS demo server–the "Oral History" module.

We're going to start by showing you how to make a basic but fully functional version of the module that can be made only by constructing a relevant `module.xml`. Then, after you've run that through the FAIMS-tools and produced a fully functioning but simple version, we'll teach you some ways to add complexity and functionality by modifying the necessary files directly.

5.1 Introduction to `Module.xml`

Earlier sections introduced the basic necessary files of a module. You should already have some idea what these were:

1. a Data Definition Schema (`data_schema.xml`).
2. a User Interface Schema (`ui_schema.xml`)
3. a User Interface Logic file (`ui_logic.bsh`)
4. a Translation (Arch16n) file (`faims.properties`)
5. A server-side validation file (`validation.xml`)
6. CSS Files for styling the modules (`ui_styling.css`)

You've already learned that you don't have to design and code all of these files from scratch. Instead, you'll create one file that serves as a set of instructions for the FAIMS Tools to create the necessary files for you. That one file is called `module.xml`.

5.2 The Module Creation Process

Before getting into the nitty-gritty specifics of how to structure `module.xml`, it's useful to review what the overall process of creating a module will look like. Below is an overview of the steps you'll follow.

Quiz 5.1 Test your knowledge: a gentle descent into `module.xml`

1. What are some programs you can use when modifying `module.xml`? Which programs should you avoid? If you don't remember, review the previous section, "Setting Up Your Development Environment."

Back on your Ubuntu install in the virtual machine, open up a Terminal window using the Ubuntu Dash program. You can read how here: *https://help.ubuntu.com/community/UsingTheTerminal#In_Unity*.

The Terminal is Ubuntu's command line tool that we'll use to navigate the file system and run commands and programs, so this is probably a good time to make sure you're acquainted with it before proceeding.

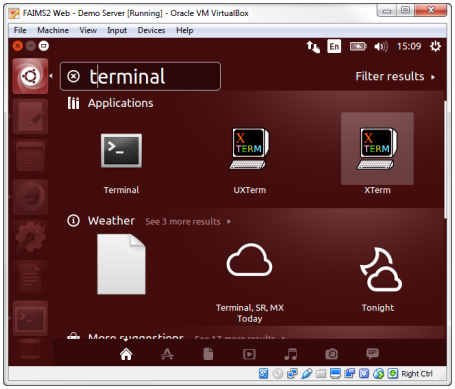

Figure 5.1 How to open the terminal using Ubuntu.

Start your chosen text editor and open the `module.xml` file. If you're using the text editor we recommended for the virtual machine, you can simply get it from the directory where you installed the FAIMS-Tools (usually `FAIMS-Tools/generators/christian/module.xml`). If

you're using a text editor on your main machine, you'll need to export `module.xml` outside the virtual machine and work from there.

Before you make any edits, it's a very good idea to save an extra copy of `module.xml` file as `moduleTemplate.xml`. The version you're going to develop your module in and run through FAIMS-Tools needs to be named `module.xml`, but if you want to do any other module development in the future you're going to want to have a blank lying around to work with. Once you've made your backup copy, you don't have to worry about messing around in `module.xml`.

Once you've finished coding, editing, and troubleshooting `module.xml` (and don't worry; we'll explain how to do all of that in the coming section), the instructions are complete and you're ready to generate your module. From the same directory you originally found `module.xml` in, run the commands in Codeblock 5.1 from the command line terminal. You won't have to type the rest of a command: just hit tab and the terminal will try to guess[8].

```
$   cd ~/FAIMS-Tools/generators/christian
$   ./generate.sh
```

Code 5.1 BASH shell commands to run the generator.

The `./generate.sh` command will look through `module.xml` and create the necessary files that the FAIMS server needs to create a module.

Navigate to the newly created "module" folder and you'll see that the script created 6 new files. That wasn't so hard, was it?

[8] The first command could be typed with fewer characters `cd ~/F<tab>g<tab>c<tab>m<tab>)`

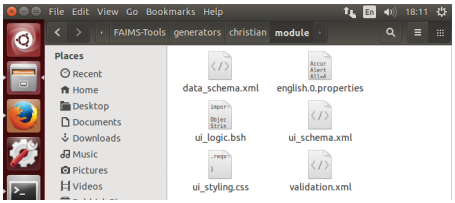

Figure 5.2 The files generated by the generator.

In the web browser, on your Ubuntu installation, open up the FAIMS server and select the "Modules" tab. On the "Modules" tab, click on the "Create Module" button.

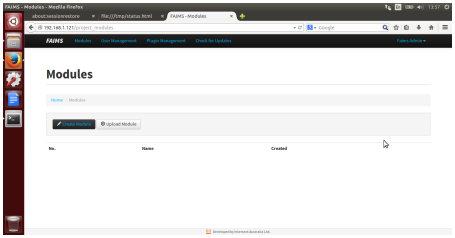

Figure 5.3 Modules page on the FAIMS server.

In a web browser, load up the web interface for the FAIMS server and log in.

The "Create Module" page offers a number of fields that allow you to describe the module name (required), version, year, description, author, and more. Give your module the name "Simple Sample Module". On the right hand side of the screen are several boxes with file upload buttons (marked "browse"). For each of these boxes, click "browse" and attach the module necessary files you just created. Once again, the files for each box are:

1. Data Schema: `data_schema.xml`
2. UI Schema: `ui_schema.xml`
3. Validation Schema: `validation.xml`

4. UI Logic: `ui_logic.bsh`
5. Arch16n (optional, since it's only useful if you designed your module to support interchangeable terms, but it can be ugly if you leave it out): `arch16n.properties` or `english.0.properties`
6. CSS (optional, but recommended): `ui_styling.css`

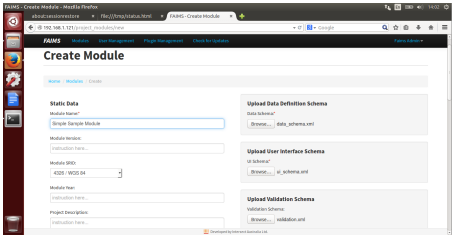

Figure 5.4 Filling in the create module page.

Click the "Submit" button at the bottom of the page to have the FAIMS Server compile your module.

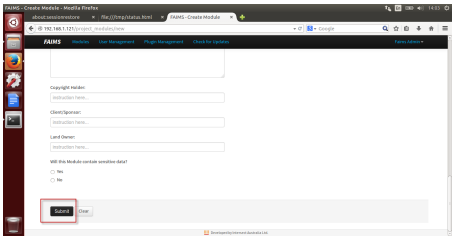

Figure 5.5 The important submit button.

Once the server creates the module, you'll be directed back to the main "Module" tab.

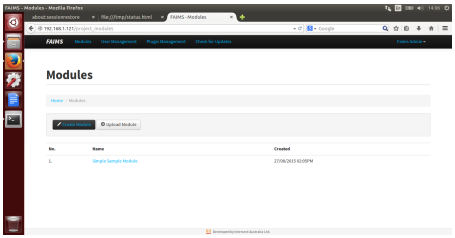

Figure 5.6 Now the module appears.

If you click on the module name, you'll see a screen where you can manage the module, including editing the module metadata, schemas, add user accounts, and downloading or exporting your module. You may also browse any records uploaded from mobile devices using the "Module Details" area and "Search Entity Records" button. Near the bottom of the page, you can delete the module. For more information on deploying or deleting modules from the server, see FAIMS Handout 103 here: FAIMS Handout 103

Now, open up the FAIMS mobile app on your Android device. Click on the three vertical dots in the upper right to open the "Settings" menu.

> **Some Samsung devices may show the menu differently, such as the Samsung S III. A Nexus 7 tablet shows it in the top right corner.**

In the Setting menu, you'll see the option to select a specific FAIMS server to connect to. By default, this is set to the FAIMS Demo server in Sydney, Australia. Click on the dropdown and you'll see that you can set the server address manually (through the "New Server" option) or, if you're connected to the same network as the server, you can use the "Auto Discover Server" to expedite the server setup. We'll assume that your Android device is currently on the same wifi network as your server, so choose "Auto Discover Server" and then hit the "Connect"

button. If auto-detection does not see your server, you may also select "New Server" from the dropdown list and enter the IP address manually. The IP address is visible in the address bar of your internet browser when you are connected to the FAIMS server and will look similar to `192.168.1.133`. Often, the default port of 80 will work.

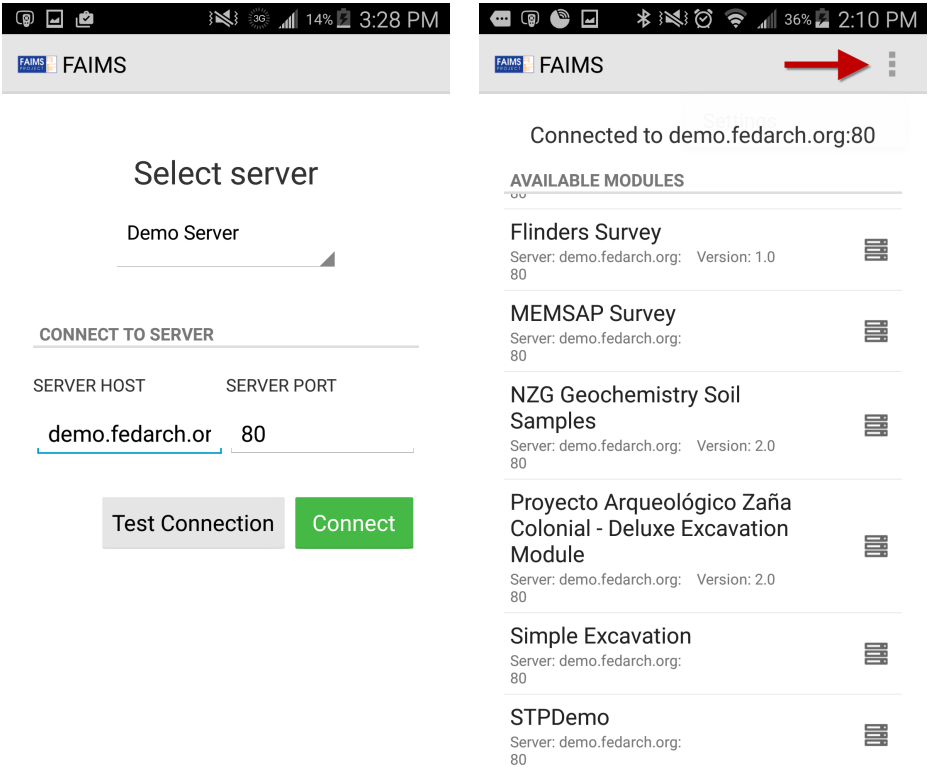

Figure 5.7 Tap the three vertical dots to open "Select Server."

After a few seconds to a minute, the FAIMS app will find your server. Once connected, you will see a list of modules available to use, either locally on your device (designated by a blue phone icon) and those available for download from the server (designated by a black server icon). Once you have downloaded a module, it will show both icons so long as you are still connected to a server that also has the module.

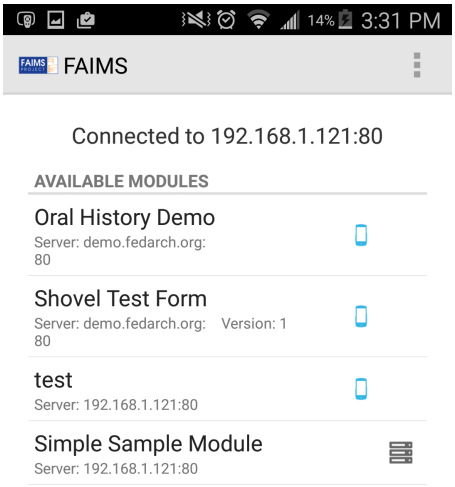

Figure 5.8 Blue phone icons after the module is downloaded.

Tap on the item "Simple Sample Module" from the list, which is the one we just uploaded to the server, and the FAIMS app will copy it to your device. A sidebar with the module metadata will appear. You can browse that quickly, and then tap the "Load Module" button.

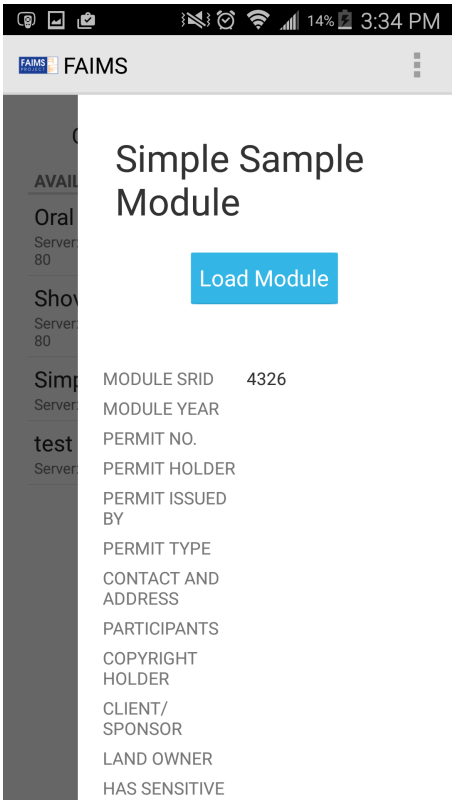

Figure 5.9 Sidebar with module metadata.

Once the module has loaded, you'll be presented with the first tab group of the module, in this case, a user login screen. User logins allow FAIMS to track which users are responsible for which changes. As any archaeologist who has had to decipher and track down which initials belonged to the field worker who sloppily wrote them on a level sheet or artifact bag can tell you, automatically attaching user names to record can make life much easier when going through field collections and forms back at the office or lab. You can add more users via the FAIMS server by selecting the module and clicking the "Edit Users" button.

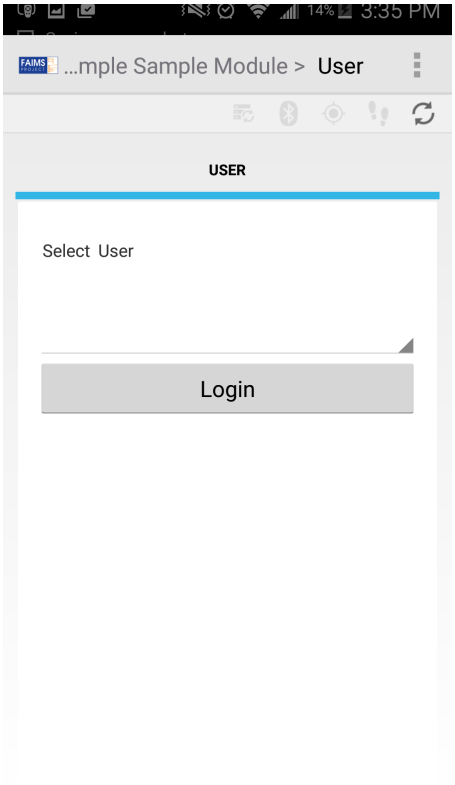

Figure 5.10 User login menu.

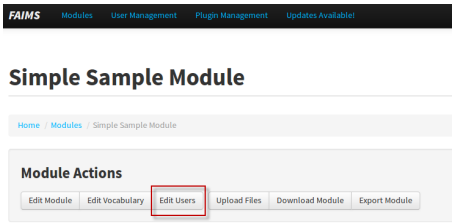

Figure 5.11 Add or remove users throught the "Edit User" tab.

And there you have it. That's essentially what making a module with a `module.xml` file will look like.

Of course, as you've probably realized, the part we glossed over–designing and troubleshooting `module.xml`–was also the tricky part. Fortunately, it's not that tricky after all.

5.3 Understanding The Structure of XML Documents

Codeblock 5.2 is a very unusual description of a man named Richard. It's not code (it won't do anything if you plug it into FAIMS-Tools, because it contains no recognizable instructions), but it's written in a format very similar to what your module's code will look like. Take a minute and look it over and try to guess, just from the text and the way it's formatted, what Richard looks like.

```
<Richard>
    <Old/>
    <Hair>
        <Grey/>
        <Wavy/>
    </Hair>
    <Shirt>
        <Black/>
        <Sleeveless/>
        <Printed>
            <Midnight Oil Diesel and Dust Tour/>
            <Faded/>
        </Printed>
    </Shirt>
    <Watch>
        <Gold/>
    </Watch>
</Richard>
```

Code 5.2 An XML description of Richard.

If you can read this and understand that Richard is old, that he's got wavy grey hair, that he's wearing a black Midnight Oil Dust and Diesel shirt with faded print, and that he has a gold watch, congratulations: you have already grasped the basic principle of how code in XML

documents is structured. If not, try again with that information in mind.

Do you see how the description has a kind of flow to it? How the big category "Richard" is divided into smaller and more specific categories containing individual details? "Grey" and "Wavy" are singular details contained within the category "Hair," which is itself contained within the person "Richard"; therefore Richard has hair that is both grey and wavy. "Faded" and "Midnight Oil Diesel and Dust Tour" are contained within the category "Printed," which is contained along with "Black" and "Sleeveless" within the category "Shirt," which is contained within the person "Richard;" therefore Richard has a shirt, the shirt is black and sleeveless and printed, and the print is faded and says "Midnight Oil Diesel and Dust Tour." Because the details are contained within categories that have a beginning and an end, you know they pertain only to the thing that contains them; we know that Richard's shirt is not wavy, his hair is not black, and he as a person is not Faded. We also know that he's old, because within the person Richard is the detail "Old," but not whether or not his watch is old because the detail "Old" only appears in the person "Richard."

Let's take that basic understanding and look at this section of actual code from the Oral History module's `module.xml` (Codeblock 5.3). You probably won't know what it means or does yet yet, but for now, just focus on the structure as we explain each part of it in detail.

```
<User f="nodata">
    <User>
        <Select_User t="dropdown" f="user" />
        <Login t="button" l="Control" />
    </User>
</User>
```

Code 5.3 Oral History's `module.xml`.

The first thing to understand is the purpose of the angle brackets, <>. By enclosing text that has utility in XML, the angle brackets create

something called a *tag*. The tag is the basic, functional unit of XML documents.

Tags contain information, and the information they contain is referred to as an *element*[9]. In our first example, the tags `<Richard>` and `</Richard>` defined the boundaries of the element Richard, and everything between them was contained within that element.

Usually, elements are defined by two tags: the *opening tag*, and the *closing tag*. Opening tags signify that a new element is beginning. Creating an opening tag instructs that computer that everything that follows until the closing tag (which will be the same as the opening tag, but begin with a `/`) is part of that element. For example, the tag `<User>` states that everything until the closing tag, `</User>`, is part of the element "User" being outlined.

You may notice that there's two `</User>` tags above. You can probably guess from the way Codeblock 5.3 is indented that the very last `</User>` closing tag corresponds to the very first tag, `<User f="nodata">` (we'll explain later why this isn't closed by the tag, `</User f="nodata">`). However, it's worth mentioning that formatting doesn't really matter to FAIMS-Tools when it interprets how your code works; indenting when you start a new element is just a good practice for helping you keep track of your code. So in instance, and others like it: how does FAIMS-Tools decide which `</User>` closing tag belongs to which `<User>` opening tag?

Remember that opening and closing tags are used to show that elements are contained within one another. That's the key word to keep in mind: *contained*. It may help to think of opening and closing tags as functioning like parentheses or quotation marks. If we consider the following sentence:

[9] Remember when we talked about archaeological and relationship elements? Now may be a good time to review.

I went to the store (the one that had the food I like (eggs, milk, bacon) and low prices) in my car.

It's clear that the idea (eggs, milk, bacon) needs to be concluded before the idea of (the one that had the food I like and low prices) can be resumed and itself concluded. Therefore, we instinctively know that the first right parentheses belongs to the most recent left parentheses, not to the first one. Beginning and end tags work the same way. You can't close the first "User" element if the second "User" element, more recently begun, hasn't itself been closed.

You may have noticed that the XML Code Snippet 5.4 contains two tags which don't fall into either the *opening tag* or *closing tag* categories:

```
<Select_User t="dropdown" f="user"/>

<Login t="button" l="Control"/>
```

Code 5.4 Empty element tags are an efficient way to create GUI elements with no additional elements inside them.

These are known as *empty-element tags*. Sometimes, an element is complete with only a single instruction. It's not creating something larger that has multiple qualities; it's just a single detail, such as a simple button the user can press. In these cases, it isn't really necessary to have both an opening and a closing tag, as there's nothing to put between them. The above tags are really just shorthand for the following in Code Snippet 5.5:

```
<Select_User t="dropdown" f="user">
</Select_User>

<Login t="button" l="Control"/>
</Login>
```

Code 5.5

There's no point in having an opening AND closing tag if there's nothing you're going to put between them. Hence the name, "empty element tags"; they denote elements which do not contain any other elements.

Quiz 5.2 Test your knowledge

1. What's the connection between "tags" and "elements"?
2. If you've got two of the same opening tag and neither has been closed yet, which will the first closing tag you write belong to?
3. Where do you put the slash (/) to denote an empty element tag?

5.4 Understanding The *FAIMS* XML Format

So that's how XML is structured. How does that structure relate to FAIMS modules?

FAIMS modules have the same kind of hierarchical structure as other XML documents: all **GUI elements** (ie, the user interface) such as buttons, text fields, and dropdown menus appear inside of **tabs**.

Consider for a moment the module in Figure 5.12 (left image). There are **GUI elements** (or, "useful parts you can interact with") which all belong to the "Recording Form" tab. If the user were to tap on a different tab, for instance the "Interview Details" tab, they would see a different set of GUI elements belonging to that different tab.

The right image in Figure 5.12 shows all the **tabs** which belong to a presently displayed **tab group** called "Form". Entering a different tab group would cause a different set of tabs to be displayed.

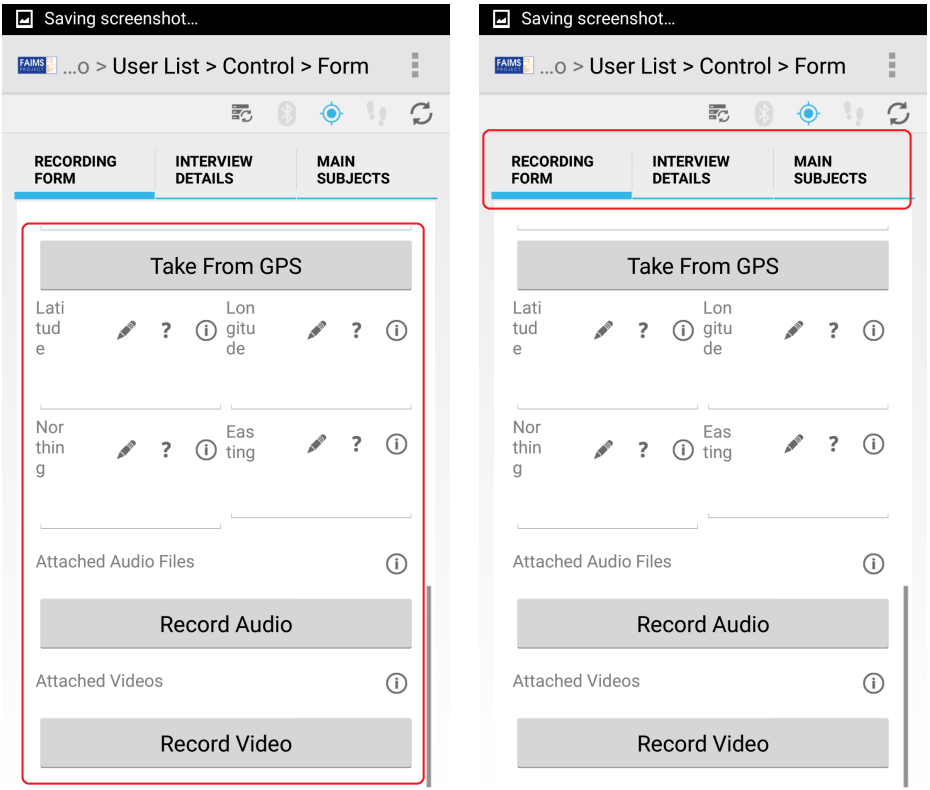

Figure 5.12 GUI elements in a tab and tabs in a tab group.

Figure 5.13 shows two different tab groups—"Control" and "Form"—each containing their own sets of tabs.

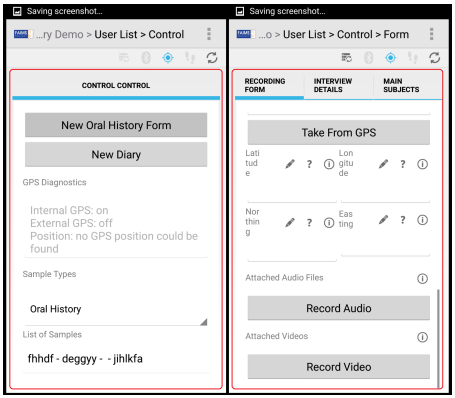

Figure 5.13 "Control" and "Form" have different tabs and GUI elements.

So the hierarchy flows: your module has tab groups have tabs which have GUI elements. When you structure the elements of `module.xml`, it'll be contained exactly the same way. Your module element will contain tab group elements will contain tab elements which will contain GUI elements, many of which may be empty elements as they'll stand on their own without needing to contain anything else. Make sense so far? See what it looks like in practice in Figure 5.6.

```
<?xml version="1.0"?>
<module>
    <Tab_Group>
        <Tab_1>
            <Select_User t="dropdown" f="user"/>
            <Login t="button" l="Control"/>
        </Tab_1>
        <Tab_2>
            <Button t="button" />
        </Tab_2>
    </Tab_Group>
</module>
```

Code 5.6 The hierarchy of modules, tab groups, tabs, and GUI elements.

Remember that you're creating a version of `module.xml` that XML-Tools can read and interpret in creating your necessary files. Because XML-Tools understands the hierarchical structure we've just explained, it'll automatically understand whether something is a tab group or tab just from where it falls in the hierarchy. XML-Tools knows that `<Tab_Group>` is a tab group not because of its name, but because it appears straight away from the `<module>` tags without being contained in anything else.[10] Similarly, elements which appear directly within a tab group are automatically understood to be tabs. XML elements within tabs are interpreted as GUI elements.

```
<module>
    <A>
        <B>
            <C/>
            <C/>
        </B>
        <B>
            <C/>
        </B>
    </A>
    <A>
        <B>
            <C/>
            <C/>
            <C/>
        </B>
    </A>
</module>
```

Code 5.7 Can you infer which elements are tab groups, tabs, and UI elements?

[10] There are some caveats to this rule, but it is true in the vast majority of instances.

Quiz 5.3 Test your knowledge

1. Review the structure in Codeblock 5.7.
2. Without knowing what's really inside the tags, can you figure out which elements are the tab groups, which are the tabs, and which are UI elements? If you have difficulty guessing that the "A" elements are tab groups, the "B" elements are tabs, and the "C" elements are the GUI, you may need to review the previous section.

5.5 Exploring a Simple Module

Now that you understand the how the XML structure relates to FAIMS modules, let's explore the simple module we uploaded in the last section.

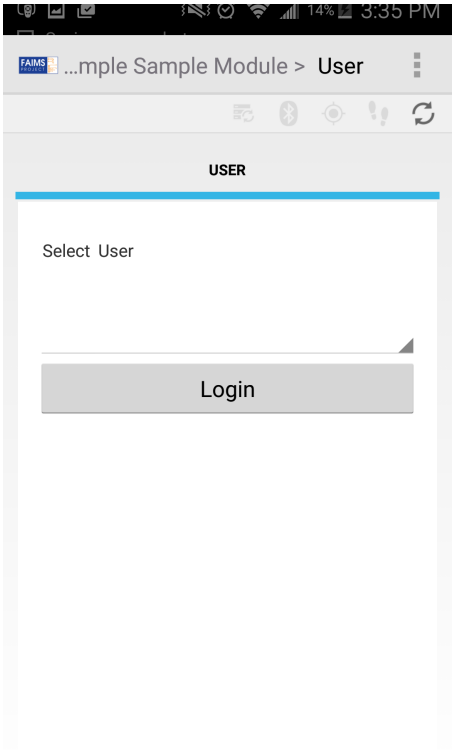

Figure 5.14 A user login screen.

To create this first login screen (Figure 5.14), we used the XML code in Codeblock 5.8.

```
<User f="nodata">
    <User>
        <Select_User t="dropdown" f="user"/>
        <Login t="button" l="Control"/>
    </User>
</User>
```

Code 5.8 XML code for a user login screen.

The outermost set of **<User>** tags comes right after the **<module>** tags, so they create a **tab group**. You can name these whatever you want. The only rule you should follow, for reasons we'll get into very shortly, is that it should start with an uppercase letter. The *tag name* you write is displayed in the breadcrumb navigation bar at the top of the screen.

Notice that the topmost **<User>** tag contains the *attribute* `f` with an *attribute value* of `nodata`. Including the word `nodata` in the `f` attribute's value prevents the FAIMS-Tools from automatically generating unwanted code associated with the data schema. In practical terms: inputs that are provided in the "User" tab group, including the "Select User" dropdown menu, are not considered "data" that must be saved to the FAIMS database. We chose to put `nodata` here because we only want the "User" tab group to be for letting people log in.

Now that we've got our **<User>** tab group, it's time to make our actual **<User>** tab with its relevant GUI elements. We're naming this tab "User" as well, because it's where the User logs in and that seems like a good name, but we could have named it something else if we thought we'd get it confused with the tab group. Whatever tag name we choose will be displayed in the list of tabs. When you name a tab group something, its tag name must be capitalized.

The **<Select_User>** and **<Login>** elements represent GUI elements, or the actual things users will interact with when they're viewing a tab. Notice the `t` attribute, which is where we can define part of what this element looks like or does. You'll find a list of these in the FAIMS cookbook. For **<Select_User>** we're using `t="dropdown"`, which means this GUI element is a drop-down menu.

So how do we determine what's in that drop down menu? In this case, by creating an attribute, `f="users"`, which in this context FAIMS

understands to mean "get the list of usernames from the server and put them here."[11]

Setting `t="button"`, as in the case of `<Login>`, creates a button you can tap. The purpose of the button is described by the text that comes after it; otherwise, it's just a button, which probably isn't going to be very useful for your module. Here we've included the code `l="Control"` within the tag, which causes FAIMS to link to the "Control" tab group when it is tapped. In fact, the `l` attribute works not only for buttons, but many other GUI elements as well. Note carefully that the "Control" tab group linked to by the `l` attribute's value is defined further down in the `module.xml` file; this code only functions because the destination is valid. If we had `1="Control"` and then didn't actually have a "Control" to link to, it's safe to say this wouldn't work. Also note that references are case-sensitive, so writing `control` with a lower-case "c" would fail.

So let's say we're using our module, we've just clicked the dropdown menu and chosen our user, then we've hit the button that says "Login." If you're using the finished sample module, you'll find yourself looking at the screen in Figure 5.15.

[11] **Note from FAIMS programmer Christian Nassif-Haynes:** If you use `t=dropdown`, then it is possible for the user to avoid logging in (in error or intentionally) by selecting the null option and clicking the login button. If you want to prevent logging with the null user option (de facto avoiding the login), then you need to manually modify the `ui_logic.bsh` file after it's been generated or use `t=list`, following the guidelines below. Lists, unlike dropdowns, do not allow null elements so using a `t="list"` for login, as in the code below, prevents the problem of logging in with null user.
<User f="nodata"> <User f="noscroll"> <Select_User t="list" f="user" l="Control"/> </User> </User>

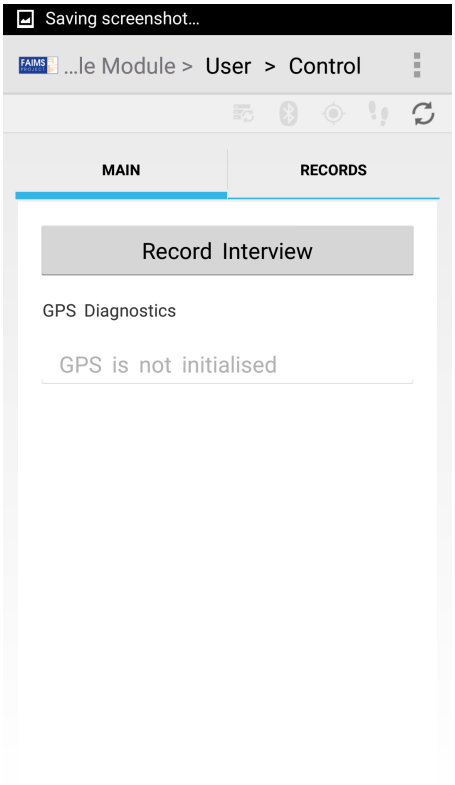

Figure 5.15 The "main" tab.

Let's take a look at the code for this tab group here. As you review Codeblock 5.9, see if you can figure out which elements are tab groups, tabs, and GUI elements.

```
<Control f="nodata">
    <Main>
        <Record_Interview t="button" l="Interview"/>
        <GPS_Diagnostics t="gpsdiag"/>
    </Main>
    <search>
        Records
    </search>
</Control>
```

Code 5.9 Matching XML to FAIMS elements.

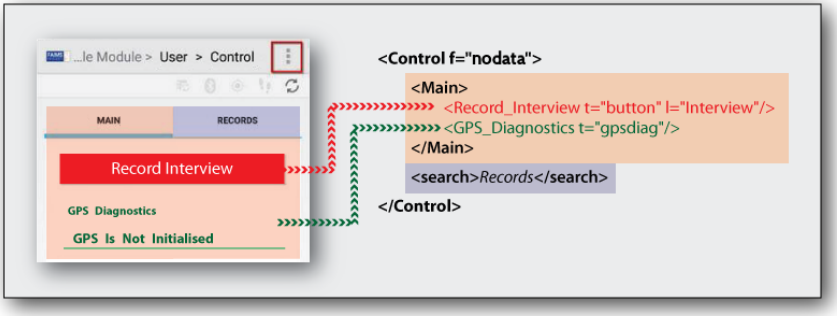

Figure 5.16 How XML in `module.xml` describes different FAIMS Elements.

The XML in `module.xml` describes the different FAIMS elements and how they should appear in the module.

This "Control" tab group encompasses two other elements, "Main" and "Search." Let's look at each individually.

The `<Main>` element creates the first tab. Inside that tab we have a GUI element, `Record_Interview`. The `t="button"`, which means this element is a button; `1="Interview"`, so the button links to another tab somewhere else in the module called "Interview." The other GUI element here is "GPS Diagnostics," which has the element type `gpsdiag`. This element type specifically means that in the final module, it will create text labels and display information about the Android device's GPS

location. In the screenshot, we see the `gpsdiag` element at work: it's telling us that our phone's GPS is "not initialized," a charming way of sayingn "not turned on." Until we do turn it on, this is the best `gpsdiag` is going to do.

Let's turn on our Android device's internal GPS antenna by tapping *the three vertical dots* in the upper right of the screen to open the settings menu.

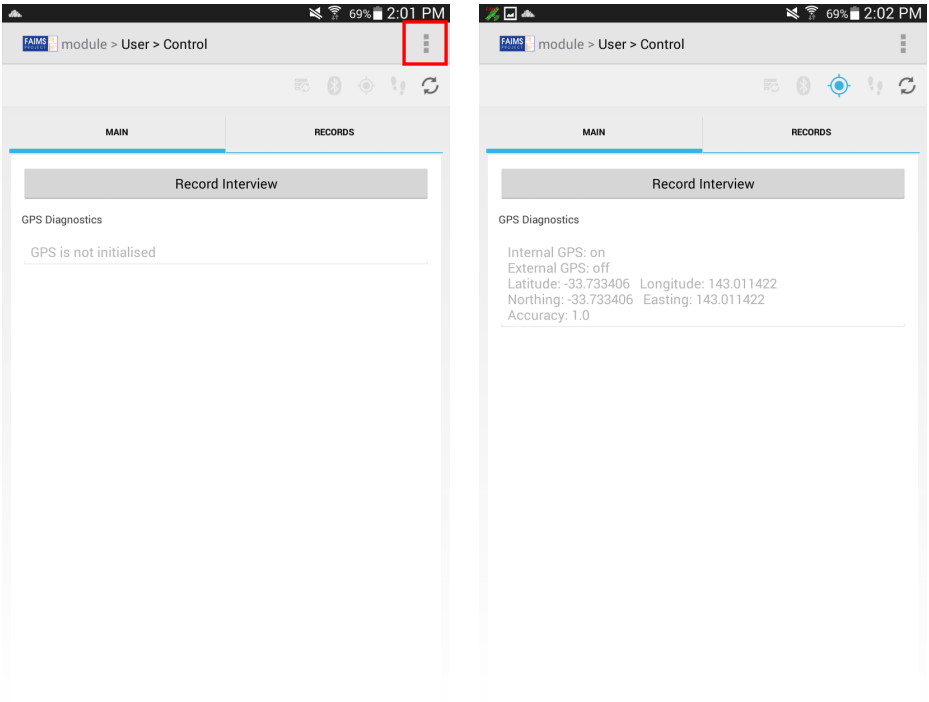

Figure 5.17 Turn on internal GPS by tapping the dots in the red box.

With the GPS antenna turned on, the `gpsdiag` element now displays quite a bit more information, including the status of the GPS antenna, location in both Latitude/Longitude and Easting/Northing, and an accuracy measurement.

So much for the **<Main>** tab. Let's look at the next tab, which you may have already noted is a little peculiar. For one thing, the tag name begins with a lowercase letter. Didn't we tell you never to do that? Furthermore, the text contained within, "Records," isn't in tags. What's going on here?

Don't worry; you haven't missed anything. It's just that there are certain kinds of tabs that FAIMS-Tools is already specially programmed to recognize. These kinds of tabs are complicated and a lot of work to make, so rather than ask you to create them, we've created a shorthand that FAIMS-Tools recognizes and runs with. These "shorthand" tabs have lowercase names, which is why you generally shouldn't come up with a lowercase tag name; you might accidentally write the name of a shorthand FAIMS-Tools recognizes, which will create a mess as it tries to follow its prewritten instructions at the same time as the instructions you've created.

In this case, the tab is "search," which, once some data has been collected, will contain GUI elements that allow you to search records according to various criteria, including term or entity type. We don't have to include code for all these GUI elements; FAIMS-Tools knows to include them in a tab labelled with the shorthand "search". The only thing we have included is the plaintext "Records", without a tag. When FAIMS-Tools sees text like this written inside an element, it labels the element that way instead of basing its name on the tab group. This is why you see "Records" as the tab label when you might have expected to see "search".

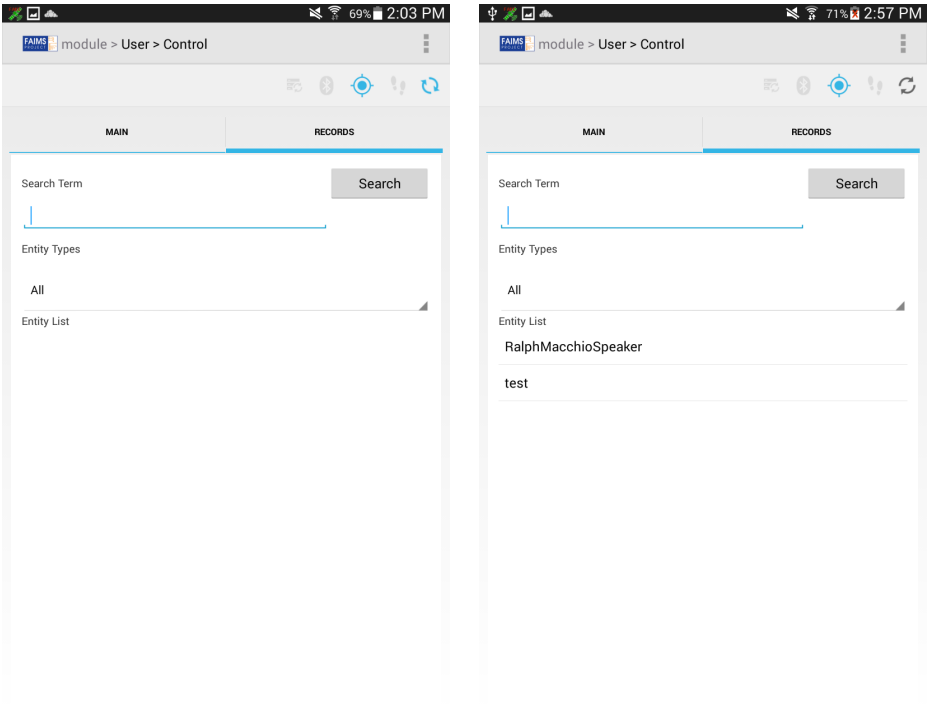

Figure 5.18 The Search element before and after adding a few records.

Going back to the finished module, let's tap the "Main" tab and then click the "Record Interview" button, which, you may recall, will take us to "Interview." "Interview" looks like Figure 5.19:

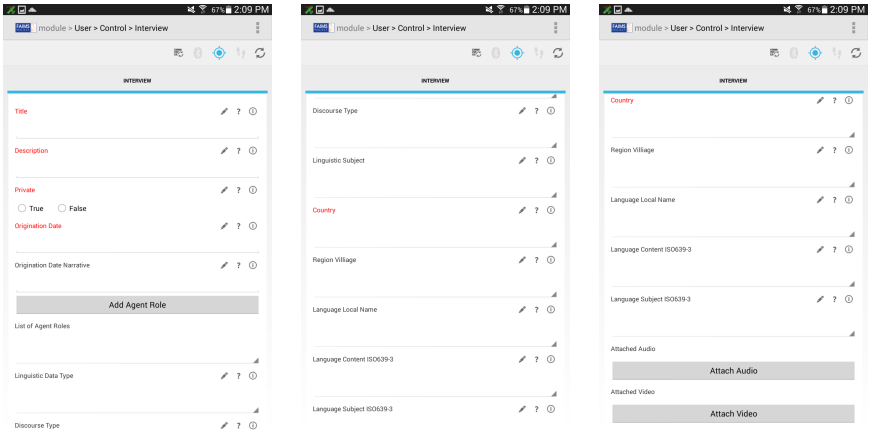

Figure 5.19 Scroll to see all fields in the "Interview" tab.

The code to create this long screen is below. This is a lot more code than you've seen before in one go, but it serves as a very useful example of a lot of different techniques, so resist the urge to skim it. Take your time, and when you don't recognize what a tag is or what an element is for, see if you can guess what it does just from context.

```
<Interview>
    <Interview>
        <Title f="id notnull">
            <desc>This title should be a sensible title, unique to each
item, briefly summarising the contents of the item, for example
"Ilocano songs recorded in Burgos, Ilocos Sur, Philippines, 17 April
1993"</desc>
        </Title>
        <Description f="notnull">
            <desc>Description may include but is not limited to: an
abstract, table of contents, reference to a graphical representation of
content, or a free-text summary account of the content. {[}DCMT{]}
Description may also offer an annotation, or a qualitative or
evaluative comment about the resource, such as a statement about
suitability for a particular application or context.</desc>
        </Description>
        <Private t="radio" f="notnull">
```

```
            <desc>Choose either "false", meaning that the metadata for
the item should be publicly available, or "true", meaning that the
metadata for the item should be hidden (perhaps because you plan to
check it and edit it later).</desc>
            <opts>
                <opt>True</opt>
                <opt>False</opt>
            </opts>
        </Private>
        <Origination_Date f="notnull">
            <desc>Date the item was captured or created, using the
format yyyy-mm-dd. If you are unsure of the day, month or decade enter
the first day of the relevant period: e.g. "1970s" 1970-01-01, "2001"
2001-01-01, "February 1993" 1993-02-01. If entering a date of this
type, clarify in the originationDateNarrative field. If you really did
record on 1 January 2001, say so in the originationDate field.</desc>
        </Origination_Date>
        <Origination_Date_Narrative>
            <desc>Use this field to provide any necessary comments on
the scope of the value you entered in the origination date field, e.g.
"unknown date in February 1993"</desc>
        </Origination_Date_Narrative>
        <Add_Agent_Role t="button" lc="Agent_Role"/>
        <List_of_Agent_Roles t="dropdown" ec="Agent_Role"/>
        <Linguistic_Data_Type>
            <desc>If data are relevant to linguistics, choose one of
the three basic linguistics data types. Primary text: Linguistic
material which is itself the object of study; Lexicon: a systematic
listing of lexical items; Language description: describes a language or
some aspect(s) of a language via a systematic documentation of
linguistic structures. If your data are not relevant to linguistics,
leave this field blank.</desc>
            <opts>
                <opt>Lexicon</opt>.
                <opt>Language Description</opt>
                <opt>Primary Text</opt>
            </opts>
        </Linguistic_Data_Type>
        <Discourse_Type>
```

```
            <desc>Used to describe the content of a resource as
representing discourse of a particular structural type. Dialogue:
interactive discourse with two or more participants; drama: planned,
creative, rendition of discourse involving two or more participants;
formulaic: ritually or conventionally structured discourse; ludic:
language whose primary function is to be part of play, or a style of
speech that involves a creative manipulation of the structures of the
language; oratory: public speaking, or of speaking eloquently according
to rules or conventions; narrative: monologic discourse which
represents temporally organized events; procedural: explanation or
description of a method, process, or situation having ordered steps;
report: a factual account of some event or circumstance; singing: words
or sounds {[}articulated{]} in succession with musical inflections or
modulations of the voice; unintelligible: utterances that are not
intended to be interpretable as ordinary language.</desc>
            <opts>
                <opt>Dialogue</opt>
                <opt>Drama</opt>
                <opt>Narrative</opt>
                <opt>Procedural</opt>
                <opt>Ludic</opt>
                <opt>Singing</opt>
                <opt>Oratory</opt>
                <opt>Report</opt>
                <opt>Unintelligible speech</opt>
                <opt>Formulaic</opt>
            </opts>
        </Discourse_Type>
        <Linguistic_Subject>
            <desc>Use to describe the content of a resource if it is
about a particular subfield of linguistic science.</desc>
            <opts>
                <opt>Phonology</opt>
                <opt>Text And Corpus Linguistics</opt>
                <opt>Historical Linguistics</opt>
                <opt>Language Documentation</opt>
                <opt>Lexicography</opt>
                <opt>Typology</opt>
            </opts>
```

```
        </Linguistic_Subject>
        <Country f="notnull">
            <desc>This should be the standard name of the country in
which the file was recorded (see
http://www.ethnologue.com/country_index.asp). Prefix the country name
with the two-letter ISO3166-1 code
(http://www.iso.org/iso/country_codes.htm).</desc>
            <opts>
                <opt>PH - Philippines</opt>
                <opt>AU - Australia</opt>
            </opts>
        </Country>
        <Region_Villiage>
            <desc>Indicate the geographical scope of the item. Enter
data in the order locality, state or province, country.</desc>
            <opts>
                <opt>{[}locality{]}, {[}state or province{]},
{[}country{]}</opt>
                <opt>Burgos, Ilocos Sur, Philippines</opt>
            </opts>
        </Region_Villiage>
        <Language_Local_Name>
            <desc>The purpose of this field is to reflect language
names in local use, with local spellings, if different from official
name.</desc>
            <opts>
                <opt>Language - local spelling {[}free text{]}</opt>
                <opt>Ilocano</opt>
            </opts>
        </Language_Local_Name>
        <Language_Content_ISO639-3>
            <desc>Content language is the language included in your
data (spoken and/or written). Insert the 3-letter ISO 639-3 code for
your language, and the standard name of the language as spelt in the
ethnologue entry {[}search on www.ethnologue.com/site_search.asp{]}.
Separate the code and the language with a hyphen, e.g. "ilo -
Ilocano"</desc>
            <opts>
                <opt>mis - Uncoded languages</opt>
```

```
                <opt>und - Undetermined languages</opt>
                <opt>mul - Multiple languages</opt>
                <opt>zxx - No linguistic content</opt>
                <opt>{[}3-letter ISO639-3 code{]} - {[}Ethnologue name
of language{]}</opt>
                <opt>ilo - Ilocano</opt>
                <opt>eng - English</opt>
            </opts>
        </Language_Content_ISO639-3>
        <Language_Subject_ISO639-3>
            <desc>Subject language is the language that is the subject
of your research. Insert the 3-letter ISO 639-3 code for your language,
and the standard name of the language as spelt in the ethnologue entry
{[}search on www.ethnologue.com/site_search.asp{]}. Separate the code
and the language with a hyphen, e.g. "ilo - Ilocano"</desc>
            <opts>
                <opt>zxx - No linguistic content</opt>
                <opt>{[}3-letter ISO639-3 code{]} - {[}Language subject
of your research{]}</opt>
                <opt>mis - Uncoded languages</opt>
                <opt>und - Undetermined languages</opt>
                <opt>mul - Multiple languages</opt>
                <opt>ilo - Ilocano</opt>
            </opts>
        </Language_Subject_ISO639-3>
        <Attached_Audio t="audio"/>
        <Attached_Video t="video"/>
    </Interview>
</Interview>
```

First, the easy stuff. You should already be able to guess what the first element, **<Interview>**, is: a tab group. Since there's another opening tag also called **<Interview>**, you've probably also guessed that the tab group **<Interview>** has a tab labelled **<Interview>**. So let's skip straight to the GUI elements located within this tab.

The first element is **<Title>** (Codeblock 5.10).

```
<Title f="id notnull">
    <desc>This title should be a sensible title, unique to each item,
briefly summarising the contents of the item, for example "Ilocano
songs recorded in Burgos, Ilocos Sur, Philippines, 17 April
1993"</desc>
</Title>
```

Code 5.10 Breaking down **<Title>**.

For the **<Title>** element, there is no UI "type" (represented by `t=`) specified. So it becomes the default "type;" a text input field. (See *Type Guessing for GUI Elements in FAIMS-Tools* for an explanation of how this was determined.) The flag `f=notnull` designates that this is a required field; the record cannot be saved if it is empty. If you can imagine your stress levels skyrocketing because of teammembers not remembering to enter in their (X), set the relevant element to `f=notnull` and they will have no choice but to remember.

<desc> allows you to set a description that your users can access by tapping and holding for a few seconds on the info button as in Figure 5.20.

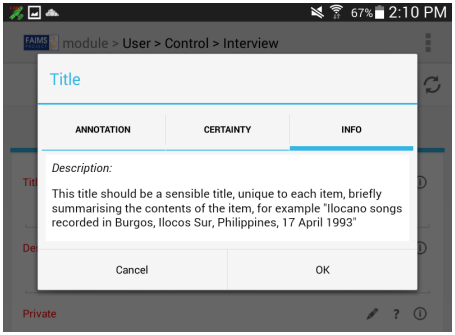

Figure 5.20 A description pop-up appears.

That wraps up the **<Title>** element. Now we have an opening tag for another GUI element, **<Private>**. Private is a radio button UI object, required, has a description, and includes two options ("True"

and “False”). Review the code in Codeblock 5.11 and see if you can understand how all of this is accomplished.

```
<Private t="radio" f="notnull">
    <desc>Choose either "false", meaning that the metadata for the item
should be publicly available, or "true", meaning that the metadata for
the item should be hidden (perhaps because you plan to check it and
edit it later).</desc>
    <opts>
        <opt>True</opt>
        <opt>False</opt>
    </opts>
</Private>
```

Code 5.11 XML for **<Private>**.

This next element (Codeblock 5.12), **<Add_Agent_Role>**, designates a button element (’t="button"’) that links to the tab group `Agent_Role`. This will allow users to register new Agent Roles.

```
<Add_Agent_Role t="button" lc="Agent_Role"/>
```

Code 5.12 XML for **<Add_Agent_Role>**.

Note that this time, instead of using an `l` to redirect elsewhere, `lc` was used. The difference is that using `lc` instead of `l` establishes a parent-child relationship, with the entity linking becoming a parent and the entity being linked to becoming a child. This is useful for organizing your module’s data in a neat, hierarchical way.

Codeblock 5.13 designates a dropdown menu, **<List_of_Agent_Roles>**, which is populated with a list of `Agent_Role` records. Specifically, they will be the `Agent_Role` records which were saved using the button element above. The FAIMS-Tools knows these are the right records to display because the button and dropdown menu appear in the same tab group.

```
<List_of_Agent_Roles t="dropdown" ec="Agent_Role"/>
```

Code 5.13 XML for `List_of_Agent_Roles`.

The final two element types used in this tab are the `t="audio"` and `t="video"` types (Codeblock 5.14). These two element types allow you to record audio and video files and attach them to your records.

```
<Attached_Audio t="audio"/>

<Attached_Video t="video"/>
```

Code 5.14 XML for attaching audio and video.

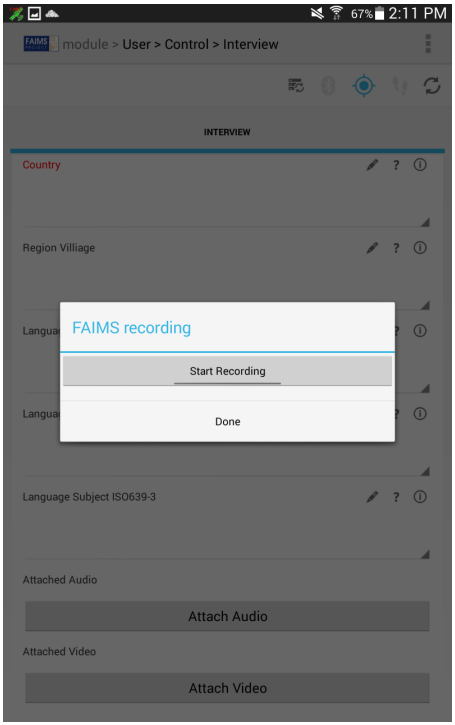

Figure 5.21 With the audio UI element, you'll get a popup window that allows you to start and stop your audio recording.

The final elements are on the Agent Role tab group, and include a few element types we've already seen: two text input fields: **<First_Name>** and **<Last_Name>** both with flags that designate them as `ids` and a dropdown menu, **<Role>** which contains a few options and is also flagged as an `id` (Codeblock 5.15).

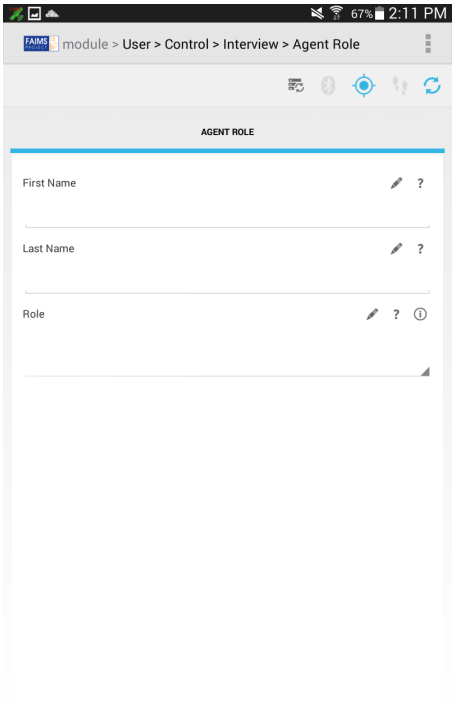

Figure 5.22 "Agent Role" tab group appears when adding agent roles.

Now that we've explained each part, go back and review the whole one more time. You can go a long way creating modules for your team only by using the techniques we've explicitly covered above. It's very possible that you've already learned everything you need to know to make an effective module for your team.

```
<Agent_Role>
    <desc>Enter participant name in the format Lastname, Firstname.
Choose the participant role from the closed vocabulary provided. Use
the description field to provide additional information on role or
agents. Enter participant name in the format Lastname, Firstname.
Choose the participant role from the closed vocabulary provided. Add
more participants by clicking the "+" button to the right. If you need
to provide extra information on the agent or the role, use the item's
"Description" field to provide additional information on role or
agents.</desc>
    <Agent_Role>
        <First_Name f="id"/>
        <Last_Name f="id"/>
        <Role f="id">
            <opts>
                <opt>Data Inputter</opt>
                <opt>Performer</opt>
                <opt>Speaker</opt>
                <opt>Developer</opt>
                <opt>Transcriber</opt>
                <opt>Photographer</opt>
                <opt>Interpreter</opt>
                <opt>Singer</opt>
                <opt>Signer</opt>
                <opt>Compiler</opt>
                <opt>Recorder</opt>
                <opt>Depositor</opt>
                <opt>Interviewer</opt>
                <opt>Editor</opt>
                <opt>Author</opt>
                <opt>Translator</opt>
                <opt>Researcher</opt>
                <opt>Annotator</opt>
                <opt>Participant</opt>
            </opts>
        </Role>
    </Agent_Role>
</Agent_Role>
```

Code 5.15 XML for “Agent Role” tab group.

As an exercise, follow these instructions and produce your own copy of the module's code in `module.xml`. Then save that to the server and run the generate.sh script to produce necessary files. Go to "create module" and upload the necessary files to the server as the "Simple Sample Module."

Congratulations; you've just a simple module.

5.6 Iterating to Match the Oral History Module

Now that we've created a very straightforward version of the Oral History module, let's layer in some additional features. By the end of this section, we will have discussed all the steps that went into making the version of the Oral History Module you can download from the FAIMS Demo Server.

Select "Demo Server" from the dropdown menu in the FAIMS setting menu, then download the "finished" version of the Oral History module. Our goal in this section will be to update the version you produced to match this more complex iteration.

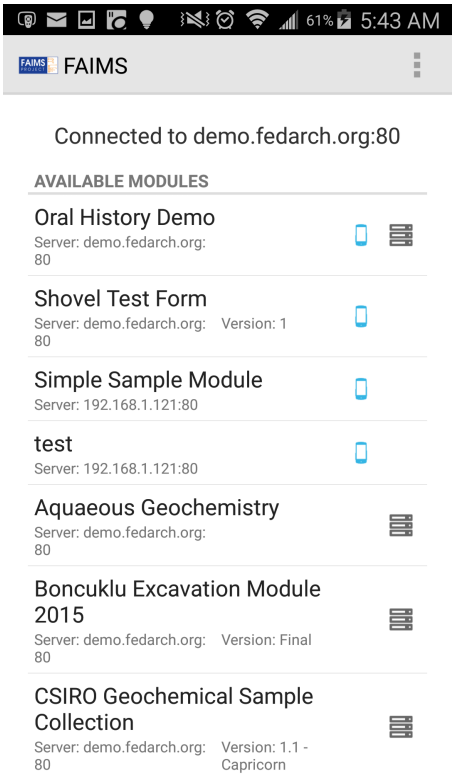

Figure 5.23 Tap "Oral History Demo" to download.

The Oral History module has a bit more metadata information included than the version we've already discussed. You can update this information in the "Module" tab on your FAIMS server installation.

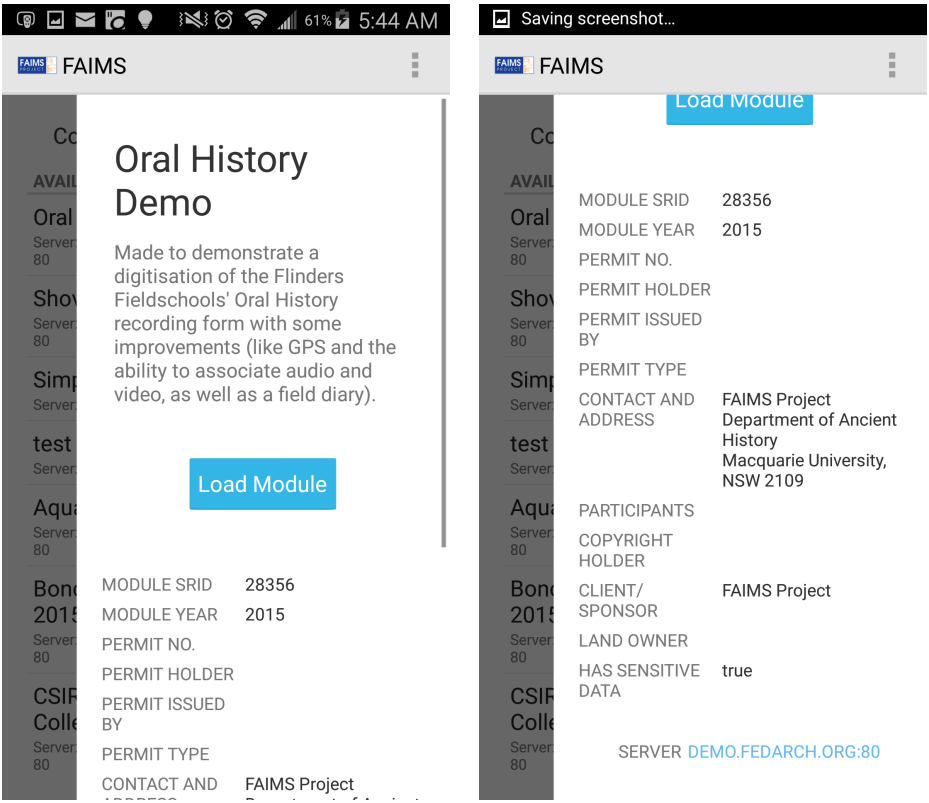

Figure 5.24 More detailed metadata.

The first screen of the Oral History Demo (Figure 5.25) looks similar to our module.

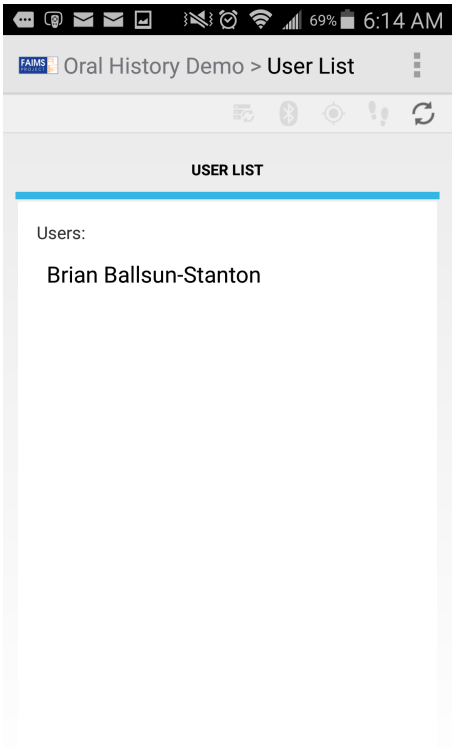

Figure 5.25 Match your model to "Oral History Demo" by adding a User List.

In the same `module.xml` file we used for our version of Oral History update the code in Codeblock 5.16 to match Codeblock 5.17 (new code in **bold**).

```
<User f="nodata">
    <User>
        <Select_User t="dropdown" f="user"/>
        <Login t="button" l="Control"/>
    </User>
</User>
```

Code 5.16 Current code.

```
<User f="nodata">

    <User_List>
        <Users t="list" f="user" l="Control">
            Users:
        </Users>
    </User_List>

</User>
```

Code 5.17 This code replicates “Oral History Demo”.

This replicates the “User List” tab illustrated in the above screenshot.

Save your modified `module.xml`, run `./generate.sh` in your Ubuntu terminal again, edit your current module and upload the new necessary files in place of the old ones. Now download this updated module to your FAIMS app.

Importantly, to download the updated version of the module, you must touch and hold the “Simple Sample Module” in the list of modules until the dialogue in Figure 5.26 is displayed.

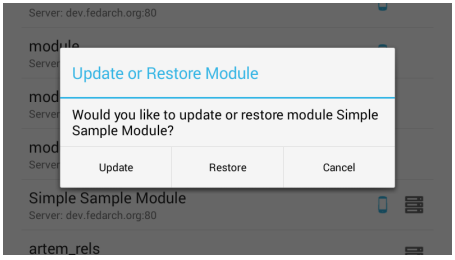

Figure 5.26 Tap and hold “Simple Sample Module” to open this pop-up.

Tap “Update”, then “Update Settings”, wait for the update to occur, and finally load the module as usual.

Note: While we can update Simple Sample Module to have an interface that mimics the Oral History module, we're going to make some changes that will make the module not completely functional. This is because, as you may recall, you can't change the Data Schema of a module once it's already been uploaded to the server. So the interface elements will look right, because the necessary files that govern those can be changed out freely, but the parts actually responsible for managing the data you collect to the server won't have gotten the memo that the module's been altered. If this weren't an exercise, and we wanted an absolutely functional module, we'd just create a new one from the updated necessary files and tell our team members to switch to it.

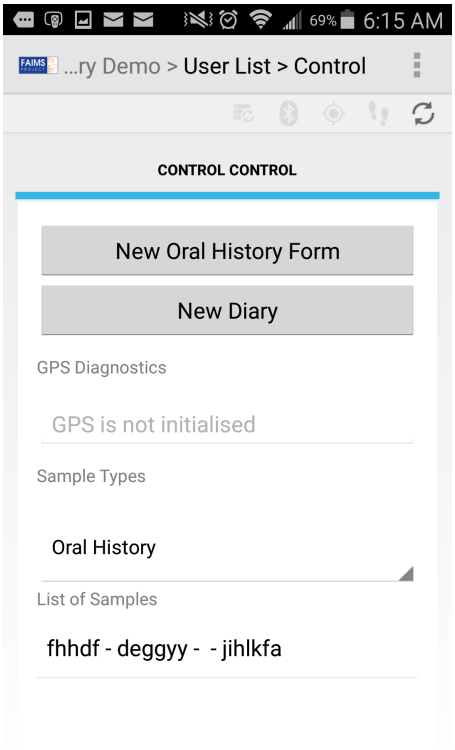

Figure 5.27 "Oral History Demo" combines "Main" and "Record" into "Control".

On the screen in Figure 5.27, we need to condense a few elements from our original module. Change the code in Codeblock 5.18 to match the code in Codeblock 5.19.

Codeblock 5.19 replaces the "Main" and "Record" tab groups with a single one labelled "Control". (In a moment, we will define the "Form" and "Diary" tab groups linked to by the above buttons. We will also have to populate the "Sample Types" dropdown and "List of Samples" list by writing some additional code.)

```
<Control f="nodata">
    <Main>
        <Record_Interview t="button" l="Interview"/>
        <GPS_Diagnostics t="gpsdiag"/>
    </Main>
    <search>
        Records
    </search>
</Control>
```

Code 5.18 Current code.

```
<Control f="nodata">

    <Control>
        <New_Oral_History_Form t="button" l="Form"/>
        <New_Diary t="button" l="Diary"/>
        <GPS_Diagnostics t="gpsdiag"/>
        <Sample_Types t="dropdown" />
        <List_of_Samples t="list" />
    </Control>

</Control>
```

Code 5.19 Condense “Main” and “Record” into “Control”.

Save `module.xml` and run the generate.sh script again. Now use them to update your module again. You should see something similar to Figure 5.28 after selecting a user in the uploaded module.

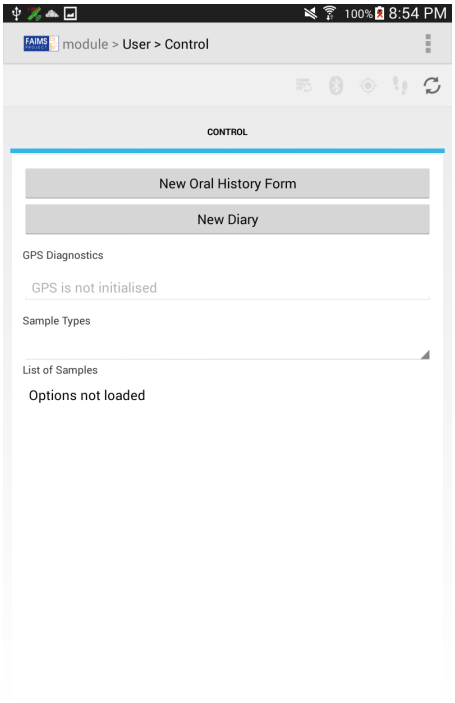

Figure 5.28 "Control" tab in your updated module.

For the next step, we'll add new UI tab groups for each of our buttons: "Form" and "Diary" that we linked to in Codeblock 5.20.

Make sure that the `<Form>` and `<Diary>` elements are written directly within `<module>` so that the FAIMS-Tools correctly interprets them as tab groups. It may help, when drafting new elements, to write both the opening and closing tags and then fill in the middle.

Now, under each of these new tab groups, we'll create individual tabs. The "Form" group has three tabs: "Recording Form", "Interview Details", and "Main Subjects". Note that if these tab titles contain multiple words, you must use underscores between each word. When

```
<Form>
    <Recording_Form>
    </Recording_Form>
    <Interview_Details>
    </Interview_Details>
    <Main_Subjects>
    </Main_Subjects>
</Form>
<Diary>
    <Diary>
    </Diary>
</Diary>
```

Code 5.20 Add “Form” and “Diary” tab groups.

FAIMS creates the title for each tab, underscores will be replaced with spaces.

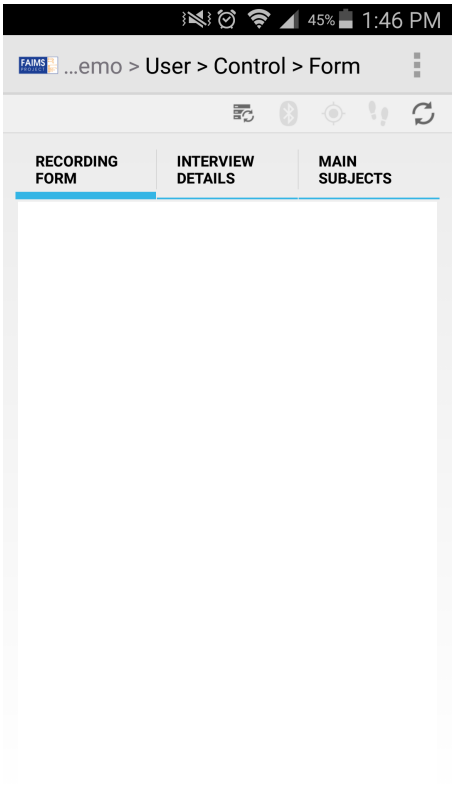

Figure 5.29 Three newly created empty tabs.

Now we have the individual tabs for each section, but those tabs don't have any content just yet. For simple text fields, like "Birth Place" and "Parents' Birth Place" you can simply add a self-closing tag with the field's title as in Codeblock 5.21.

```
<BIRTH_PLACE/>
```

Code 5.21 A self-closing "Birth Place" tag.

Also, non-alphabetical characters, like apostrophes are not allowed as tag names in XML, so the FAIMS-Tools would fail to generate a module which contains the code in Codeblock 5.22.

```
<PARENTS'_BIRTH_PLACE:/>
```

Code 5.22 The apostrophe causes this to fail.

If such characters must be included, the solution is to firstly give the element a sensible name without an apostrophe or colon as in Codeblock 5.23.

```
<PARENTS_BIRTH_PLACE/>
```

Code 5.23 First name the element without an apostrophe.

Then, to make FAIMS-Tools display the apostrophe and colon in the GUI, write them as the element's text as in Codeblock 5.24.

```
<PARENTS_BIRTH_PLACE>
    PARENTS' BIRTH PLACE:
</PARENTS_BIRTH_PLACE>
```

Code 5.24 Write any special characters in the element's text.

This is similar to what we did with the "search" feature in the last section.

Now, note that every tab group which you intend to save requires at least one *identifier*. In the original Oral History module, the identifiers were `PERSON` and `LANGUAGE_GROUP`. We can use the `f` attribute to denote that in our new module as in Codeblock 5.25.

```
<PERSON f="id"/>
<LANGUAGE_GROUP f="id"/>
```

Code 5.25 Denoting an *identifier*.

For the GPS fields and their corresponding "Take From GPS" button, refer to Codeblock 5.26. **`<gps>`** is a special, self-contained shortcut tag that FAIMS will replace with several fields for the latitude, longitude, Easting, and Northing, as well as a button for inserting this data from GPS.

```
<gps/>
```

Code 5.26 XML for GPS fields and button.

To add fields for attached files, you can use the `Audio` and `Video` tags, as in Codeblock 5.27. Simply set the type `t` to video or audio as required.

```
<Attached_Audio_Files t="audio"/>
<Attached_Videos t="video"/>
```

Code 5.27 XML for attaching audio and visual files.

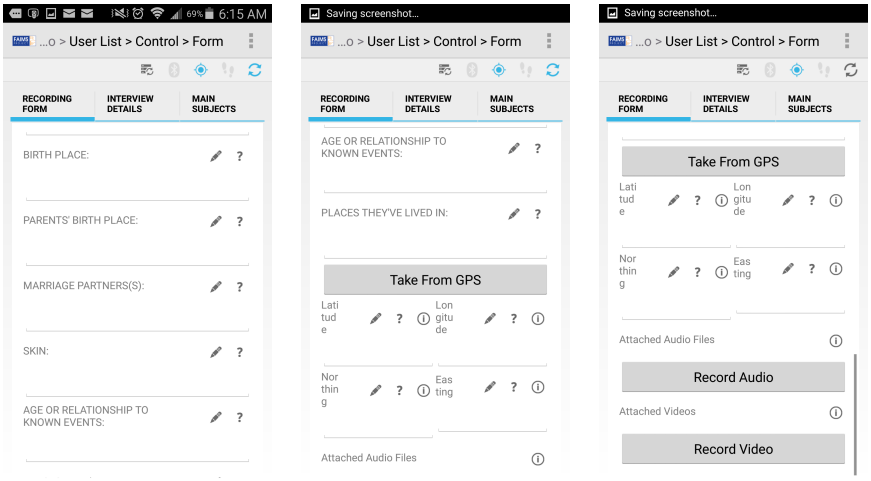

Figure 5.30 Examples of simple text fields, GPS input, and attaching files.

To add the radio buttons, use the code in Codeblock 5.28.

```
<Recorded t="radio">
    Recorded?
    <opts>
        <opt>Recorded</opt>
        <opt>Notes Only</opt>
        <opt>No</opt>
    </opts>
</Recorded>
```

Code 5.28 XML for radio buttons.

Note that to have the radio buttons' contents appear would require that the entire module is re-created and uploaded to the FAIMS server. Updating the existing module on the server, would cause the menu to appear but lack its options.

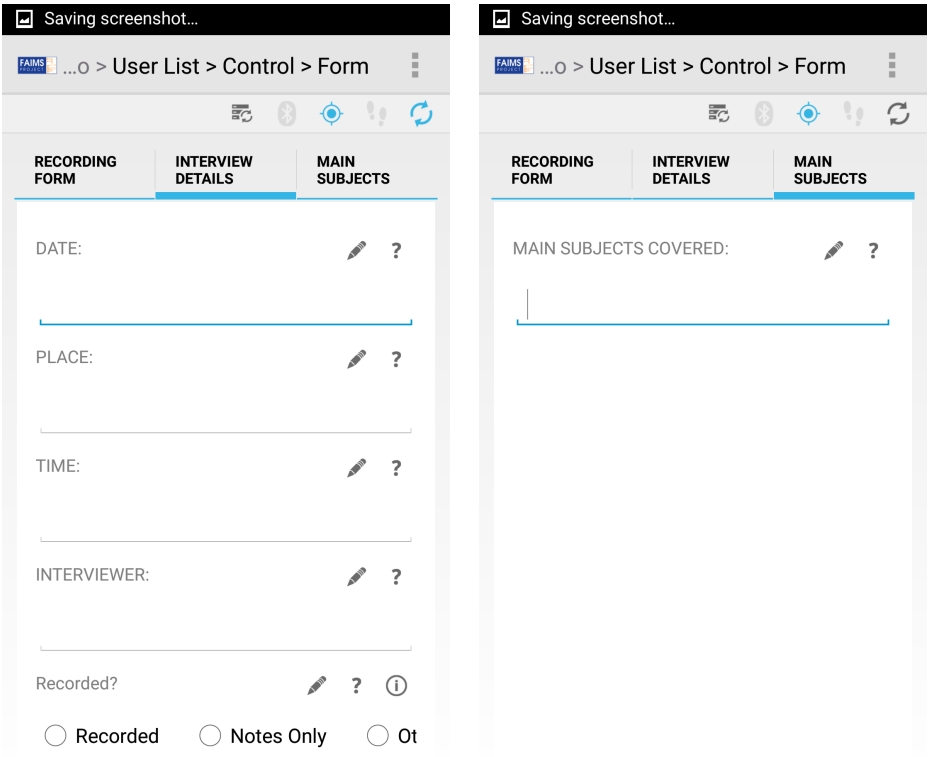

Figure 5.31 Radio buttons in "Interview Details" and a simple text field in "Main Subjects".

The "Timestamp" and "Created By" values are not actually fields that we'll allow users to enter in manually, so we won't put in a data entry field. Instead, we'll make FAIMS set and update these fields when the record is created. Since there's no simple shortcut for that, for now we'll put in the code in Codeblock 5.29.

```
<Timestamp/>
<Created_by/>
```

Code 5.29 XML for “Timestamp” and “Created by”.

There’s actually not a lot more we can do until we’ve generated our necessary files. Then, we’ll take `ui_logic.bsh` and make a few alterations that make use of these tags.

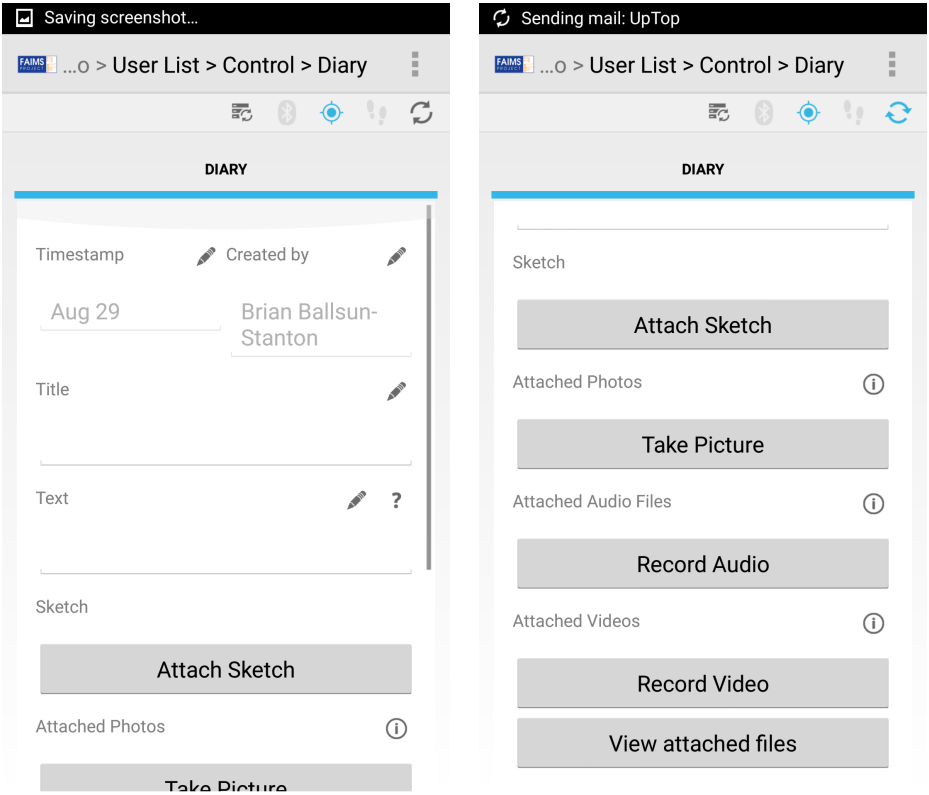

Figure 5.32 "Timestamp" and “Created by” are automatically filled in by FAIMS.

5.7 Additional Features

In no particular order, here are some other useful things you can do for your module in `module.xml`.

5.7.1 Add a Picture Gallery

Remember when we told you how to add a tarball of images to your module? If you use the element type "picture," you can allow user selectable options to be displayed as these images. Take a look at the tab in Codeblock 5.30: Each picture is referenced by the tag, then labeled with the non-tagged text inside the element.

```
<Script t="picture">
    <desc>Type of used script.</desc>
    <opts>
        <opt p="picture1.jpg">Archaic-Epichoric</opt>
        <opt p="picture2.jpg">Old-Attic</opt>
        <opt p="picture3.jpg">Ionic</opt>
        <opt p="picture4.jpg">Roman</opt>
        <opt p="picture5.jpg">Indistinguishable</opt>
        <opt p="picture6.jpg">Other</opt>
    </opts>
</Script>
```

Code 5.30 XML for a picture gallery.

5.7.2 Hierarchical Dropdown

"Hierarchical dropdown" is a fancy way of saying "selecting one option sometimes brings up more specific options." You do this very simply, by including **<opt>** tags inside of other **<opt>** tags. For example, Codeblock 5.31.

```
<Script t="dropdown">
    <desc>Type of used script.</desc>
    <opts>
        <opt>Archaic-Epichoric
            <opt>A specific type of archaic-epichoric script
                <opt>An even more specific type of that specific type
archaic-epichoric script</opt>
            </opt>
            <opt>Another type of archaic-epichoric script</opt>
        </opt>
        <opt>Old-Attic</opt>
        <opt>Ionic</opt>
        <opt>Roman</opt>
        <opt>Indistinguishable</opt>
        <opt>Other</opt>
    </opts>
</Script>
```

Code 5.31 XML for hierarchical dropdowns.

5.7.3 Using the Translation File

5.7.4 Autonumbering

Basic autonumbering can be achieved using a combination of the `f="autonum"` flag and the **`<autonum/>`** tag. By flagging an input with `autonum`, one indicates to the FAIMS-Tools that the ID of the next created entity—the entity containing the flagged field—should be taken from the corresponding field generated using the **`<autonum/>`** tag. For instance the `Creatively_Named_ID` in Codeblock 5.32 will take its values from a field in Control which is generated by the use of the **`<autonum/>`** tag.

The field will appear to the user as “Next Creatively Named ID” and will initially be populated with the number 1. When the user creates a `Tab_Group` entity, it will take that number as its “Creatively Named

```
<module>
    <Control>
        <Control>
            <Create_Entity t="button" l="Tab_Group" />
            <autonum/>
        </Control>
    </Control>
    <Tab_Group>
        <Tab>
            <Creatively_Named_ID f="id autonum" />
        </Tab>
    </Tab_Group>
</module>
```

Code 5.32 XML for autonumbering.

ID". The "Next Creatively Named ID" will then be incremented to 2, ready to be copied when a subsequent `Tab_Group` entity is created.

Multiple fields can be flagged as being autonumbered as in Codeblock 5.33.

5.7.5 Restricting Data Entry to Decimals for a Field

- Single flag to denote as a number field

5.7.6 Type Guessing for GUI Elements in FAIMS-Tools

The FAIMS-Tools `generate.sh` program will attempt to make a reasonable assumption about what the `t` attribute should be set to if it is omitted from a GUI element's set of XML tags.

If the XML tags do not contain a set of `<opts>` tags nor the `f="user"` flag, `t="input"` is assumed. For example, Codeblock 5.34.

If the XML tag is flagged with `f="user"`, `t="dropdown"` is assumed as in Codeblock 5.35.

```
<module>
    <Control>
        <Control>
            <Create_Entity t="button" l="Tab_Group" />
            <autonum/>
        </Control>
    </Control>
    <Tab_Group>
        <Tab>
            <Creatively_Named_ID f="id autonum" />
            <Creatively_Named_ID_2 f="id autonum" />
        </Tab>
    </Tab_Group>
    <Other_Tab_Group>
        <Tab>
            <Creatively_Named_ID_3 f="id autonum" />
        </Tab>
    </Other_Tab_Group>
</module>
```

Code 5.33 XML for autonumbering multiple fields.

```
<Entity_Identifier f="id"/> <!-- This'll be an input -->
```

Code 5.34 Without `<opts>` or `f="user"`, `t="input"` is assumed.

If the XML tags contain an `<opts>` element and no descendants with `p` attributes, `t="list"` is assumed. For example, Codeblock 5.36.

If the XML tags contain an `<opts>` element and one or more descendants with p attributes, `t="picture"` is assumed. For example, Codeblock 5.37.

Finally, if the XML tags have the `ec` attribute, `t="list"` is assumed.

There are arguments both for and against the use of the type guessing feature because, while improving succinctness, it also makes the `module.xml` file less intelligible to uninitiated programmers. Because of this, the FAIMS-Tools will display warnings when a module is generated from an XML file whose GUI elements are missing their `t` attributes. These

```
<List_of_Users f="user"/> <!-- This'll be a dropdown -->
```

Code 5.35 When flagged with `f="user"`, `t="dropdown"` is assumed.

```
<Element> <!-- This'll be a list -->
    <opts>
        <opt>Option 1</opt>
        <opt>Option 2</opt>
    </opts>
</Element>
```

Code 5.36 Tags containing **`<opts>`** with no descendants with `p` attributes are assumed to be lists.

can be hidden by adding `suppressWarnings="true"` to the opening **`<module>`** tag as in Codeblock 5.38.

5.7.7 Annotation and Certainty

5.7.8 Exporting Data

5.8 Advanced FAIMS Programming

5.8.1 `module.xml` Cheat Sheet

For more information about the different XML attributes, flags, and relationship tags, we have a README file that you can access online here: *https://github.com/FAIMS/FAIMS-Tools/blob/master/generators/christian/rea* or in the generators/christian/ directory where you downloaded the FAIMS-Tools.

We'll repeat some of the information here for your reference. Be sure to open the README for the most up to date information: to learn how to include more advanced controls and scripting in your modules, look

at the *FAIMS Development Cookbook*, which includes code snippets for all of the things FAIMS can do.

```
                    FAIMS TOOLS' MODULE AUTOGENERATOR

          The FAIMS-Tools module autogenerator ("autogen") produces
          FAIMS definition files (e.g. ui_logic.bsh, ui_schema.xml,
             data_schema.xml, etc) from a single module.xml file.

DEPENDENCIES AND SETUP

  The FAIMS-Tools autogen has the following recommended requirements:
    1. A Debian-based OS released 2016 or later.
    2. Any packages installed via `./install-dependencies.sh`, which can be
       found in the same directory as this readme.
  If the above requirements cannot be satisfied, you might still be able to use
  the FAIMS-Tools autogen. See 'Alternative Setup (Docker)' for more
  information. Older distros (around 2014 or earlier) typically produce working
  modules, however the wireframes don't render correctly. Ubuntu 14.04, for
  example, is known to produce working modules.

  The following OSes are known to work with the FAIMS-Tools autogen completely,
  including the correct generation of wireframes:
    - Debian 8
    - Debian 9
    - Ubuntu 16.04
    - Ubuntu 17.10
    - Ubuntu 18.04
--------------------------------------------------------------------------------

ALTERNATIVE SETUP (DOCKER)

  If you have Docker installed, you can use `docker-generate.sh` and
  `docker-validate.sh` in place of `generate.sh` and `validate.sh`,
  respectively. The `docker-*.sh` scripts merely run the `generate.sh` and
  `validate.sh` scripts within a Docker container.

  The FAIMS-Tools autogen was tested to work with with Docker 18.03.1-ce, build
  9ee9f40. Other versions are likely to work as well.
--------------------------------------------------------------------------------

QUICK START

  1. Modify the example `module.xml`. If you see something there which you don't
     understand, check this file.
  2. Run `./generate.sh`. (If your module.xml isn't in this directory, you can
     also run `./generate.sh /path/to/module.xml` from this directory, or
     `path/to/generate.sh module.xml` from the module.xml directory.) The
```

```
<Element> <!-- This'll be a picture gallery -->
    <opts>
        <opt p="Lovely_Image.jpg>Option 1</opt>
        <opt >Option 2</opt>
    </opts>
</Element>
```

Code 5.37 Tags containing **<opts>** and one or more descendants with p attributes are assumed to be picture galleries.

```
<module suppressWarnings="true">
    <!-- Tab groups go here... -->
</module>
```

Code 5.38 XML to suppress warnings.

```
    generated module will appear in a directory called `module` in the same
    directory as the module.xml file.
--------------------------------------------------------------------------------

PARAMETERS AND USAGE

  generate.sh [path] [-w|--wireframe]
  generate.sh [-w|--wireframe] [path]
      Generates a module in a directory called `module`, in the same directory
      as a module.xml file.

      path
             A path to a module.xml file. This is relative to the current
             working directory. If no path is given, the autogen will look for
             a module.xml file in the current working directory.

      -w, --wireframe
             Compiles the module with a wireframe. The wireframe is stored in
             the wireframe/wireframe.pdf directory, relative to the module.xml
             file.

  validate.sh [path]
      path   A path to a module.xml file. This is relative to the current
             working directory. If no path is given, the autogen will look for
             a module.xml file in the current working directory.
--------------------------------------------------------------------------------

SCHEMA ATTRIBUTES

  b                  Binding (See 'Bindings')
  c                  Alias for faims_style_class
  e="Type"           Populates the menu with entities of the type `Type`. If the
                     `Type` is the empty string, entities of all types are shown.
  ec, lc             (See 'Child Entities'. See also, 'QR Codes and Barcodes'.)
  f                  Flags (See 'Flags')
  i                  Inherit (copy) the value of the field whose path is given
                     as the value of the `i` attribute. By default the field
                     value is only copied if the given field is in a parent
                     tab group (See 'Child Entities'). This safeguard can be
                     disabled by including an exclamation point in the path, as
                     follows: i="path/to/field!".
  l                  Link to tab or tab group in the format Tabgroup/Tab/. Links
                     to tabs are discouraged as the generated code will contain
                     a race condition. Autogenerated code containing tab links
                     should be thoroughly tested. (See also, 'QR Codes and
```

```
                    Barcodes'.)
  ll                Login link. Works the same as the `l` attribute, except the
                    user is prompted to enter their username and password
                    before the link is followed. The link is only followed if
                    the user successfully enters their username and password.
  lq                When used on a clickable UI element, this displays
                    Android's QR scanner. The scanned QR code is parsed to find
                    the first substring which has the format of a UUID. If a
                    UUID is found in the string, and a record exists on the
                    device which has that UUID, the record is loaded and
                    displayed. The user is notified if the record does not
                    exist.
  p                 In <opt> tags, equivalent to pictureURL attribute, however
                    "files/data/" is prepended.
  suppressWarnings  Deprecated. Used to prevent warnings from being shown when
                    equal to "true" and present in the <module> tag. Does not
                    suppress errors.
  sp, su            In <opt> tags and GUI elements, equivalent to
                    SemanticMapPredicate and SemanticMapURL attributes,
                    respectively.
  t                 Type of GUI element (See 'Types'). If this attribute is
                    omitted, FAIMS Tools will attempt to infer it from the
                    element's content. (See 'Type Guessing'.)
  test_mode         When this is equal to "true" and present on the <module>
                    tag, the module will be compiled with performance testing
                    enabled. Performance testing mode profiles queries and adds
                    the ability to generate records en masse from the module's
                    login tab group.
  vp                Vocabulary population. Populates the field having the vp
                    attribute with the vocab of the field whose path is the
                    value of the vp attribute.
--------------------------------------------------------------------------------

BINDINGS

  - date
  - decimal
  - string
  - time
  Other bindings are possible (e.g. by writing b = "my-binding") but generate a
  warning.
--------------------------------------------------------------------------------

FLAGS
```

Flags are specified using the `f` attribute. For in the following, the `id` and `noannotation` flags are used:

```
<My_Identifier f="id noannotation"/>
```

Although the above example shows the use of flags on a GUI/Data element, flags can also be used on tabs and tab groups. The complete list of flags is given here:

autonum	For use with <autonum> tag. (See 'Autonumbering'.)
hidden	Equivalent to faims_hidden="true".
htmldesc	Equivalent to faims_html_description="true"
id	Equivalent to isIdentifier="true".
noannotation	Equivalent to faims_annotation="false".
noautosave	Prevents a tab group from being automatically saved when the user navigates away from it or changes its contents.
nocertainty	Equivalent to faims_certainty="false".
nolabel	Prevents labels from being displayed or generated from element names.
nosync	Removes the faims_sync="true" attribute from audio, camera, file and video GUI elements.
nothumb[nail]	Removes the thumbnail="true" attribute from audio, camera, file and video elements in the data schema.
noscroll	Equivalent to faims_scrollable="false".
noui	Only allows code related to the data schema to be generated.
nodata	Generates code as usual, but omits data schema entries.
nowire	Excludes a tab group, tab, or GUI element from the wireframe.
readonly	Equivalent to faims_read_only="true".
persist	Causes field value to persist over multiple sessions on the user's device.
user	Used to indicate that a menu should contain a list of users.
notnull	Adds client- and server-side validation specifying that the field should not be left blank.

TYPE GUESSING

FAIMS Tools will attempt to make a reasonable assumption about what the t attribute should be set to if it is omitted from a GUI element's set of XML tags.

If the XML tags do not contain a set of <opts> tags nor the f="user" flag, t="input" is assumed. Example:

```
<Entity_Identifier f="id"/>          <!-- This'll be an input -->
```

```
  If the XML tag is flagged with f="user", t="list" is assumed. Example:

      <List_of_Users f="user"/>            <!-- This'll be a list -->

  If the XML tags contain an <opts> element and no descendants with p
  attributes, t="dropdown" is assumed. Example:

      <Element>                            <!-- This'll be a dropdown -->
        <opts>
          <opt>Option 1</opt>
          <opt>Option 2</opt>
        </opts>
      </Element>

  If the XML tags contain an <opts> element and one or more descendants with p
  attributes, t="picture" is assumed. Example:

      <Element>                            <!-- This'll be a picture gallery -->
        <opts>
          <opt p="Lovely_Image.jpg>Option 1</opt>
          <opt                     >Option 2</opt>
        </opts>
      </Element>
--------------------------------------------------------------------------------

TYPES

  Types of GUI element:
    - audio        <select type="file" faims_sync=true/>
                   <trigger/>
    - button       <trigger/>
    - camera       <select type="camera" faims_sync=true/>
                   <trigger/>
    - checkbox     <select/>
    - dropdown     <select1/>
    - file         <select type="file" faims_sync=true/>
                   <trigger/>
        File list with a button to add a file

    - gpsdiag      <input faims_read_only="true"/>...
    - group        <group/>
    - input        <input/>
    - list         <select1 appearance="compact"/>
    - map          <input faims_map="true"/>
    - picture      <select1 type="image"/>
```

```
    - radio         <select1 appearance="full"/>
    - video         <select type="file" faims_sync=true/>
                    <trigger/>
    - viewfiles     <trigger/>
        A button to view all files related to an archent.
    - web[view]     <input faims_web="true"/>
--------------------------------------------------------------------------------

RESERVED ELEMENT NAMES AND RECOMMENDED NAMING CONVENTIONS

  "Reserved" elements only contain lowercase letters:
    - <autonum>     A group of fields containing the next ID's of the inputs
                    marked with f="autonum".  (See 'Autonumbering'.)
    - <col>         One column in a <cols> tag
    - <cols>        Columns
    - <desc>        Description to put in the data schema
    - <logic>       UI logic which is appened to end of generated file.
    - <module>
    - <markdown>    Placed as the direct child of a t="webview" element. This
                    element's text is interpreted as pandoc markdown. It is used
                    to populate its corresponding webview.
    - <opt>         Option in <opts> tag
    - <opts>        Options for, say, a dropdown menu
    - <rels>        Intended to be a direct child of <module> and hold
                    <RelationshipElement> tags
    - <gps>         A set of fields including Latitude, Longitude, Northing,
                    Easting and a "Take From GPS" button.
    - <search>      A tab for searching all records. Its text is used as a label.
    - <str>         Contains <formatString>-related data.
    - <pos>         When the child of a <str>, gives the position (order) of an
                    identifier in a formatted string
    - <fmt>         When <str> is the parent of <fmt>, <fmt> contains the text of
                    <formatString>, which gets copied (almost) verbatim to the
                    generated data schema.

                    The <fmt> tag may also appear as the child of a tab group. In
                    this case FAIMS Tools parses the tag's text before adding it
                    to the data schema. The parsing algorithm is outlined under
                    'The FAIMS Tools Format-String Specification'.
    - <app>         When the child of a <str>, contains <appendCharacterString>
                    data.
    - <author>      A read-only field displaying the username of the current user
                    or a message if the entity it appears in has not been saved.
    - <timestamp>   A read-only field displaying the creation time of the entity
                    it appears in.
```

```
  User-defined elements should start with an uppercase letter and use
  underscores as separators:
    - <My_User_Defined_Element t="dropdown" />
  Neither of these naming conventions are strictly enforced however.
--------------------------------------------------------------------------------

INTENDED PURPOSE OF THE <rels> TAG

  When placed as a direct child of the <module> element, contents of the <rels>
  tag are copied as-is to the generated data schema. No warnings are shown if
  something is awry with its contents.

  Because the <rels> tags' contents are directly copied, in principle you could
  put anything in there which you want to appear in the data schema.
--------------------------------------------------------------------------------

SEMANTICS OF <cols> TAGS

  Direct children of <cols> tags are interpreted as columns. For example,
      <cols>
        <Field_1 t="input"/>
        <Field_2 t="input"/>
        <Field_3 t="input"/>
      </cols>
  has three columns, each containing an input. The left-most column is Field_1,
  whereas the right-most is Field_3.

  When a <col> tag is a direct child, its contents are interpreted as being part
  of a distinct column. Therefore,
      <cols>
        <Field_1 t="input"/>
        <col>
          <Field_2 t="input"/>
          <Field_3 t="input"/>
        </col>
      </cols>
  results in two columns. The left column contains Field_1, while the right
  contains Field_2 and Field_3.
--------------------------------------------------------------------------------

THE FAIMS TOOLS FORMAT-STRING SPECIFICATION

  The fundamental format-string specification can be found in the 'FAIMS Data,
  UI and Logic Cook-Book'
(https://faimsproject.atlassian.net/wiki/display/FAIMS/FAIMS+Data%2C+UI+and+Logic+Cook-
Book#FAIMSData,UIandLogicCook-Book-AttributeFormatString).
```

The reader is encouraged to familiarise themselves with it prior to learning how FAIMS Tools augments it.

Like fundamental format-strings, the FAIMS Tools format-string specification controls the way a record is displayed when it appears in a list. It specifies which of the record's fields should be displayed in those lists, what order those fields should appear in, and so forth. This is done by referring to the fields using double-curly-brace notation in a <fmt> tag. For instance, to refer to a field with the tag name `My_Field`, the programmer would write "{{My_Field}}" in the (FAIMS Tools) format-string. The following code snippet shows an example of this:

```
<My_Record>
  <fmt>{{My_Field}}</fmt>
  <Tab_1>
    <My_Field/>
  </Tab_1>
</My_Record>
```

When a record of this type is displayed in a list of search results, it will be shown as the saved contents of "My_Field".

Text can be interspersed with references to fields to create more informative format-strings:

```
<Dimensions>
  <fmt>{{Title}} - Depth: {{X}}, Height: {{Y}}, Width: {{Z}}</fmt>
  <Dimensions>
    <X/>
    <Y/>
    <Z/>
    <Title/>
  </Dimensions>
</Dimensions>
```

When the user saves a "Dimensions" record with "Title", "X", "Y", and "Z" equal to "Artefact Size", "3", "4" and "1", respectively, it will appear in a list of search results as "Artefact Size - Depth: 3, Height: 4, Width: 1".

The syntax for conditional formatting is similar to that of fundamental format strings, except the programmer must write the desired field immediately after opening the double curly braces. For instance, in the previous example, if the programmer only wished to display the "Depth: {{X}}" part when X was not zero, the format string would become:

```
        <fmt>{{Title}} - {{X if not(equal($2, 0)) then "Depth: $2"}}, Height:
{{Y}}, Width: {{Z}}</fmt>
```

Note that there cannot be a space between the opening curly braces and the field's tag name. For instance "{{ X if not(..." would fail to be parsed as one might expect. Notice also that in the if statement itself, fundamental-style dollar-sign notation (e.g. "...equal($2, 0)...") is used to refer to the field's value, as opposed to writing, for instance "equal({{X}}, 0)".

Although <fmt> tags can be used to define the FAIMS Tools format-strings discussed here, they can also be used to define fundamental style format-strings. See 'Reserved Element Names and Recommended Naming Conventions' for an outline on how this can be achieved.

AUTONUMBERING

Basic autonumbering can be achieved using a combination of the f="autonum" flag and the <autonum/> tag. By flagging an input with `autonum`, one indicates to FAIMS Tools that the ID of the next created entity---the entity containing the flagged field---should be taken from the corresponding field generated using the <autonum/> tag. For instance the Creatively_Named_ID in the below module will take its values from a field in Control which is generated by the use of the <autonum/> tag.

```
    <module>
      <Control>
        <Control>
          <Create_Entity t="button" l="Tab_Group" />
          <autonum/>
        </Control>
      </Control>
      <Tab_Group>
        <Tab>
          <Creatively_Named_ID f="id autonum" />
        </Tab>
      </Tab_Group>
    </module>
```

The field will appear to the user as "Next Creatively Named ID" and will initially be populated with the number 1. When the user creates a Tab_Group entity, it will take that number as its "Creatively Named ID". The "Next Creatively Named ID" will then be incremented to 2, ready to be copied when a subsequent Tab_Group entity is created.

Multiple fields can be flagged as being autonumbered like so:

```
    <module>
      <Control>
        <Control>
          <Create_Entity t="button" l="Tab_Group" />
          <autonum/>
        </Control>
      </Control>
      <Tab_Group>
        <Tab>
          <Creatively_Named_ID   f="id autonum" />
          <Creatively_Named_ID_2 f="id autonum" />
        </Tab>
      </Tab_Group>
      <Other_Tab_Group>
        <Tab>
          <Creatively_Named_ID_3 f="id autonum" />
        </Tab>
      </Other_Tab_Group>
    </module>
```

CHILD ENTITIES

Entities can be saved as children by the use of the "lc" attribute. For instance, writing

```
    <Add_Child t="button" lc="Child_Ent" />
```

generates a button which links to the Child_Ent tab group. When displayed by clicking the Add_Child button, the Child_Ent tab group will have auto-saving enabled and be saved as a child of the tab group that the button appeared in. For example, consider the following module.xml:

```
    <module>
      <Tab_Group>
        <Tab>
          <Add_Child t="button" lc="Tab_Group" />
        </Tab>
      </Tab_Group>
    </module>
```

Clicking the Add_Child button will cause the user to be taken to a new instance of Tab_Group which will be saved as a child of the original instance. But because the original instance was not loaded by clicking the button, it will not be saved as a child.

A list of child entities can be displayed to the user by using the "ec" attribute:

```
    <List_Of_Related_Entities t="list" ec="Type_Of_Childen" />
```
The list will be populated with entities which are children of the tab group the list appears in. The entities will be constrained to have the type "Type_Of_Children". However, writing `ec=""` produces an unconstrained list, where children of all types are displayed.

The reader should note carefully that, while including an "lc" attribute causes a corresponding <RelationshipElement> to be generated in the data schema, including an "ec" attribute does not.

LABELS

An element's text is taken as its label. For instance, the following input

```
    <My_Input t="input">
      Droopy Soup
      <desc>Similar to drippy soup, but not quite...</desc>
    </My_Input>
```

has the label "Droopy Soup". Note that following and preceding whitespace is stripped.

If a label is not provided, it is "inferred" from the element's name. More specifically, underscores in the element's name are replaced with spaces, which becomes the element's label. Therefore, the element

```
    <Droopy_Soup t="input">
      <desc>Similar to drippy soup, but not quite...</desc>
    </Droopy_Soup>
```

has the same label as in the above example. Thus, the user will see exactly the same thing in both cases. However, their representations in the data and UI schemas, and the arch16n file will be different.

You are recommended to use this "inference" feature, as it encourages consistency between the label, which the user sees, and the view's reference and faims_attribute_name, which the programmer sees. Note that it merely "encourages" consistency as the programmer can change the corresponding, generated, arch16n (english.0.properties) entry.

GENERATION OF THE ARCH16N FILE

Labels and menu options (e.g. from checkboxes and dropdowns) have arch16n entries generated for them. The left-hand side of an arch16n entry (i.e. everything to the left of the equals sign) is produced by changing all non-alphanumeric characters in the label or menu option to underscores. The right-hand side is the unmodified text.

6 Deploy and Debug Modules

7 Finis