FAIMS User to Developer Documentation

Welcome to the FAIMS User-to-Developer Guide!

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. http://context.fedarch.org/Context/UserToDev/UserToDev.pdf

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.

Test Your Knowledge:

What does this guide teach you how to create?

What should you do if you don't know the answer to a Test Your Knowledge Question?

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.

What does the module do?

On the user's end, a module looks like a paper form used for data collection, only digital. The module has fields in which a variety of data can be entered, ranging from drop-down menu selections to longer-form responses to photographs and videos. What the user sees, what data the user is asked to provide, and even what kinds of data the user is 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 collected, by whom, and how the information is related.

Why should you use a FAIMS database? Why not stick with older analog methods?

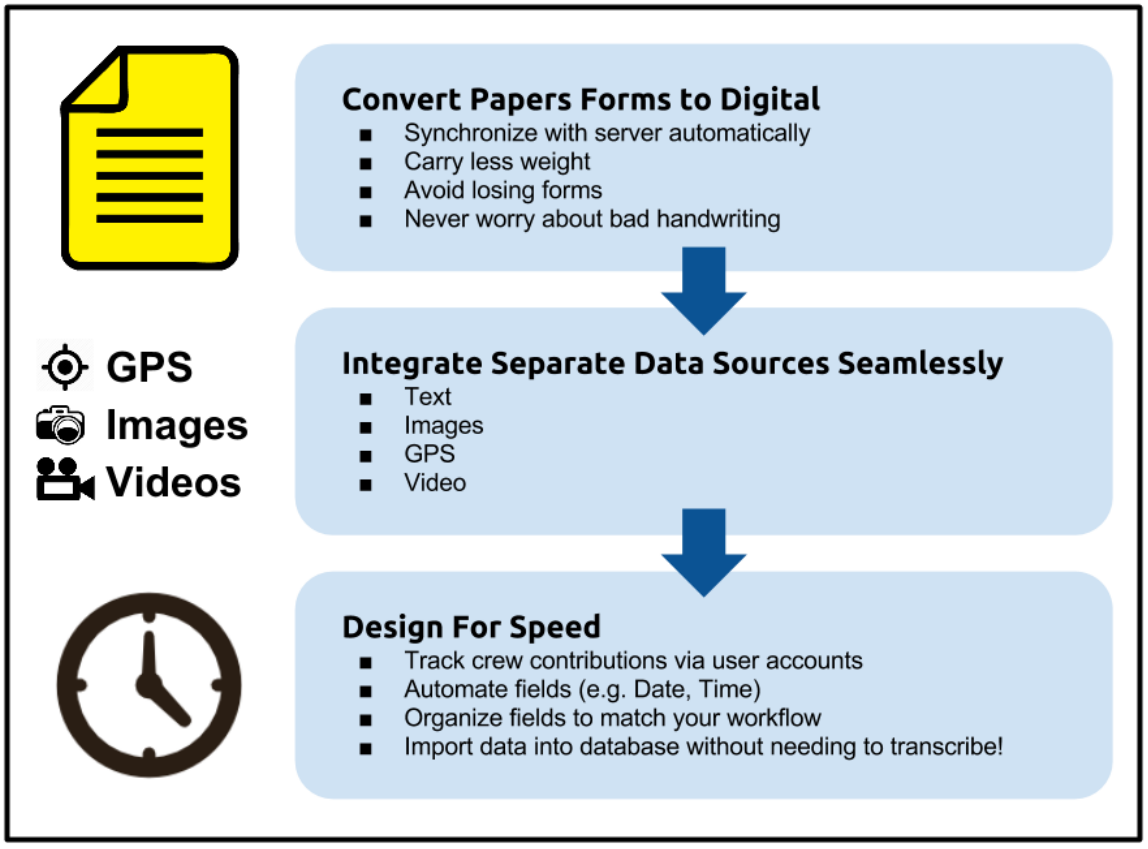
When projects rely on combinations of multiple data-gathering methods, ranging from paper forms, map notes, GPS readouts, and photographs, it eventually becomes necessary to organize the data into physical files, folders, and electronic databases. Even when the data can be organized centrally, so that everything collected can be found in the same place and format, context--where something was found, when it was found, who recorded it, how it relates to other pieces of collected data--is frequently lost.

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 attach directly video, audio, and other files such as PDFs and electronic sketches.

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, another 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!

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 like 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. Examples may include tracking individual users via 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.



The FAIMS app allows you to start out simply digitizing your paper forms, then add advanced features like video and file attachments as you need them.

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

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.

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

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

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 know that a location, like Site A, can have multiple different birds or no birds at all. The birds found in the location are understood as instances of data (a record) under another record called "Site A". We call this a *parent-child relationship.*

We know also that Bird 1 has red and white wings as well as blue plumage. Unlike the example above, in which a location could have zero or more birds, it is not in doubt whether the bird has wings or plumage. It will have both, and the form expects both 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*.

The problem is that forms that make sense to humans often don't make sense to computers. This form, for example, is not 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. It 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.

How does FAIMS handle multiple users collecting data at the same time?

FAIMS has no problem handling simultaneous usages or submissions. When two users submit at the same time, FAIMS automatically takes note of who uploaded it and when and preserves both in the data.

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

How can I 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:

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

...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:

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

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.

Test your knowledge questions:

What's the difference between data as stored by FAIMS and data as stored by a spreadsheet?

If two researchers enter data at the same time, does one of them "save over" the other's work?

Can you view collected data via your tablet?

Making Your Own Modules

In summary, 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 create the module you want.

-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.

The Parts of a Module

Modules are made from parts, called "necessary files."

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 (as you'll learn in the next section). Each necessary file serves a definite and distinct function in the final operation of the module.

You can simply use a module without ever learning what they 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.

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.

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.

Validation Schema

The Validation Schema, "validation.xml", defines what kinds of data your team *should* be collecting. It allows the module to "validate," or proofread, 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 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.

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.

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.

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.

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.

Test your knowledge:

If data entered by a user isn't valid, is it collected?

What necessary file **cannot be altered once a module has been created?**

Which necessary file **do you need to worry about if some of your team speaks English and some only French?**

Tour from the Data Schema

How to format data schema for the FAIMS system

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.

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.

Archaeological Elements 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.

Below is an example of an Archaeological Element. You don't really need to understand this yet, but it won't hurt to familiarize yourself with its structure.

```
<ArchaeologicalElement name="small">
    <description>
      An small entity
    </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</term>
        <term>Location B</term>
        <term>Location C</term>
        <term>Location D</term>
      </lookup>
    </property>
  </ArchaeologicalElement>
```

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."

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:

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

```
    </description>
    <parent>
      Above
    </parent>
    <child>
      Below
    </child>
    <property type="string" name="relationship" isIdentifier="true">
    </property>
     <property type="string" name="name">
    </property>
    <property type="dropdown" name="location">
      <lookup>
        <term>Location A</term>
        <term>Location B</term>
        <term>Location C</term>
        <term>Location D</term>
      </lookup>
    </property>
  </RelationshipElement>
```

How to Code Modules

Setting Up Your Development Environment

Before coding new modules, you'll need to download several (free) programs and files. You'll also need to configure some of these programs to make coding new modules easier and more efficient. At the end of this section, 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.

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. In these cases, 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 order to find equivalents.

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

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. Delete files, especially large media files, or uninstall programs until you have enough free space.

Installing the FAIMS Server and Virtual Machine (VM)

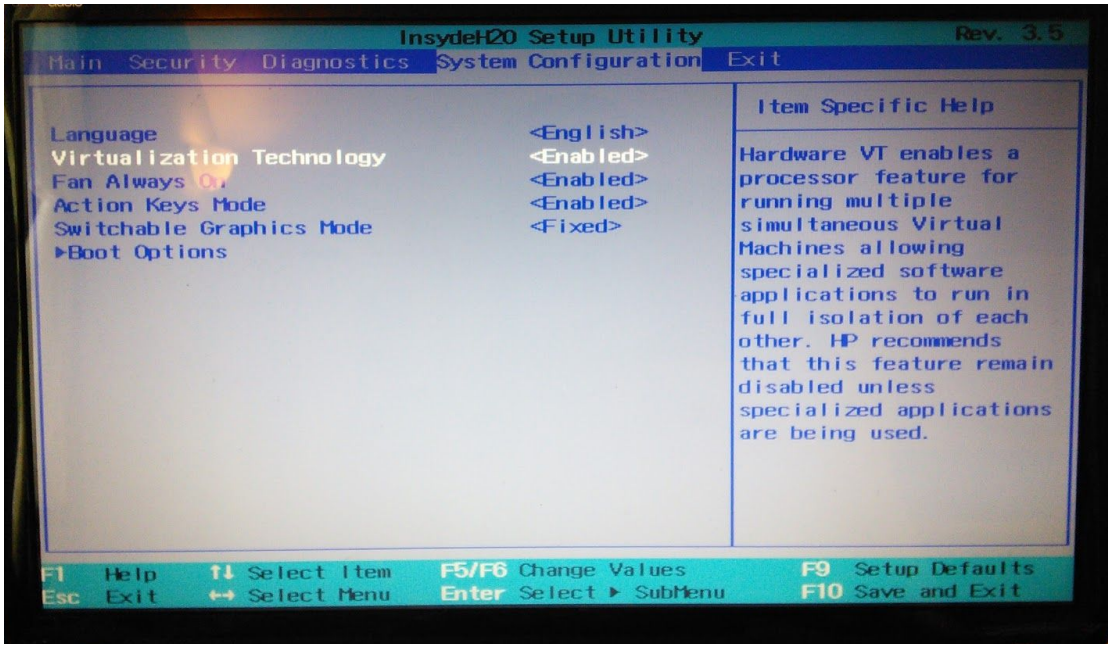
For various reasons, the FAIMS server is designed to run on a machine running an operating system called Ubuntu Linux 14.04. You probably don't use Ubuntu, and you probably wouldn't like to use it all of the time. That's why our first task here is to set up a "virtual machine" on the computer you do 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.

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/



We found the VT setting under the "Virtualization Technology" option in the System Configuration menu. Again, unless you're using a on an HP dv6qt Laptop, yours will probably be somewhere else.

Installing VirtualBox

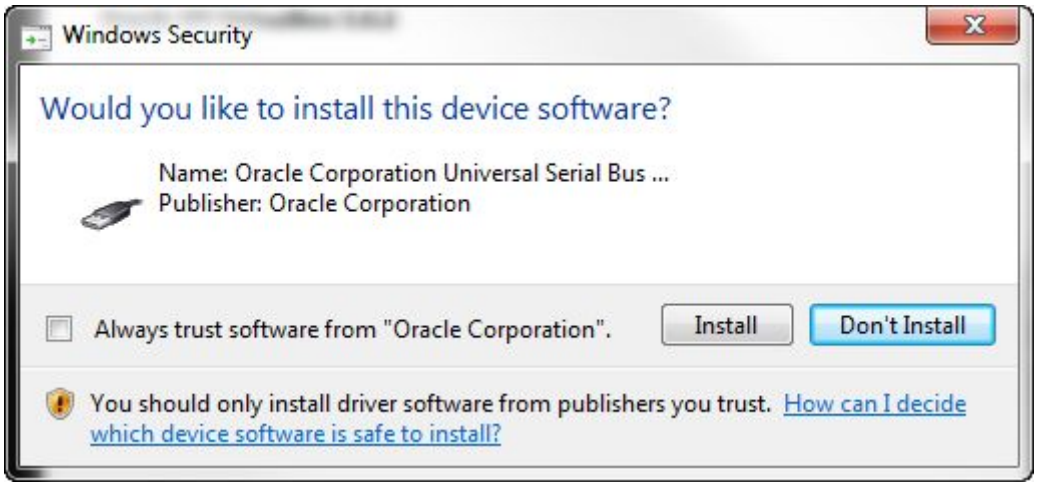
Now you can download and install the VirtualBox client, which we'll use to create a virtual machine to run the Ubuntu Linux-based FAIMS server.

https://www.virtualbox.org/wiki/Downloads

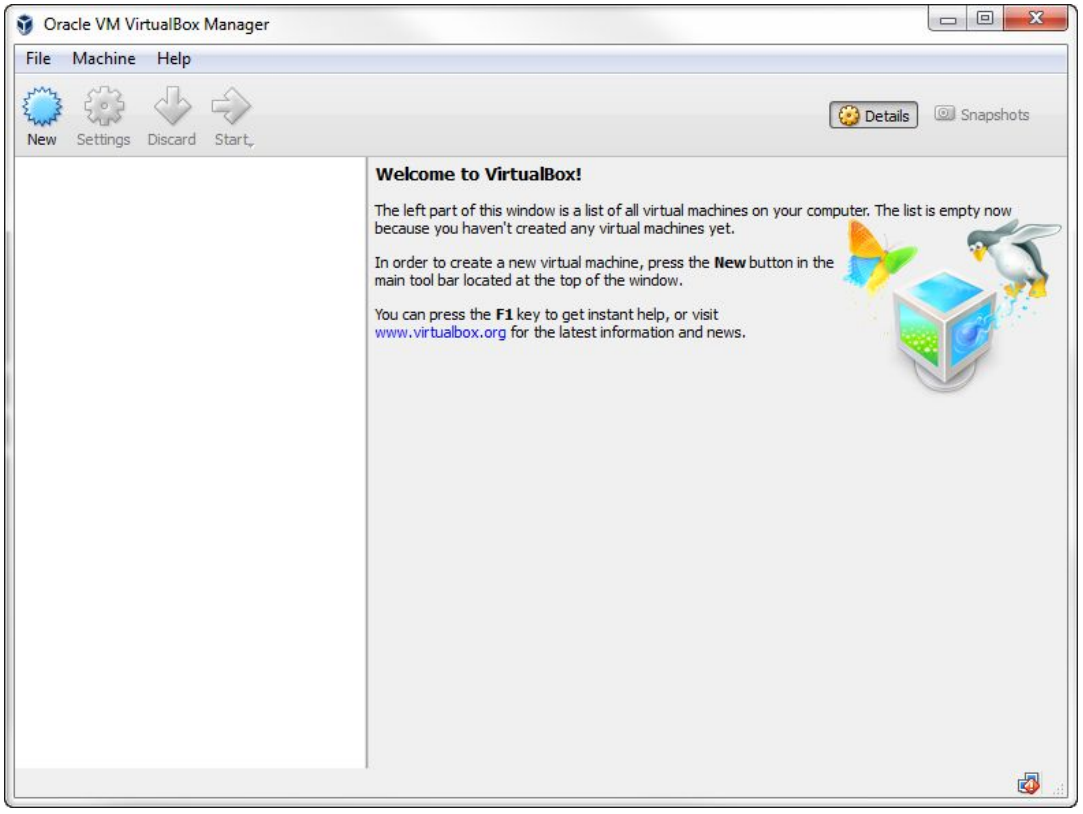
You probably won't have installed VirtualBox before, but on the off chance you have (say, if this isn't your first time trying this step) make sure you uninstall the old version **completely** before trying to install it again.

Note: As of 8/2015, VirtualBox is not compatible with Windows 10. If your operating system is Windows 10, you'll need to set up VirtualBox on a different machine.

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.



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

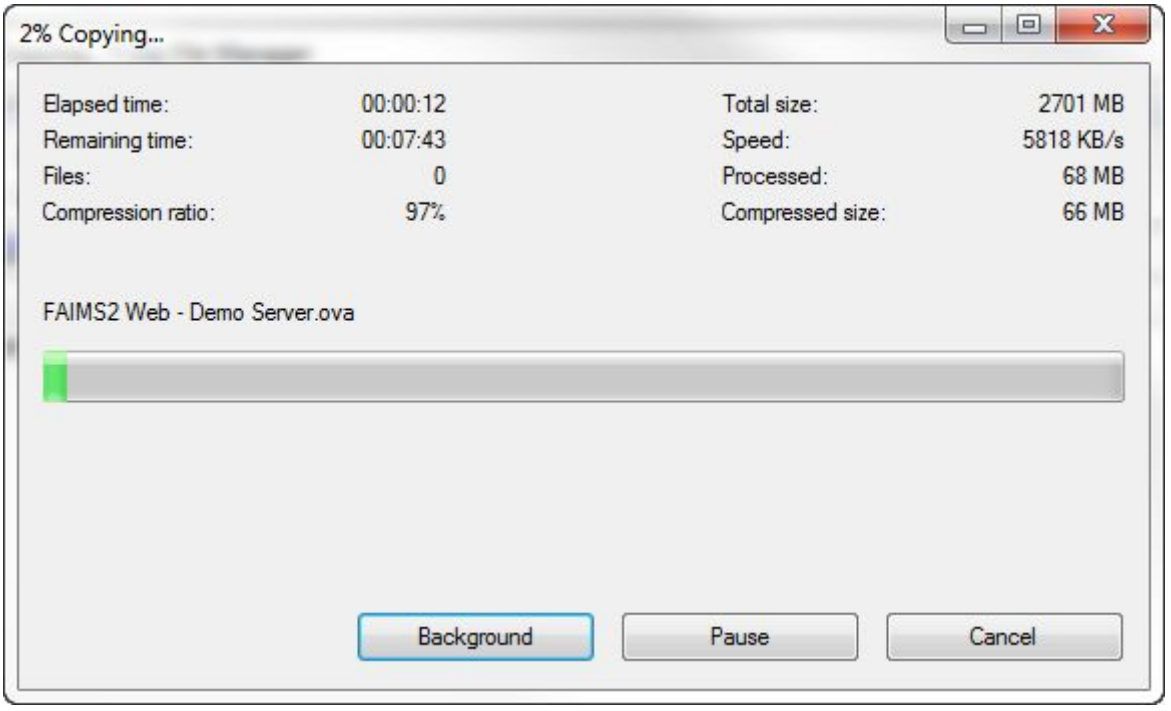


If you need to take a break, this is a good stopping point.

Next, download FAIMS Server image (2.5GB) from:
https://cloudstor.aarnet.edu.au/plus/index.php/s/OToMafgHrpsvwTG

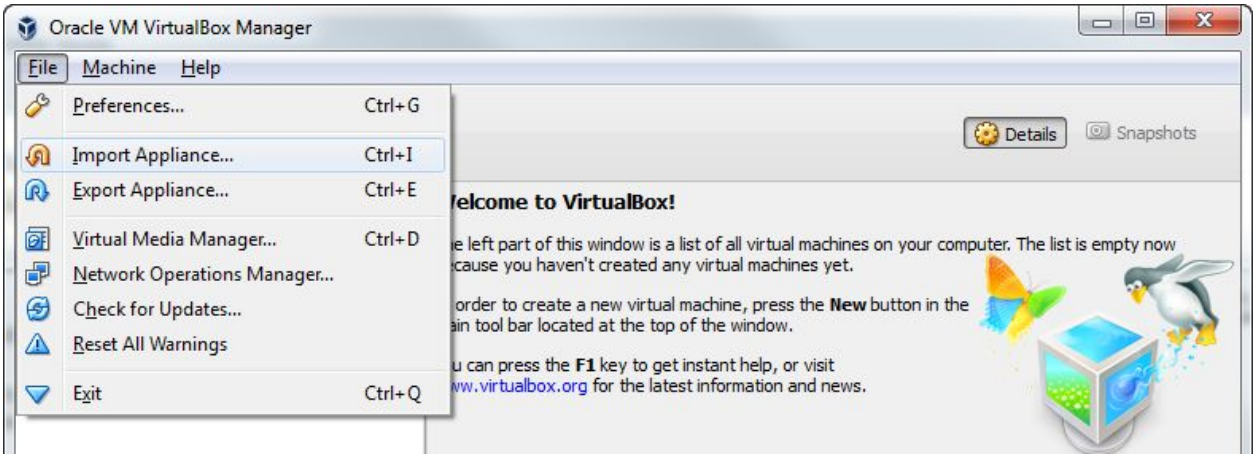
The server image is packaged as an ".ova" file, which is a type of file that can be opened by VirtualBox. This file is the complete package: it represents an Ubuntu Linux 14.04 computer system that has the FAIMS server set to automatically configure and run. This image is the same one that our FAIMS-In-A-Box kits use, which will make supporting your FAIMS installation much easier if you run into any problems.

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

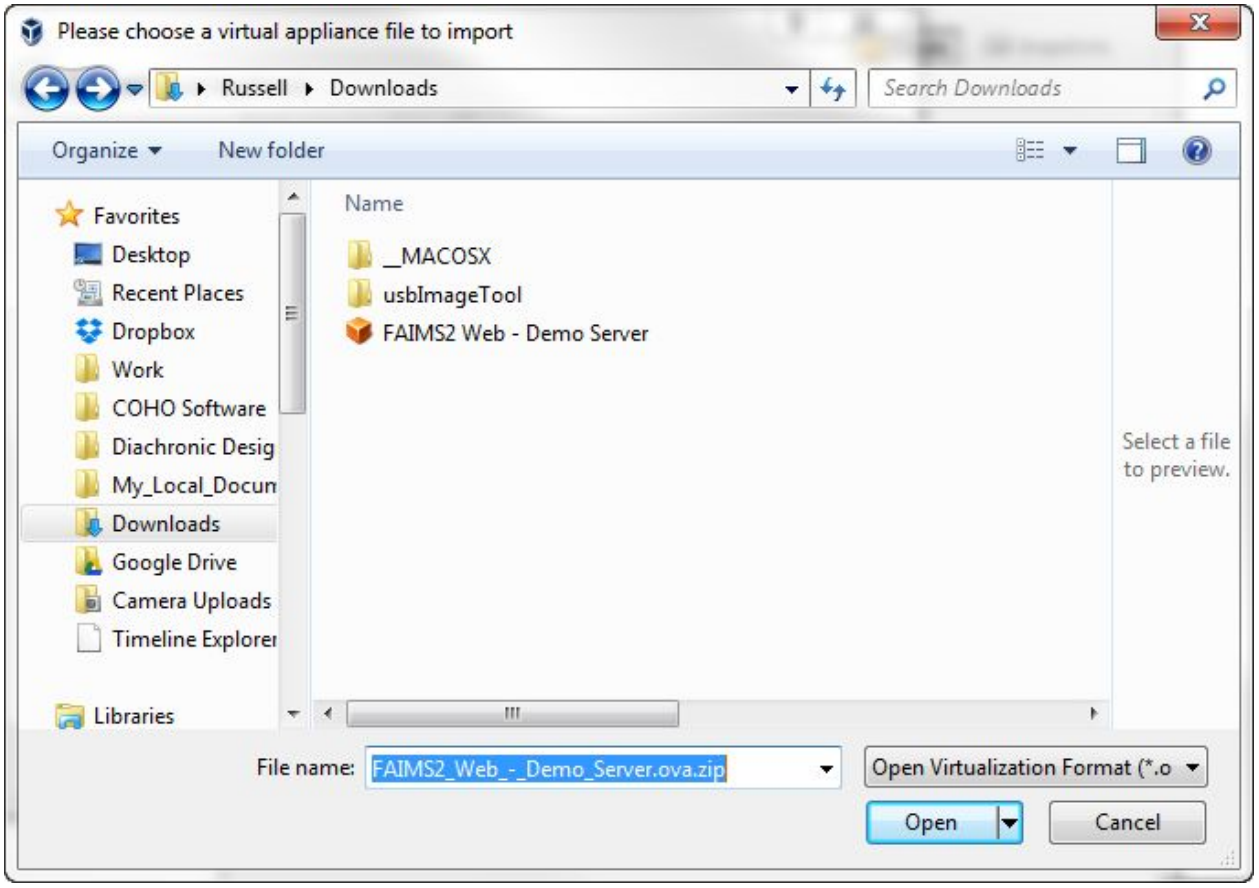


To set up the emulated machine, we'll use the preconfigured machine image we downloaded in the last step.

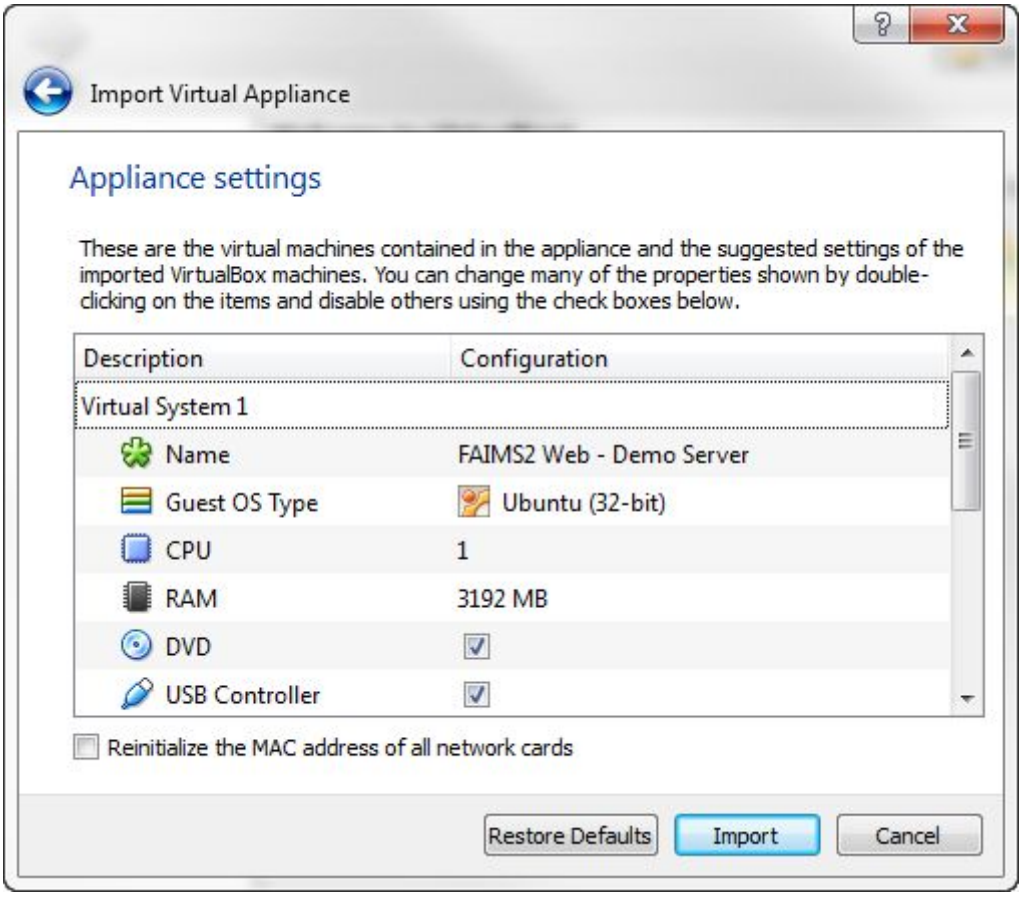
In VirtualBox, select the "File->Import Appliance" option.



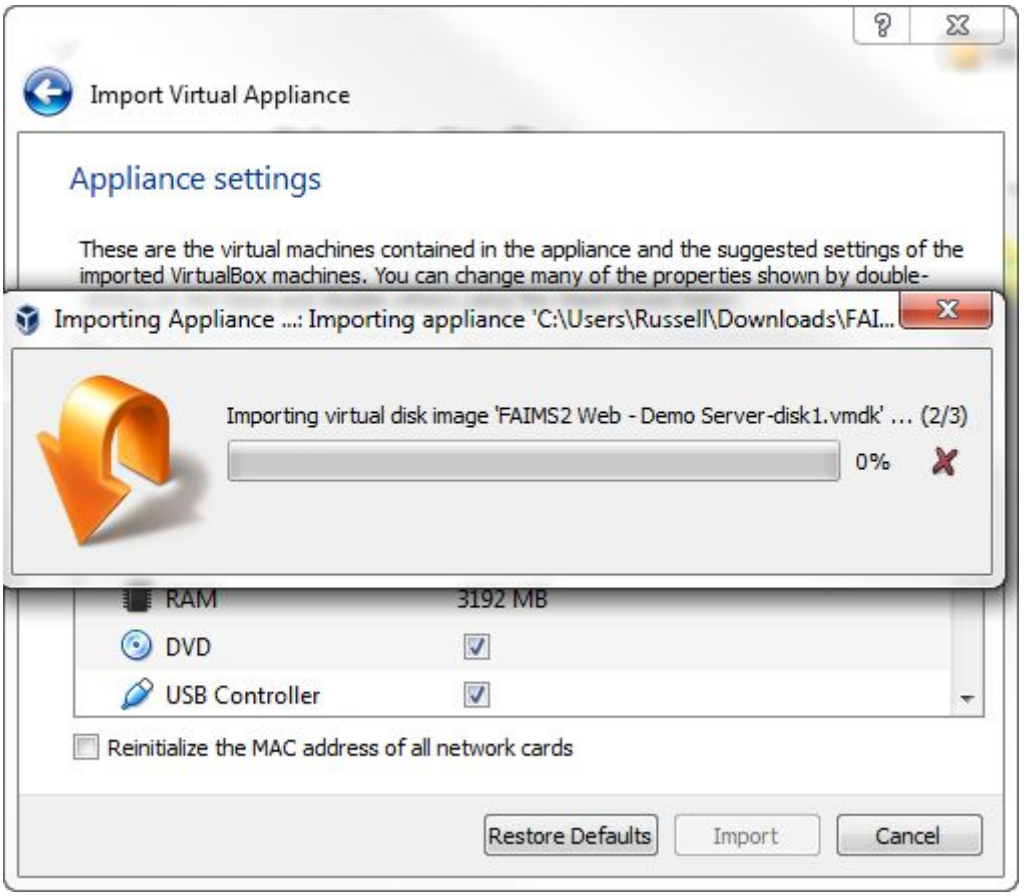
Using the file browser that appears, navigate to the directory where you downloaded the FAIMS server image and select to open it.



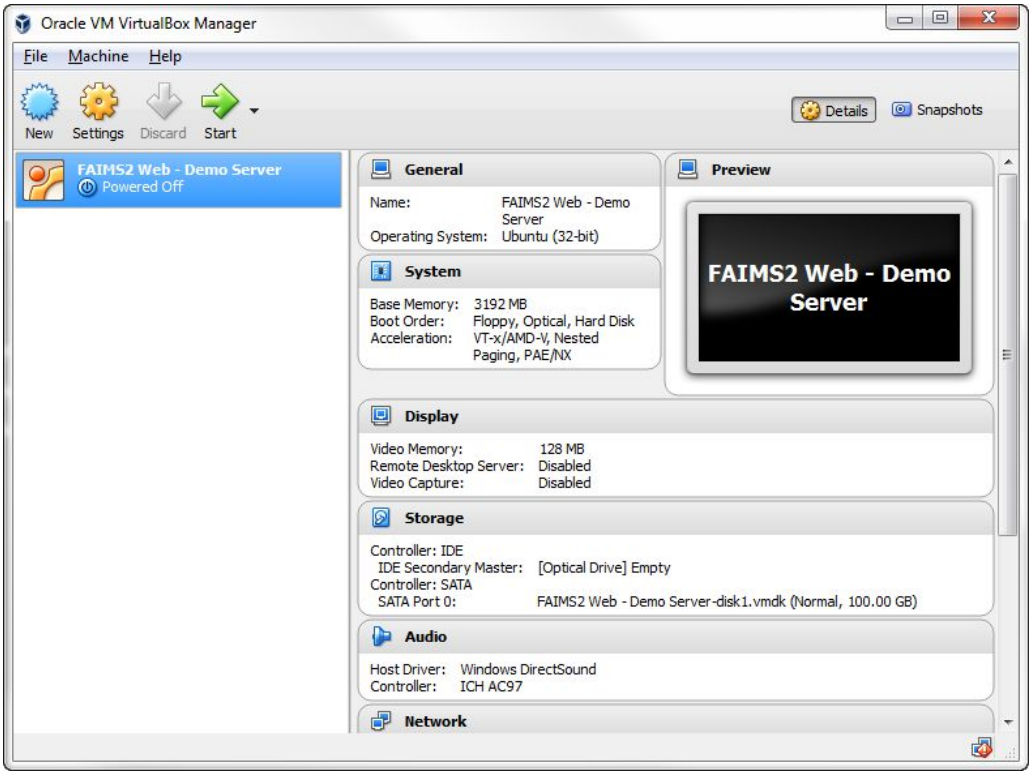
The pre-configured settings will appear. Select the “Import” option.



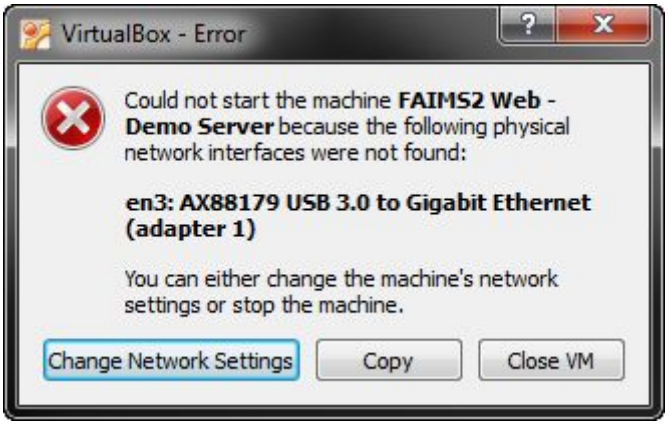
VirtualBox will now set to work importing the image and setting everything up. Depending on your system, this may take a few minutes.



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

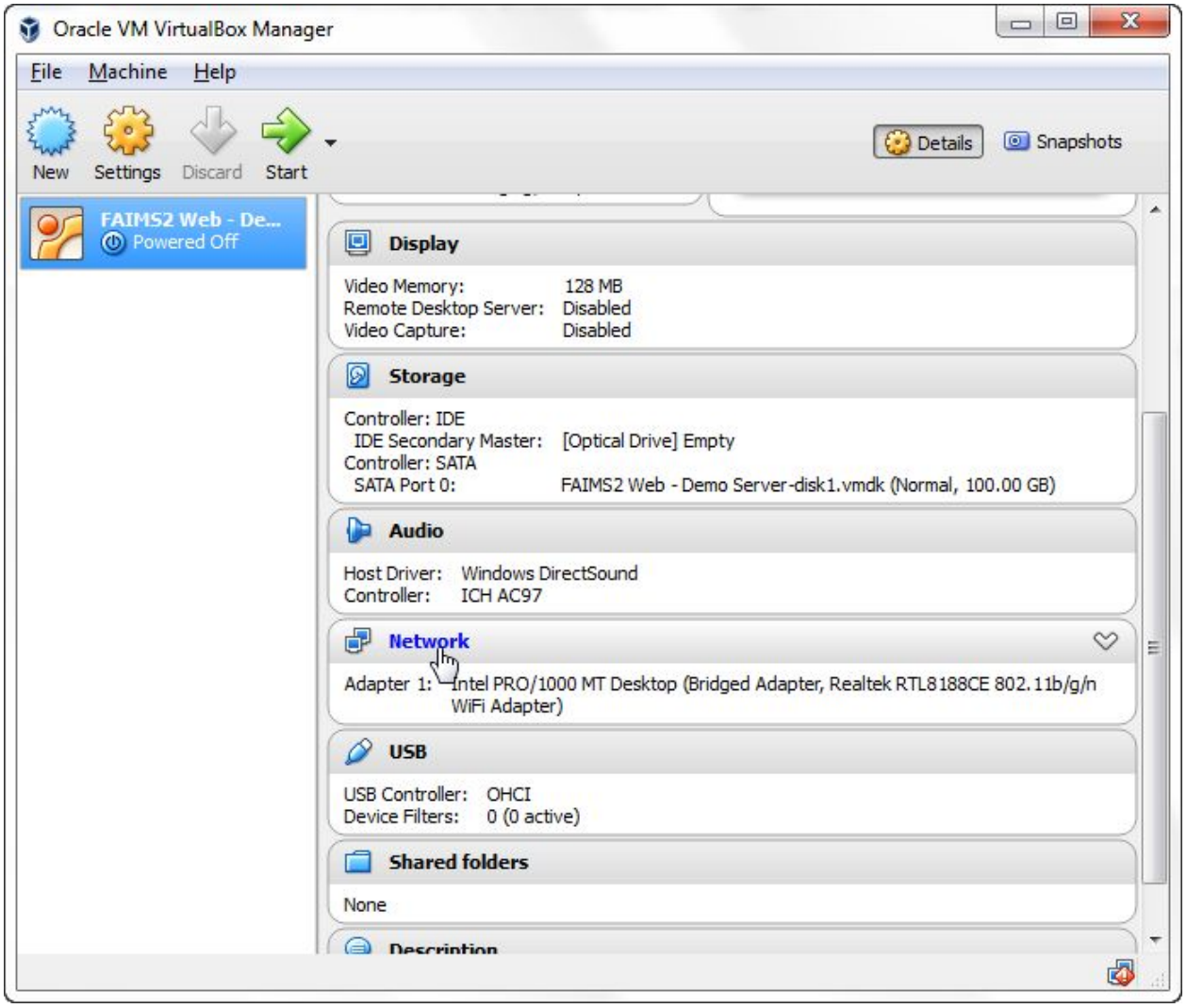


Click on the system and then click the "Start" button in VirtualBox to start the server. You *will* encounter the following error.

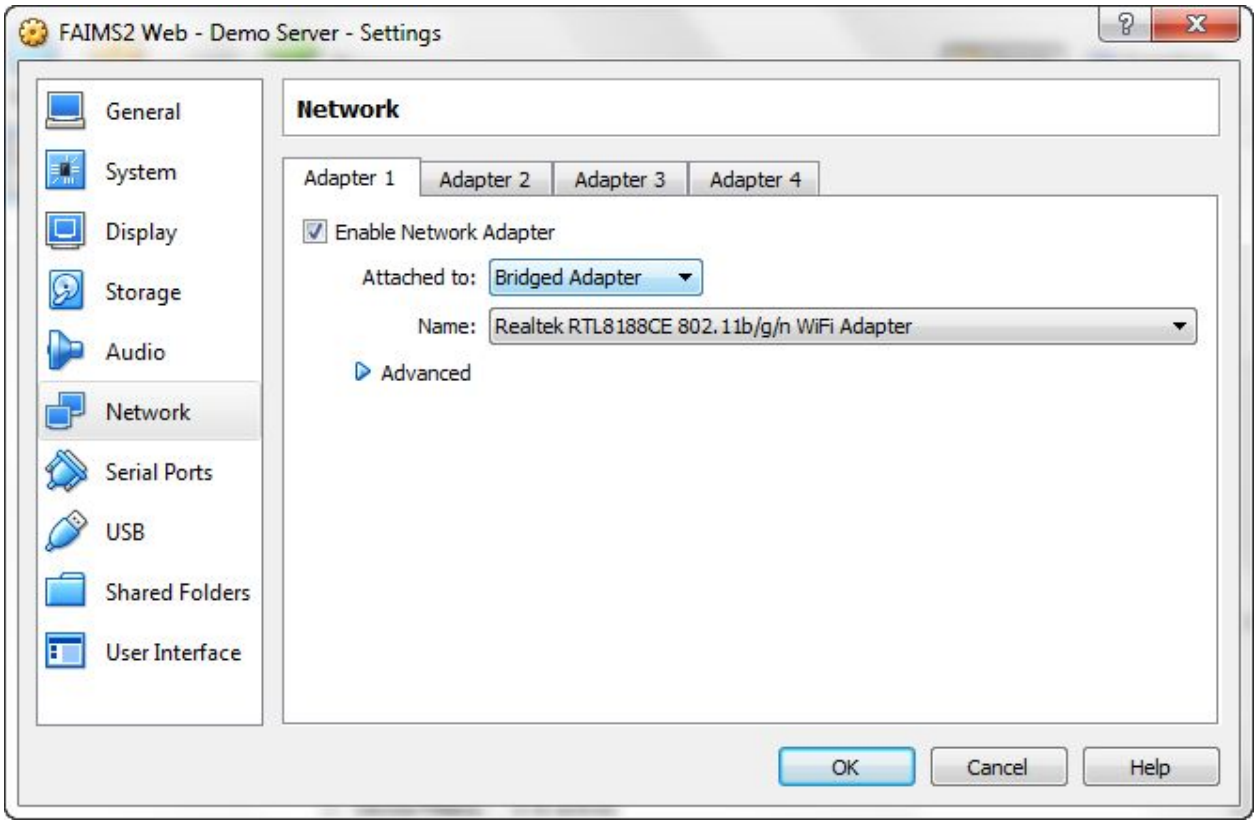


DON'T PANIC. Simply, click the "Close VM" button. You're still on the right track; sometimes, things just have to be a little fussy.

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

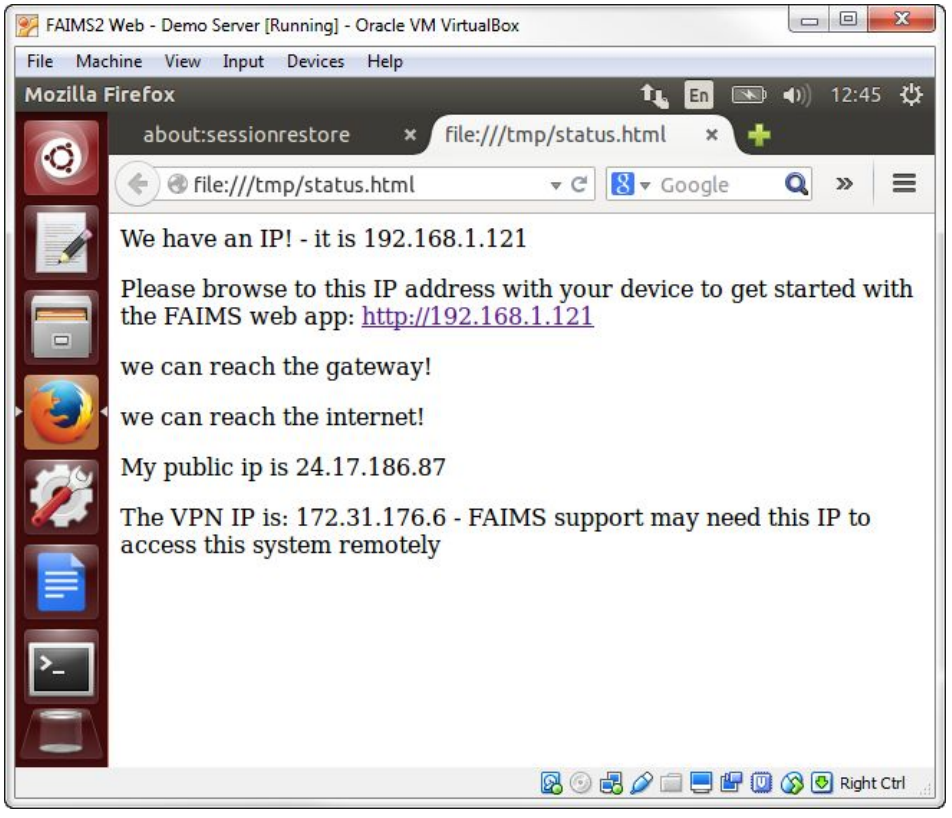


In the popup window that appears, check the "Enable Network Adapter" box on the "Adapter 1" tab, and select "Bridged Adapter" from the "Attached to:" menu.

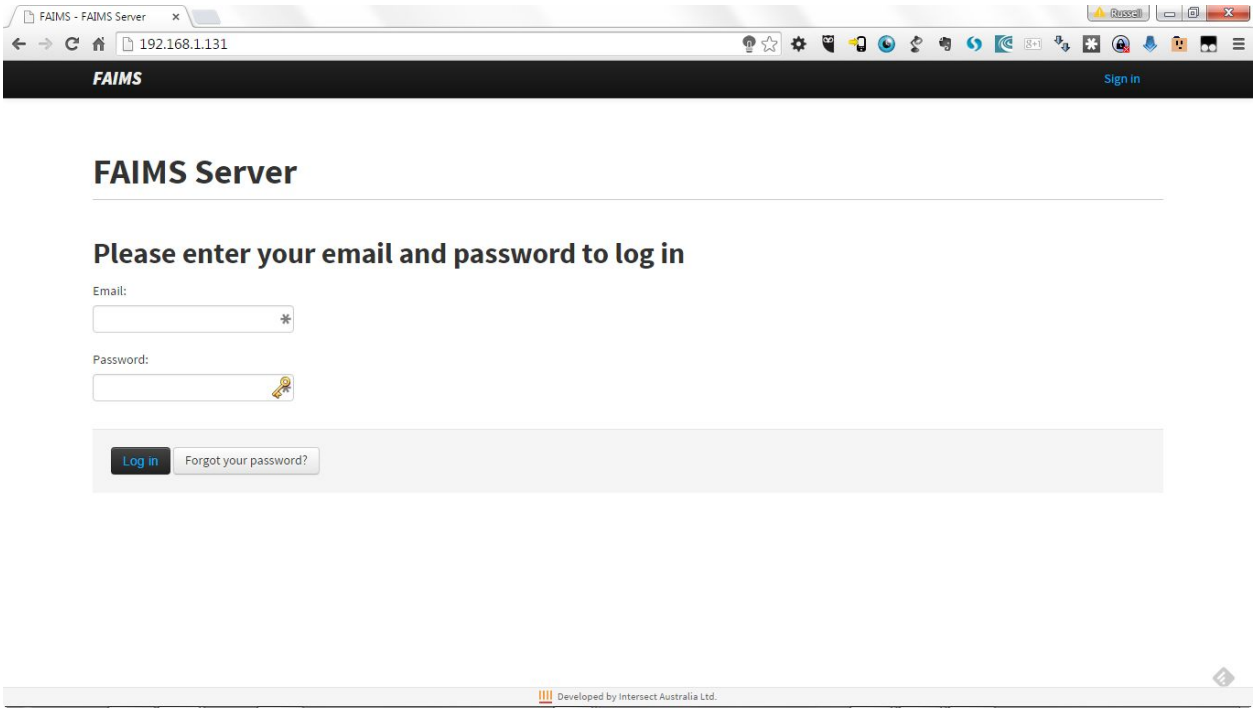


You may choose your machine's wifi adapter but, if you run into any issues, try using your hardline connection instead.

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.



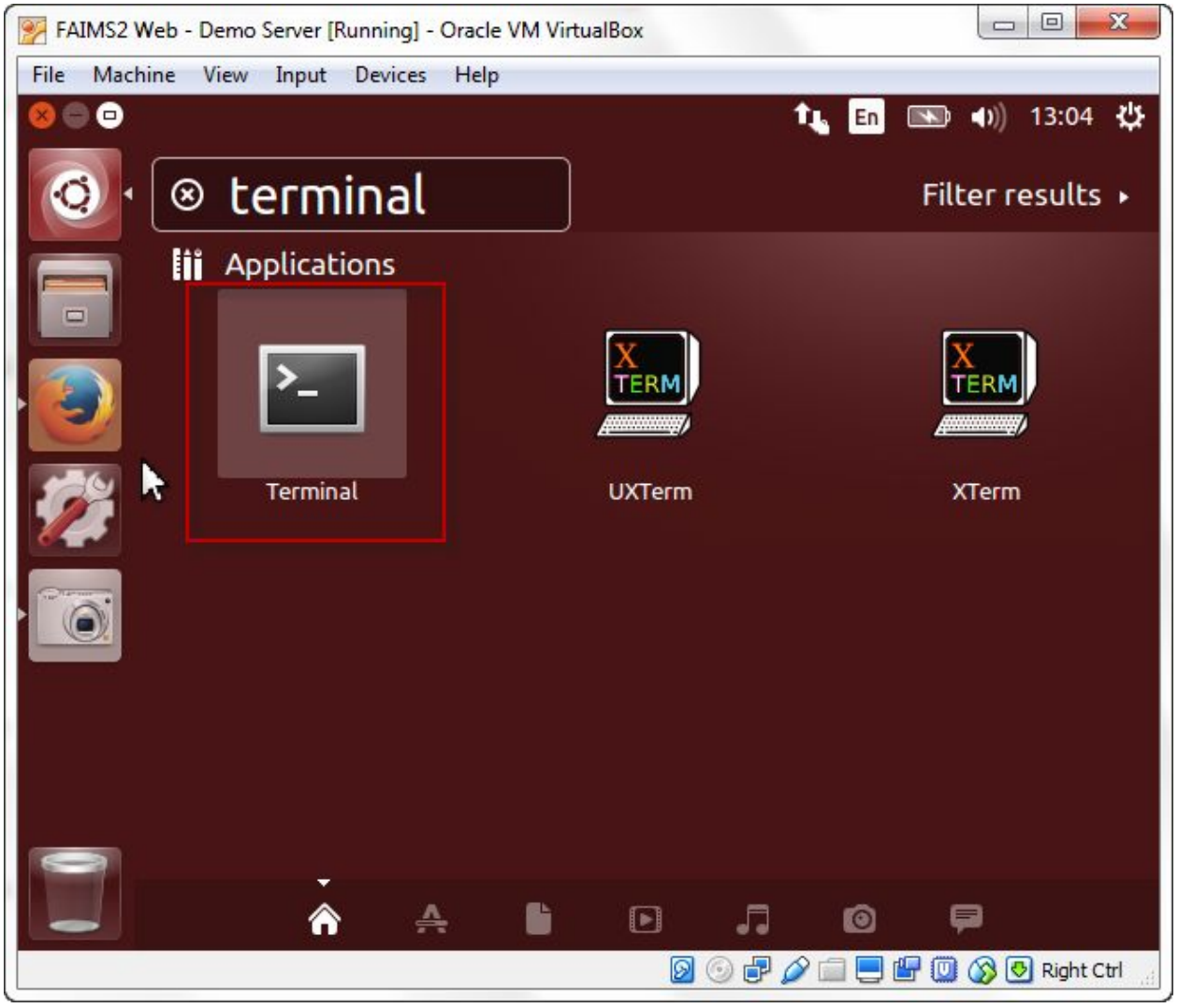
If you like, you can skip ahead a little and, back on your real machine, you can open the FAIMS server by navigating to 192.168.1.121, or whatever address is listed by your FAIMS status page, in your web browser (check your messages for the specific IP address on your machine). **Note:** The default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



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

Getting the FAIMS Server Up to Date

The final step for setting up the FAIMS server is to ensure it is up-to-date. To do this, inside Ubuntu, click on the Ubuntu symbol and type in "Terminal." Terminal (or "the terminal") is a command line prompt that allows you to run many different programs using text commands. (To learn more about the Terminal and executing Bash commands, see here: https://help.ubuntu.com/community/UsingTheTerminal#In_Unity).



Now, in the Terminal, type or copy the following commands individually (each line is a command) and press enter to run each of them:

```
wget http://fedarch.org/update.sh
```

```
bash ./update.sh
```

This may also be a good time to take a break.

Test your knowledge:

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?

What can you do if you can't find an option or setting we refer to in our instructions?

If you've installed VirtualBox before, what do you need to do before setting it up on your computer?

Before you import the server image into VirtualBox, what needs to be done to it?

Installing the FAIMS-Tools

In addition to the FAIMS server, we'll also install the FAIMS-Tools. The FAIMS server is what you'll use to publish and manage your module, but it's the scripts and programs that comprise the FAIMS-Tools that will allow you to create your module's necessary files in the first place. It's the FAIMS-Tools that allow someone to create all the necessary files of a module without significant programming experience.

Installing the prerequisites for the FAIMS-tools

FAIMS-Tools are built on top of and depend on a few other programs. Before we start using the FAIMS-Tools, we'll need to make sure all of those programs are correctly installed and configured.

All of these tools are easier to install on a Linux system than on other operating systems, so we suggest downloading and installing them on the emulated Ubuntu installation you've just set up.

The programs the FAIMS-Tools depend on are the XSLT processors *XSLT Proc* and *Saxonb-XSLT*. Some versions of Linux already have these programs installed. The Ubuntu image we instructed you to download, however, does not. Fortunately, Linux has a handy command line program, apt-get, that provides access to thousands of free software packages and makes installing programs like these straightforward and easy.

Install XSLT Proc

To install XSLT Proc, type or copy the following command and press enter to run it:

```
Sudo apt-get update && sudo apt-get install xsltproc
```

You have now installed XSLT Proc.

Installing Java on Ubuntu

Before installing the next tool, saxonb-xslt, you'll need to install Java. On your Ubuntu VM, you can install Java by running in the following command.

```
sudo apt-get install gcj-4.8-jre-headless
```

It'll probably take a few minutes to download and install.

Install saxonb-xslt

Run the following command:

```
sudo apt-get install libsaxonb-java
```

Installing the FAIMS Tools

To install the FAIMS-Tools, especially in Linux, use the version control software Git. In the Terminal, on Linux and OSX, type "*git --version*" to see if Git is already installed. Type the following command into your terminal window on your Ubuntu installation:

```
git clone https://github.com/FAIMS/FAIMS-Tools.git
```

Git will copy all the files and notes in the GitHub repository (also known as a "repo") to the directory from which you ran the Terminal command.

Installing a Text Editor for Editing Necessary Files

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 good simple text editor 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 would.*

If you decide to write the files outside your virtual machine, we don't recommend using even the simple text editors 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 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, simply download their binary installers and follow their installation wizards. On Linux, you can install gEdit, Vim, Emacs, or another text editor using apt-get. If you're on Linux, we'll assume you know how to do that already.

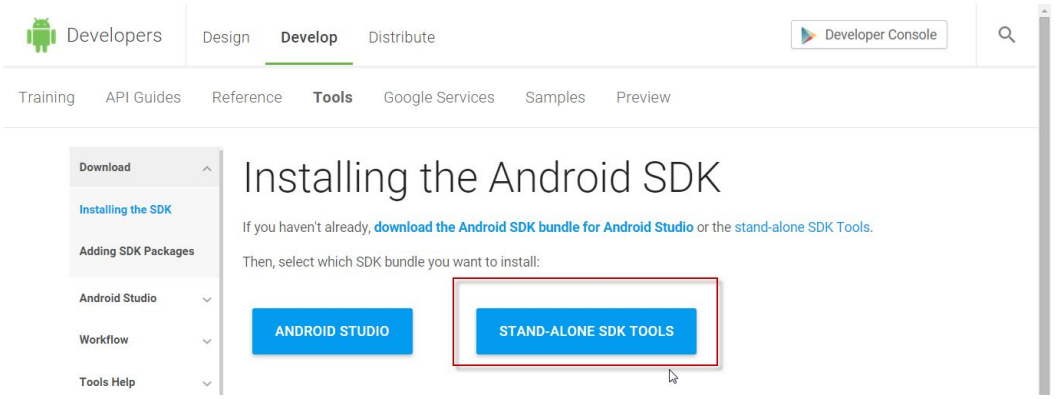
***Note from your friends at FAIMS:** One of the main reasons to use the text editors above is so that you can avoid "Smart Quotes," an automatic feature of many word processor programs that will insert differently styled or extra quotes that will cause the FAIMS server to reject your module. If you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

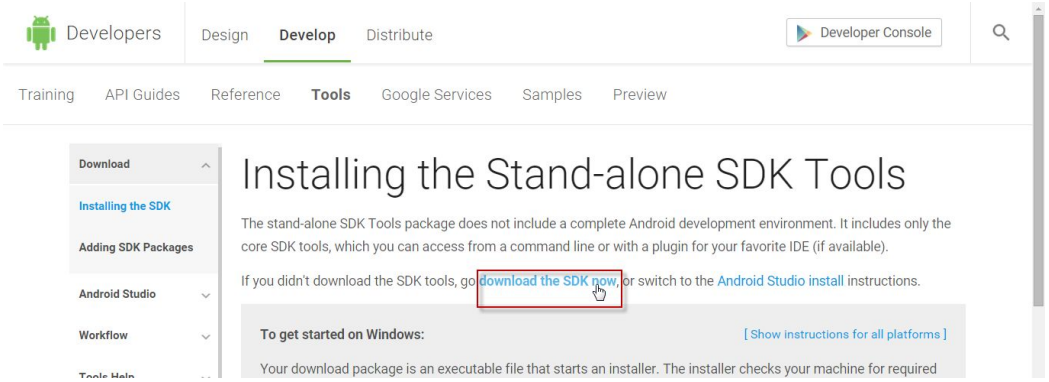
Get Access to your local files via VirtualBox

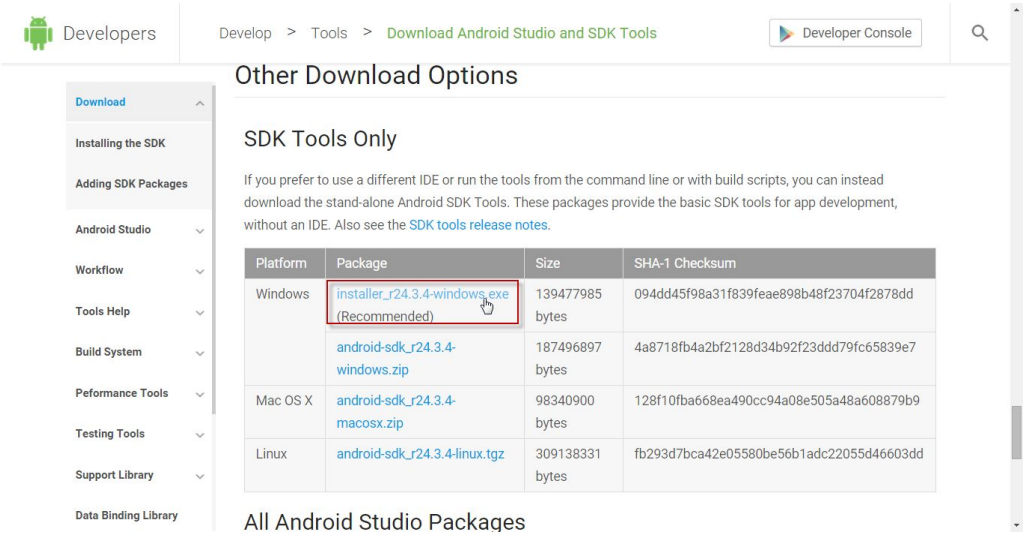
Installing the Android SDK

Next, 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.







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

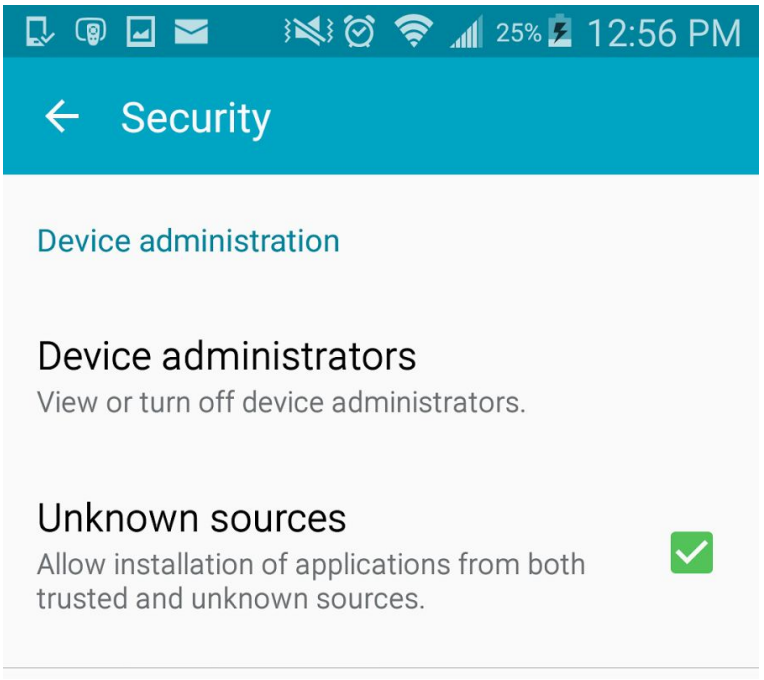
Installing the FAIMS Debug APK on your Android Device

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.

TIP: 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 is a Samsung Galaxy s5. Your settings menu may look different).



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. Below are instructions for enabling Developer Options with a Samsung Galaxy S5; if you have something different, you may need to Google your device and "enable developer options" together.

Note that if you want for USB Debugging to work when your Galaxy S5 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

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.

6. After the seventh tap you will be told that you are now a developer.
7. Go back to Settings menu and the Developer Options menu will now be displayed.

In order to enable the USB Debugging you will simply 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.

Test Your Knowledge

Q1: If the "Error: Network Device Not Found" error appears when you start VirtualBox, how do you fix it? (hint: check the instructions above)

Q2: 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

Exercise: Open up a command window (Windows key + type "cmd" + hit enter/select with mouse. In this command window, enter the text "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.

Coding Modules

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

We're going to start by showing you how to make a basic version of the module. Then, after you've made a fully functioning but simple version, we'll teach you had to add complexity and functionality by modifying the necessary files.

Introduction to Module.xml

Earlier sections introduced the basic necessary files of a module:

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_??)

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.

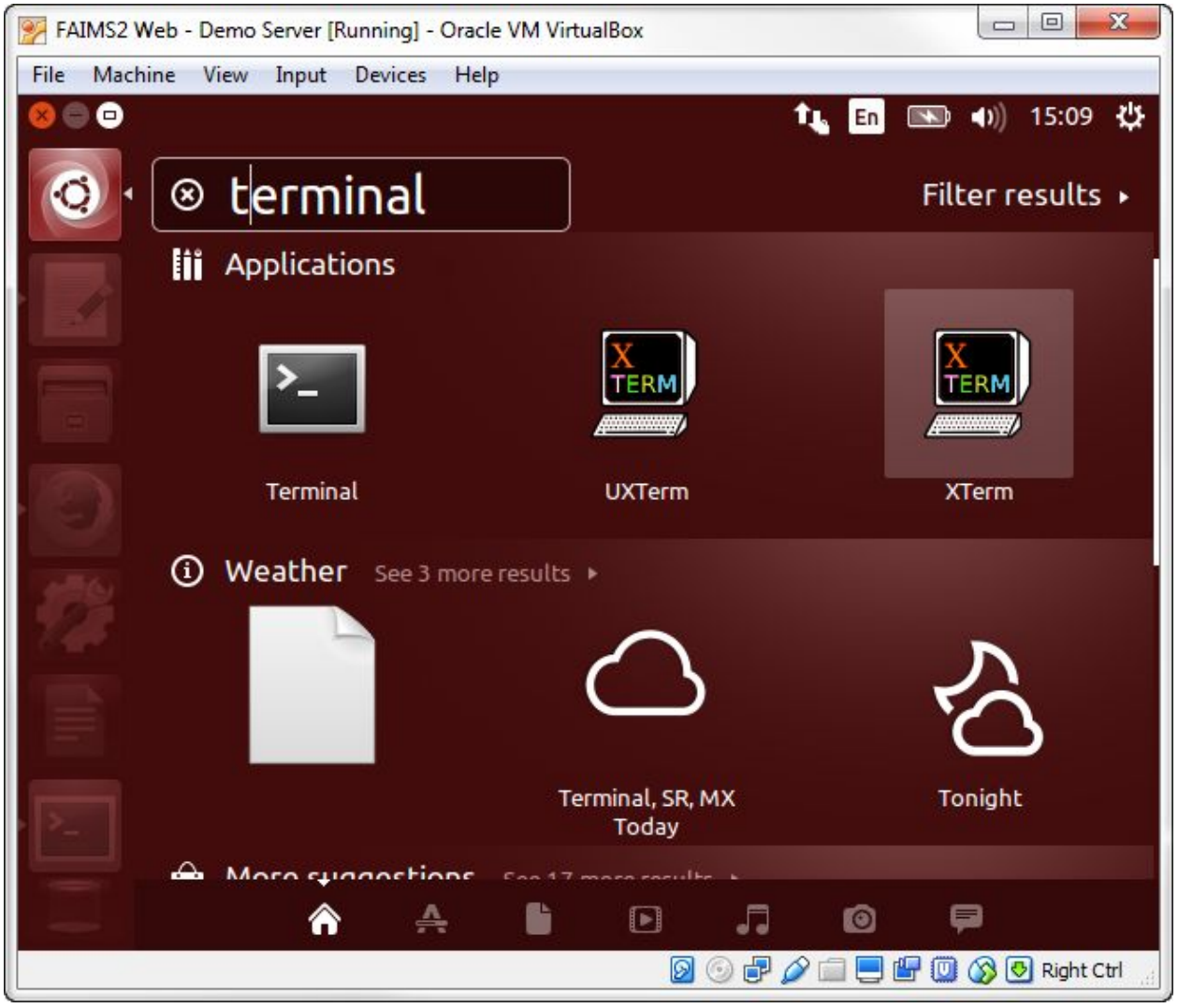
As long as you can correctly design module.xml with a text editor and use it with the FAIMS Tools, you'll be able to make any simple module you'd like. Then, if you want to add more complex functionality to your module, you can go into the freshly-created necessary files and make whatever tweaks are necessary with your text editor.

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.

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.



Start your chosen text editor and open the “module.xml” file. If you’re using the text editor we recommended forthe 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.

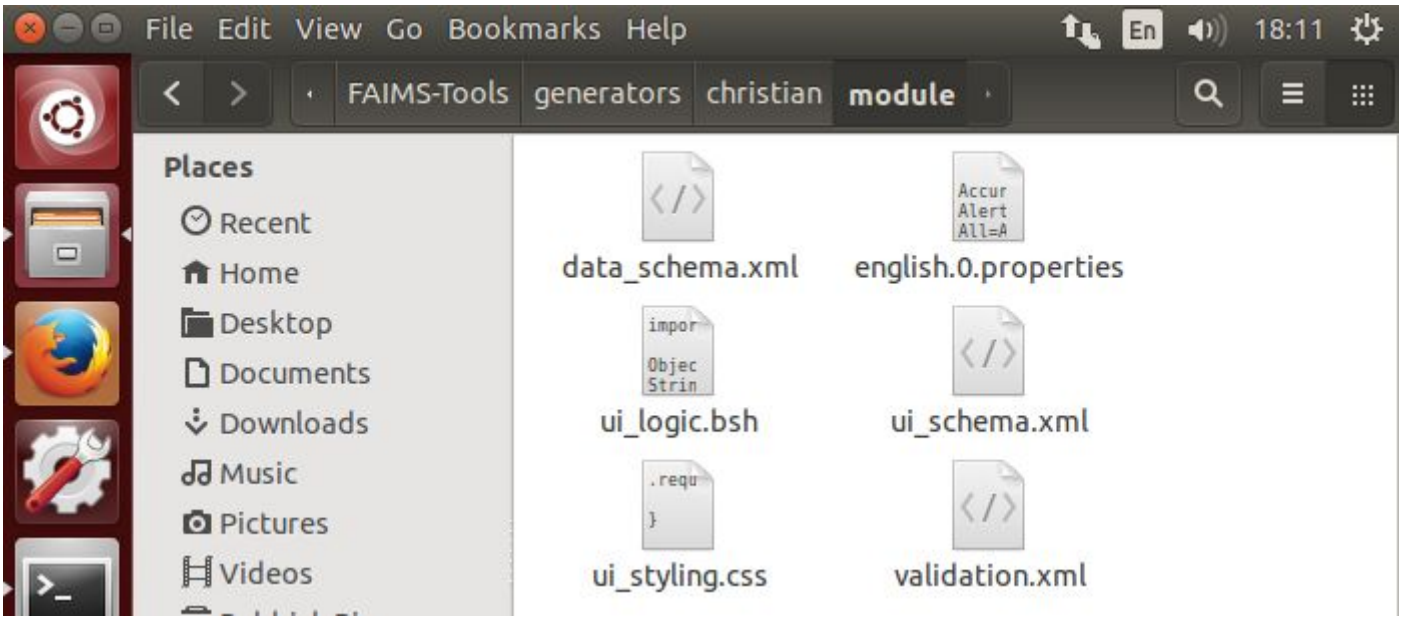
Duplicate the module.xml file and rename it “moduleTemplate.xml.” The version you’re going to feed into the tools is always going to be named “module.xml,” so it’s important that you don’t save over your only template file--you’re going to want to have one lying around the next time you want to make a module. Now you can edit module.xml to do whatever you want.

Once you’ve finished editing and troubleshooting (which we’ll get into in the next section), the instructions are complete and you’re ready to generate your module. From the same directory as module.xml, run the command do so (below) from the command line terminal. If you don't want to type the rest of a thing, hit tab and the terminal will try to guess. The following can be typed as cd ~/F<tab>g<tab>c<tab>m<tab>):

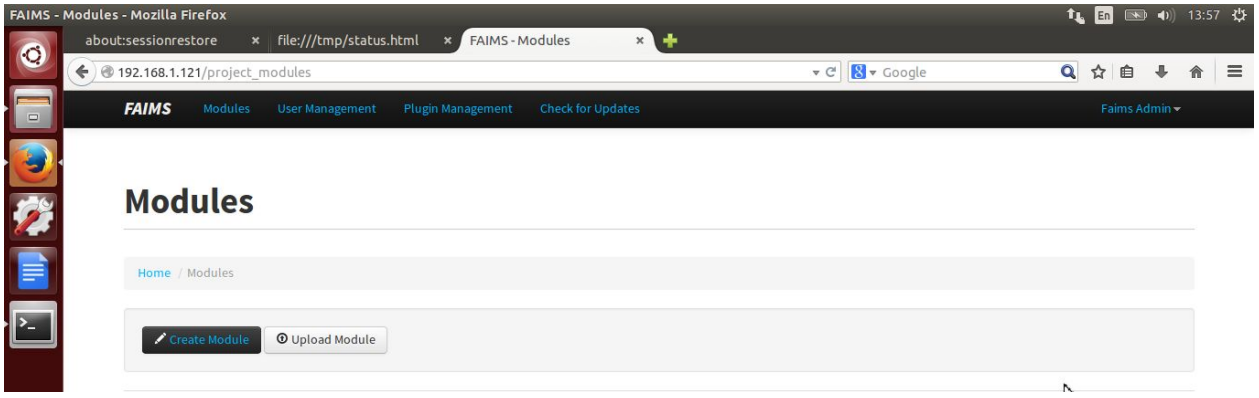
```
cd ~/FAIMS-Tools/generators/christian
./generate.sh
```

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?



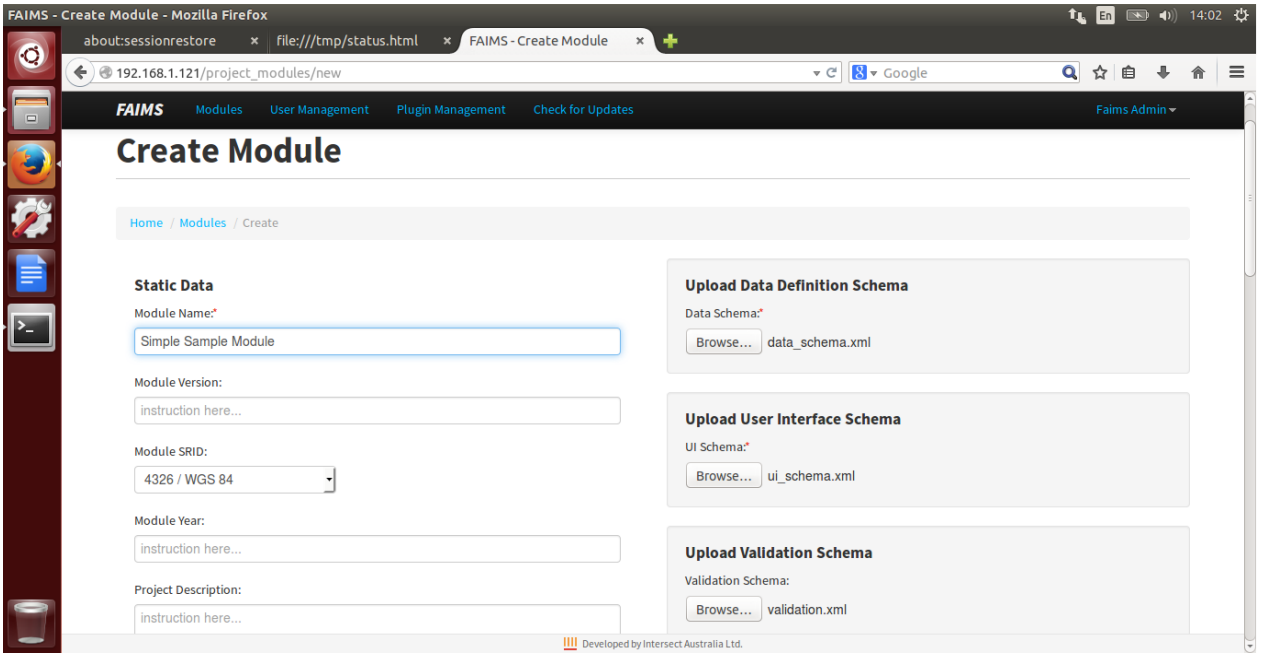
In the web browser, on your Ubuntu installation, open up the FAIMS server and select the "Modules" tab. On the “Modules” tab, click on the “Create Module” button.



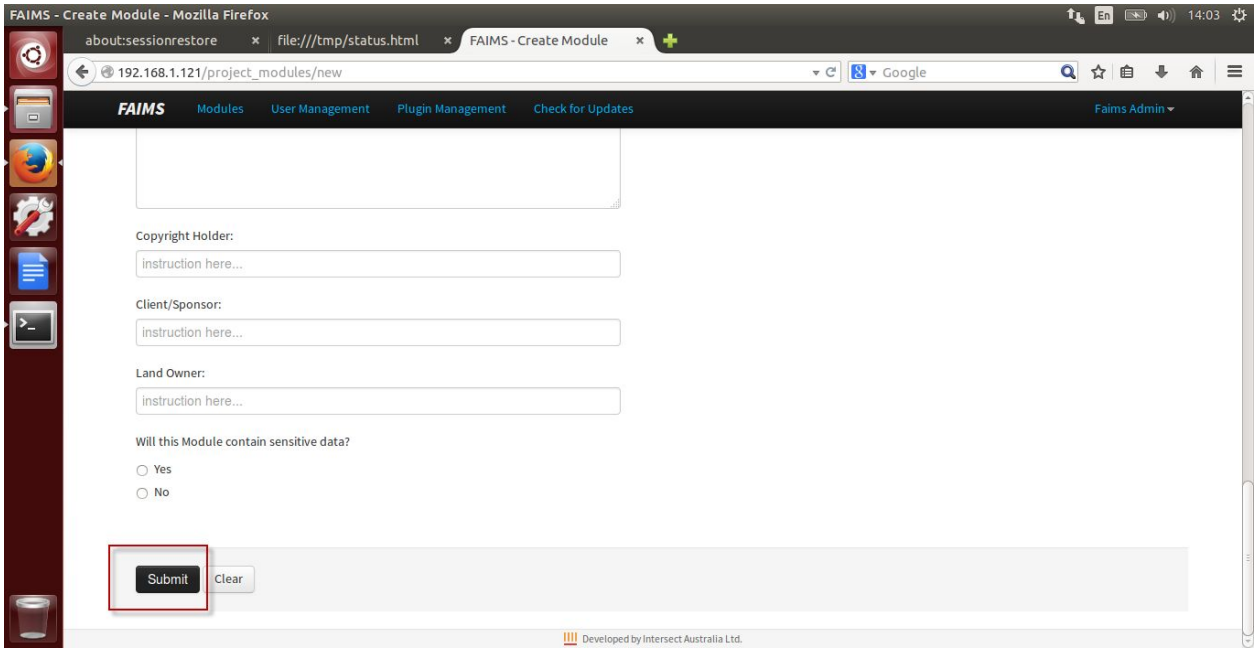
In a web browser, load up the web interface for the FAIMS server and log in.

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:

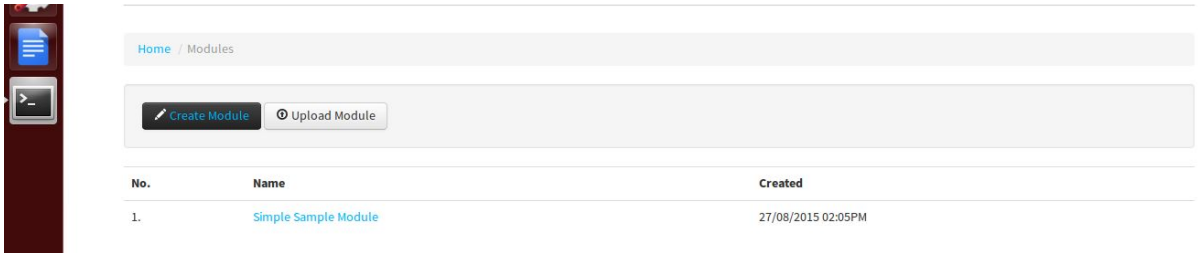
1. Data Schema: data_schema.xml
2. UI Schema: ui_schema.xml
3. Validation Schema: validation.xml
4. UI Logic: ui_logic.bsh
5. Arch16n (optional, since it’s only useful if you designed your module to support interchangeable terms, but it can be ugly if you leave it out): arch16n.properties or english.0.properties
6. CSS (optional, but recommended): ui_styling.css



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



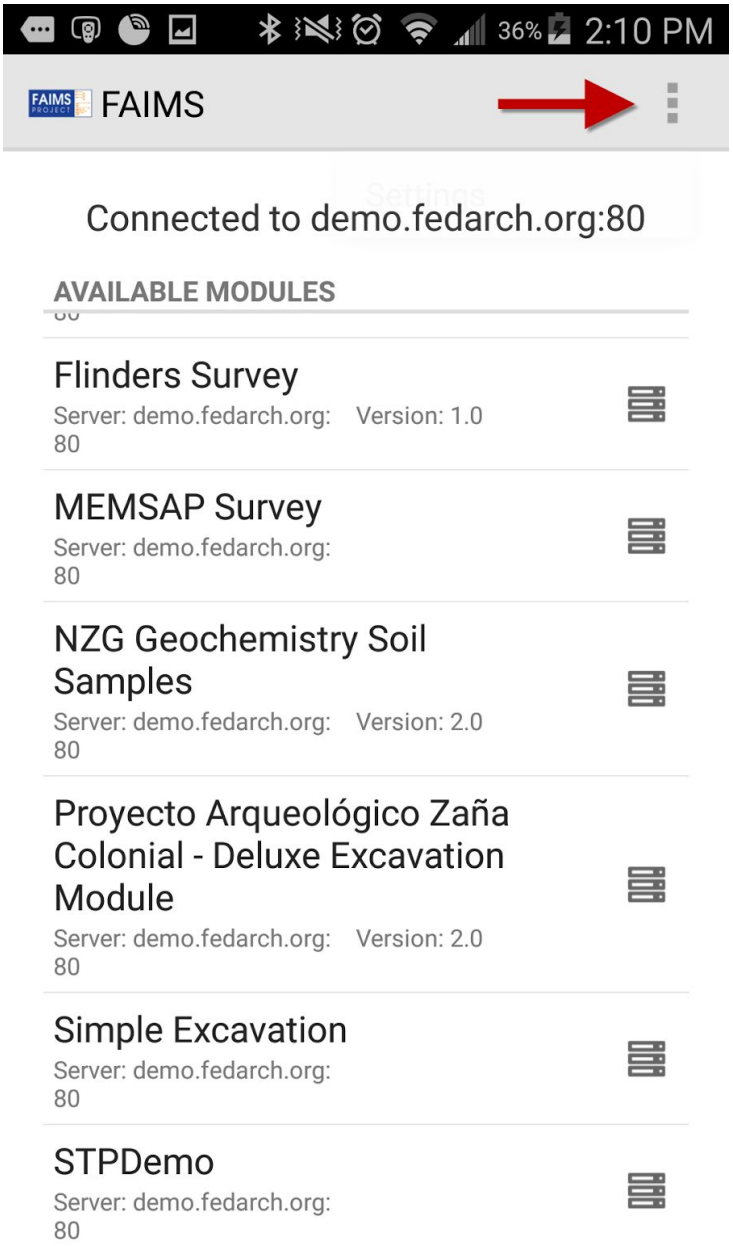
Once the server creates the module, you'll be directed back to the main "Module" tab.

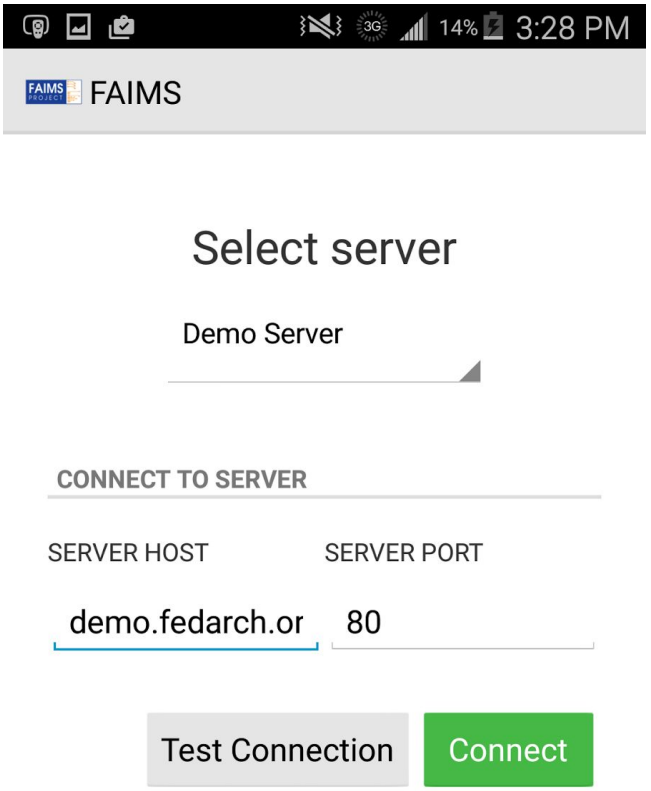


If you click on the module name, you'll see a screen where you can manage the module, including editing the module metadata, schemas, add user accounts, and downloading or exporting your module. You may also browse any records uploaded from mobile devices using the "Module Details" area and "Search Entity Records" button. Near the bottom of

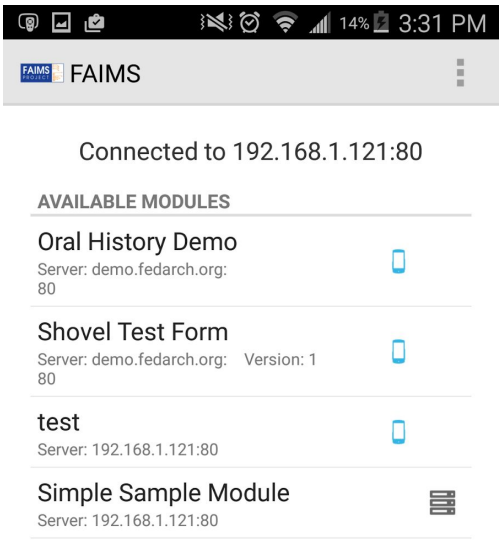
the page, you can delete the module. For more information on deploying or deleting modules from the server, see FAIMS Handout 103 here: https://www.fedarch.org/resources/handouts.pdf

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. ***Note: Some Samsung devices may show the menu differently, such as the Samsung S III. A Nexus 7 tablet shows it in the lower left 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.

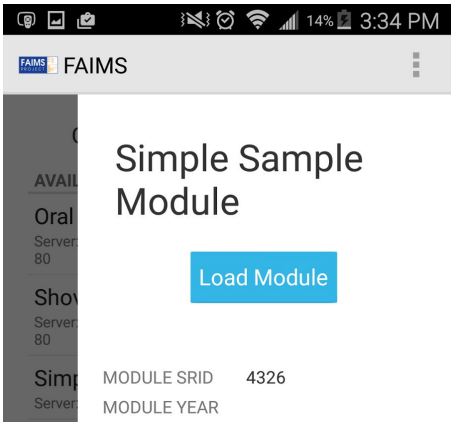




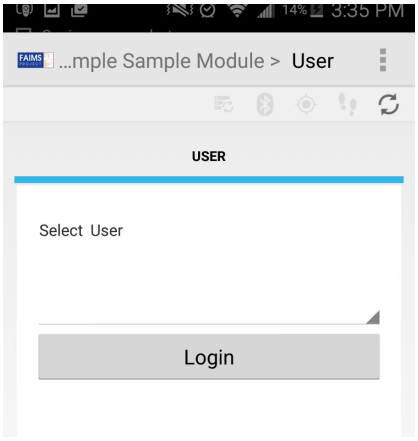
After a few seconds to a minute, the FAIMS app will find your server. Once connected, you will see a list of modules available to use, either locally on your device (designated by a blue phone icon) and those available for download from the server (designated by a black server icon). Once you have downloaded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

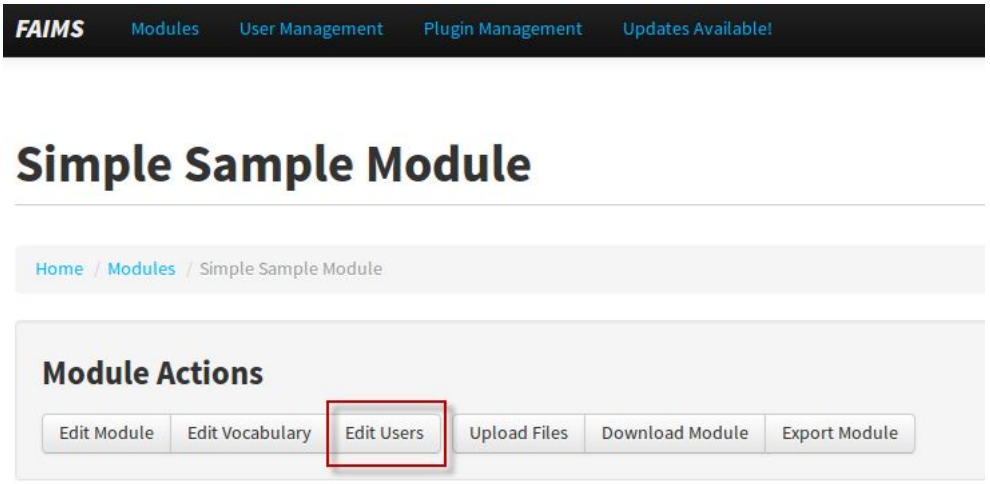


Tap on the item “Simple Sample Module” from the list, which is the one we just uploaded to the server, and the FAIMS app will copy it to your device. A sidebar with the module metadata will appear. You can browse that quickly, and then tap the “Load Module” button.



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.





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.

Let's learn how to code a module.

Understanding The Structure of XML Documents

This section will gradually introduce you to what XML code looks like, how it works, and how you can make it do what you want it to. If you're not used to code, relax and take this section slowly.

From the file module.xml, consider the following section of code.

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

Notice how everything that's written is put inside angle brackets like these <>? The brackets are like a kind of punctuation that lets the computer separate and organize instructions. The bracketed text is called a *tag*, and information contained by tags are called *elements**.

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

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 the code snippet is indented that the very last </User> closing tag corresponds to the very first tag, `<User f="nodata">`. But when writing your own code, it's important to understand: how does the computer decide which closing tag belongs to which 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*. Something can't be "contained" unless it has a beginning and an end. For that reason, if a closing tag could match with multiple opening tags in use, the computer always assume it closes the most recent one.

You may have noticed that the above code snippet 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"/>
```

These are known as *empty-element tags*. Sometimes, an element is complete with only a single instruction. In these cases, it isn't really necessary to have both an opening and a closing tag, as there's nothing to put in between them. The above tags are really just shorthand for the following:

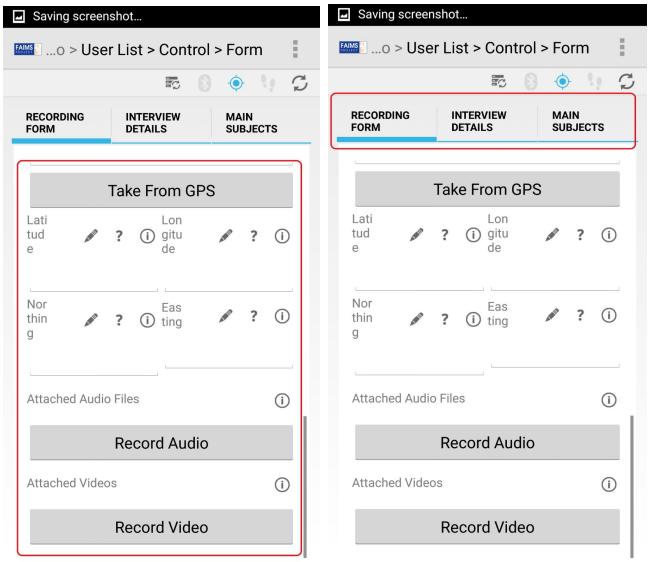
```
<Select_User t="dropdown" f="user">
</Select_User>
<Login t="button" l="Control"/>
</Login>
```

Empty-element tags are given their name as they denote elements which do not contain any other elements.

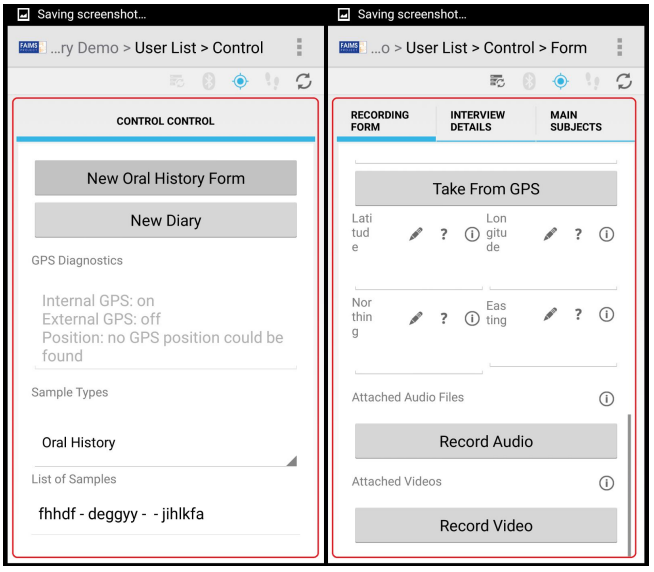
Understanding The *FAIMS* XML Format

Like an XML file, FAIMS modules are hierarchically structured. All **GUI elements** such as buttons, text fields and dropdown menus appear inside of **tabs**. Consider for a moment the module in the image shown below, left. (This is not the simple module we uploaded a moment ago.) The module in the image illustrates the use of **GUI elements** which 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 other image (below, right) 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.



Indeed, tabs are further arranged in collections called **tab groups**. The below image shows two different tab groups—"Control" and "Form"—each containing their own sets of tabs.



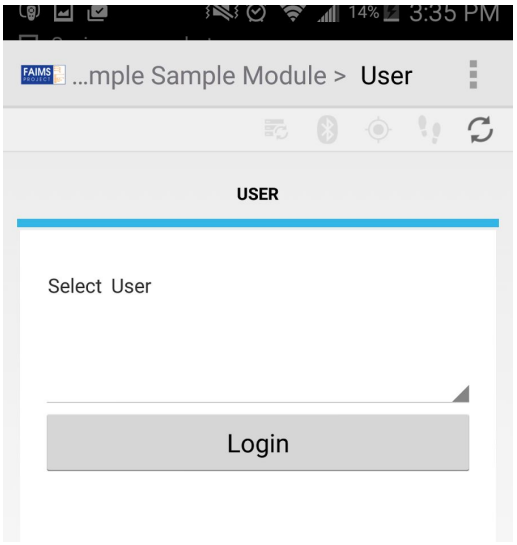
The structure of XML in the FAIMS-Tools reflects the fact that tab groups contain tabs, which contain GUI elements. In the code snippet below, notice that the <Tab_1> and <Tab_2> elements appear *within* the <Tab_Group> element. In fact, the XML-Tools determines that <Tab_Group> is a tab group, not by its name, but where it appears in the hierarchy of XML elements; elements which appear directly within the set of <module> tags are interpreted as tab groups[1]. Similarly, elements which appear directly within a tab group are deemed to be tabs. (In the below code snippet, these are <Tab_1> and <Tab_2>.) Furthermore, XML elements within tabs are interpreted as GUI elements.

```
<?xml version="1.0"?>

<module>

  <Tab_Group>

    <Tab_1>

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

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

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

```
    </Tab_1>
    <Tab_2>
      <Button t="button" />
    </Tab_2>
  </Tab_Group>
</module>
```

Now that we understand the structure of the XML which the FAIMS-Tools uses, let's explore the simple module we uploaded a moment ago to gain a more concrete understanding of how the module.xml file was translated into a FAIMS module.



To create this first login screen, we used the following XML code:

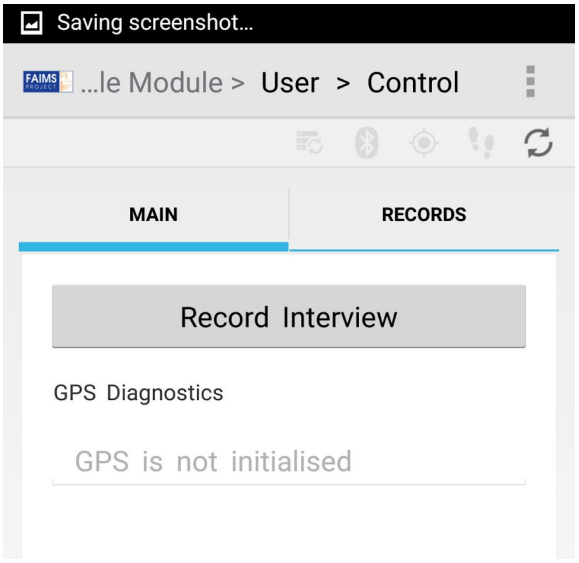
```
<User f="nodata">
  <User>
    <Select_User t="dropdown" f="user"/>
    <Login t="button" l="Control"/>
  </User>
</User>
```

The outermost set of <User> tags creates a **tab group** because they appear directly within a set of <module> tags. The *tag name* you write is displayed in the breadcrumb navigation bar at the top of the screen. The name given is arbitrary, however, it must start with an uppercase letter.

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 generating code associated with the data schema. The practical effect of this is that inputs provided in the "User" tab group, including the "Select User" dropdown menu, are not saved to the database—the "User" tab group is not considered a saveable record and only allows people to log in.

The pair of <User> tags, which are nested within the previously described set, are used to define the "User" **tab**. The tag name you write is displayed in the list of tabs. As described above for tab groups, tag names of tabs must be capitalised.

The <Select_User> and <Login> elements represent GUI elements. Notice the “t” attribute, which is used to denote the type of GUI element. Also note that setting t="dropdown"[2] creates a dropdown menu, while setting t="button" causes the <Login> element to represent a tappable button. Setting l="Control" on the button 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. Also note that references are case-sensitive, so writing “control” with a lower-case “c” would fail. Finally, writing f="user" causes the FAIMS-Tools to populate the menu with a list of usernames which are defined on the FAIMS server.



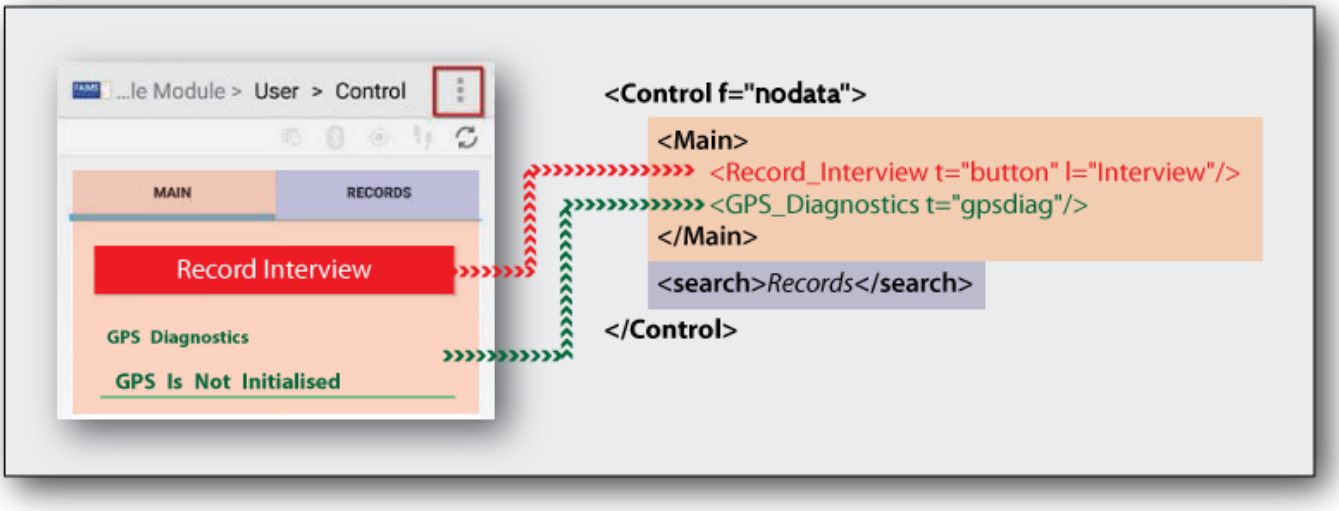
If we choose a user (“FAIMS Admin” is our only option for now) and then tap the “Login” button, we’ll be taken to a new screen or “tab group” in FAIMS, the “Control” tab group. The code to create this tab group is as follows:

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

[2] **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.

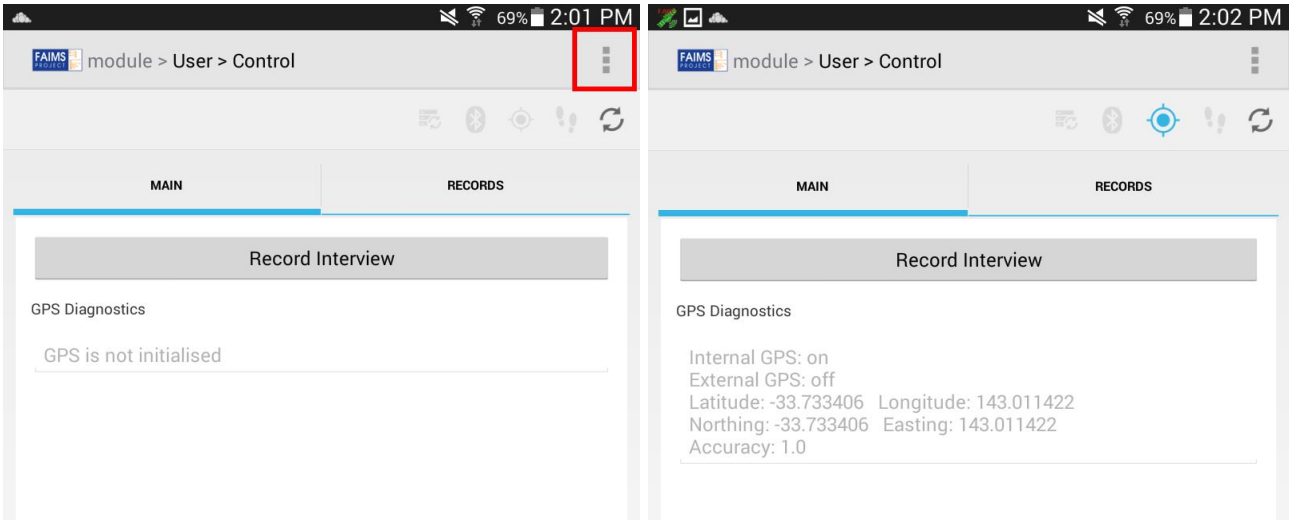
Lists, unlike dropdowns, do not allow null elements so using a t="list" for login, as in the code below, prevents the problem of logging in with null user..

```
<User f="nodata">
  <User f="noscroll">
    <Select_User t="list" f="user" l="Control"/>
  </User>
</User>
```



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

Here we have the main "Control" tab group, which groups together two other tabs, "Main" and "Search." The <Main> element creates the first tab, and contains a button element, "Record Interview," which links to the tab "Interview," and "GPS Diagnostics" which is of the GUI element type "gpsdiag." The generator script knows to take elements having the t="gpsdiag" and replace them with their respective GUI types. In this case, "gpsdiag" is a GUI type that creates text labels and displays information about the Android device's GPS location. Right now, the "gpsdiag" element is telling us that the GPS is not initialised. 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.

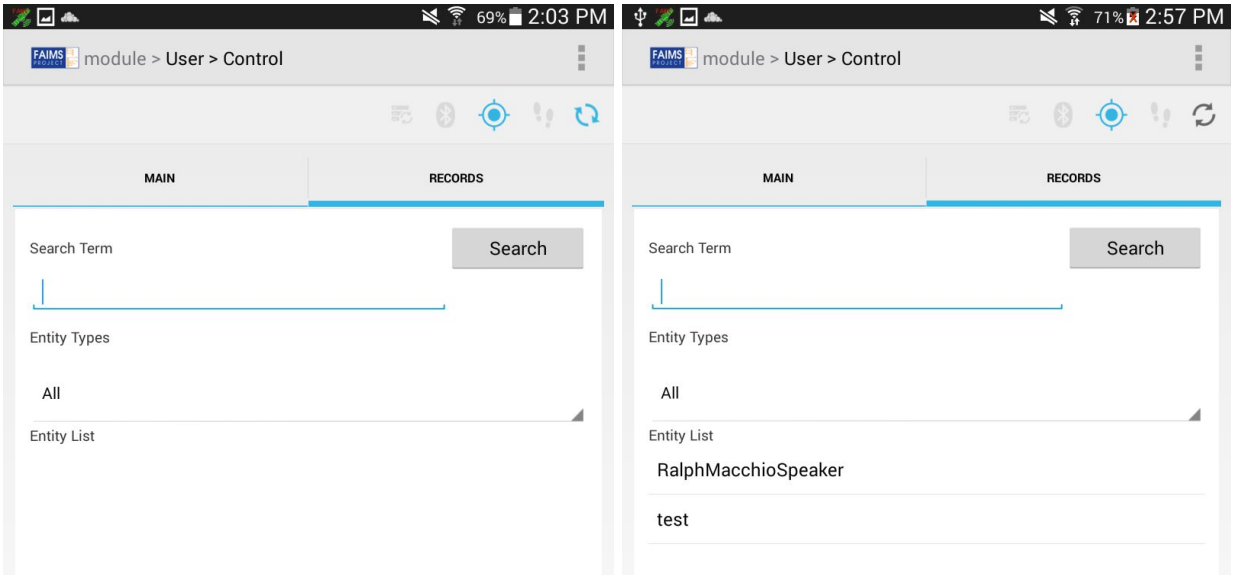


With the GPS antenna turned on, the "gpsdiag" element now displays quite a bit more information, including the status of the GPS antenna, location in both Latitude/Longitude and Easting/Northing, and an accuracy measurement.

As well as a "Main" tab, there's a "Records" tab created by the <search> element. Notice that the tab that appears in the GUI is labelled "Records" instead of the <search> element's tag name (i.e. "search"). This is because when you write text in a tab's corresponding XML element, the FAIMS-Tools takes that text as the label, overriding the default behaviour which is to take the label from the XML element's tag name.

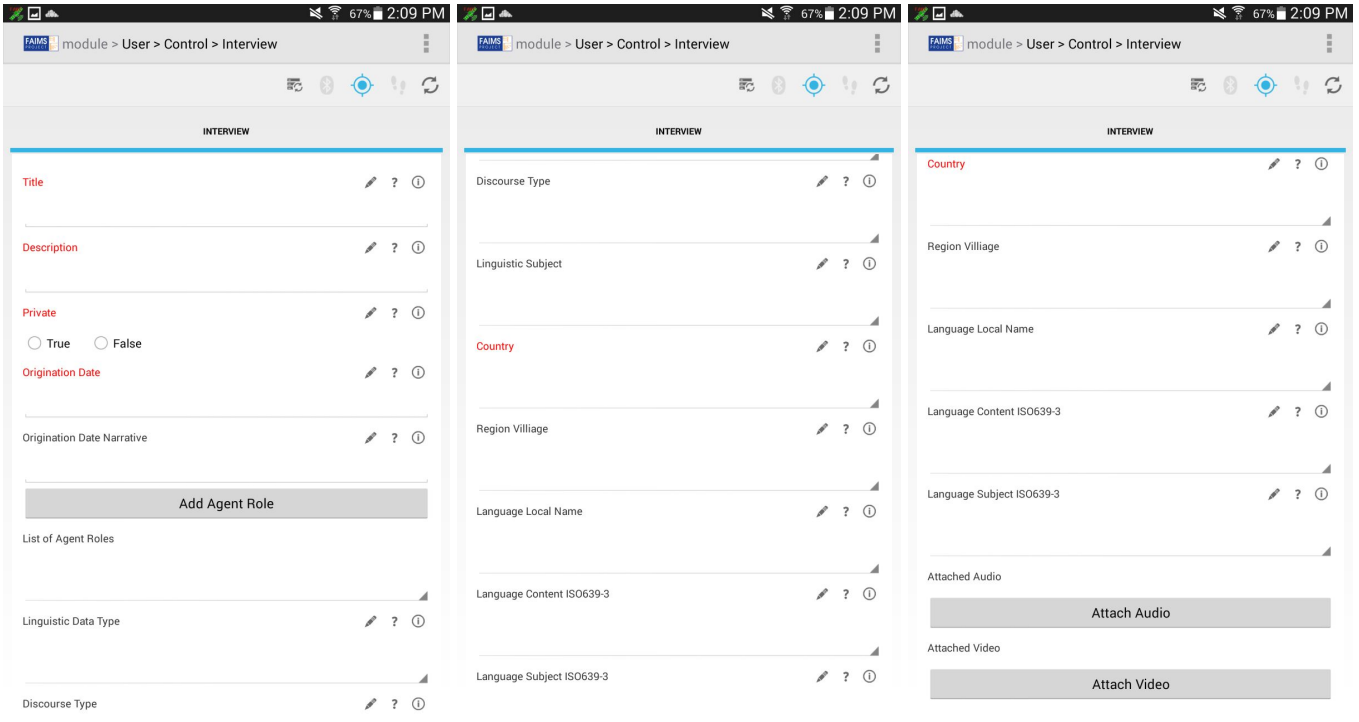
Now, if we tap on the "Records" tab, the screen will update and show several elements that FAIMS knows to include in any "search" element. Right now the search element won't do much but, once we have a few records to work with, we can search each record according to different criteria such as by term or by entity type. Notice that even though we only wrote <search> and didn't specify any GUI elements within it, the FAIMS-Tools has populated the tab group with several functioning GUI elements anyway. This is because the <search> tag has a special meaning to the FAIMS-Tools.

Because it's special, the <search> tag cannot be used as anything except for a tab for searching previously saved records. However, simply changing <search> to <Search> (i.e. capitalising the tag name) causes the FAIMS-Tools to interpret it as an ordinary element and not a search tab.



The Search element before and after adding a few records.

Backing up one screen—at the "Main" tab—if you tap on the "Record Interview" button, you'll be taken to this screen:



The code to create this long screen looks like this (the explanation of individual building blocks of this code follows after the code):

```
<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>
```

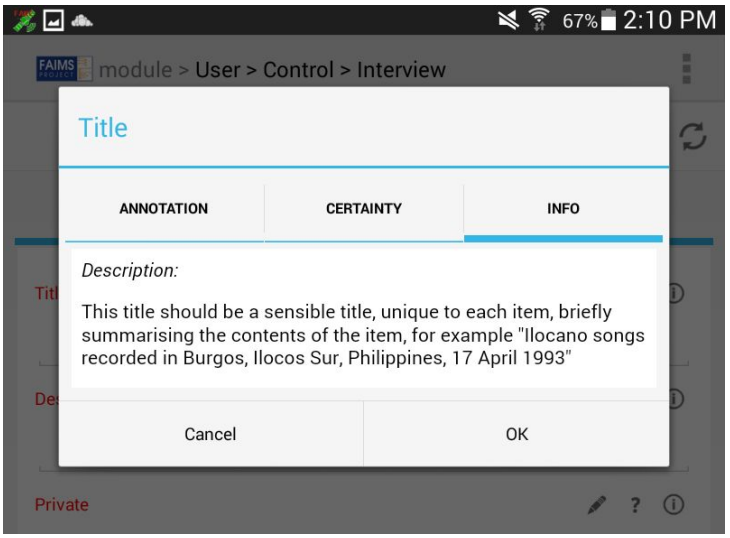
```
      <desc>Subject language is the language that is the subject of your research. Insert the
3-letter ISO 639-3 code for your language, and the standard name of the language as spelt in the
ethnologue entry [search on www.ethnologue.com/site_search.asp]. Separate the code and the
language with a 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>
```

You should already be familiar with the first tab group tag, <Interview>, and the one that creates the screen's single tab, the second <Interview> tag. There are a number of other tags in this code block, however, and they each create a User Interface (UI) element, along with enclosed tags that denote a UI element's options, e.g. dropdown menu options.

The first such element is the <Title> element.

```
 <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>
```

For the <Title> element, there is no UI type specified, so it by default becomes 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 and the record cannot be saved with it empty. The <desc> tag allows you to set a description that your users can access by tapping and holding for a few seconds on the info button:



This next code block designates that the <Private> element is a radio button UI object, it is required, has a description, and includes two options ("True" and "False"), set off with the <opts> and <opt> tags.

```
<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>
```

This next block designates a button element ('t="button"'), <Add_Agent_Role>, that links to the tab group "Agent_Role." This button allows you to register a new agent role.

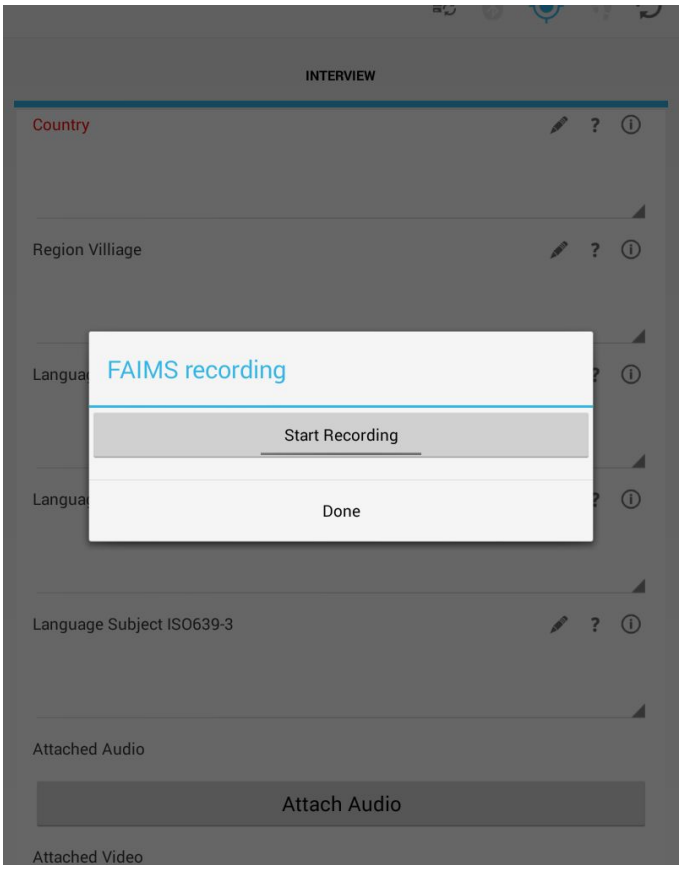
```
    <Add_Agent_Role t="button" lc="Agent_Role"/>
```

The next block 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"/>
```

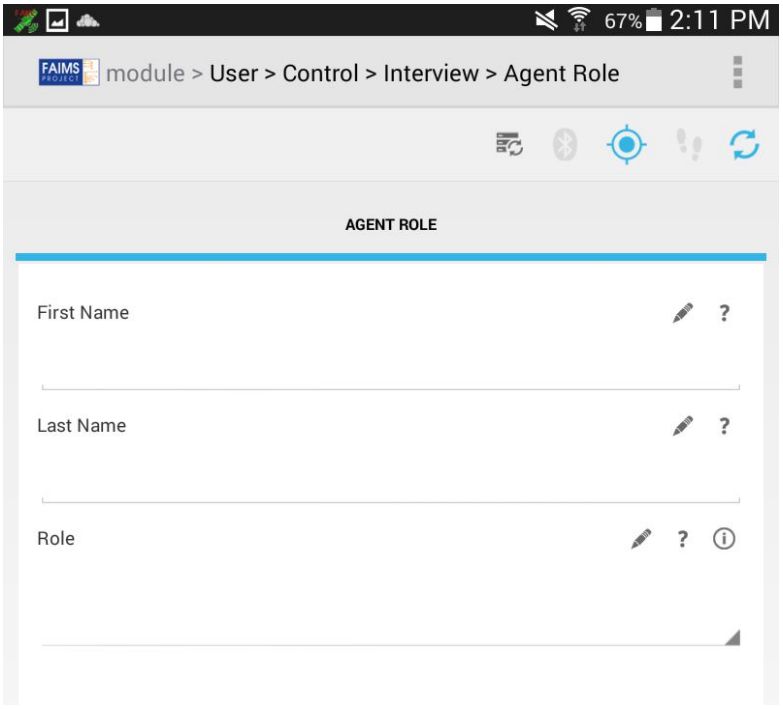
The final two element types used on this module screen are the 't="audio"'' and 't="video"' types. They allow you to record, oddly enough, audio and video files and attach them to your records.

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



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."



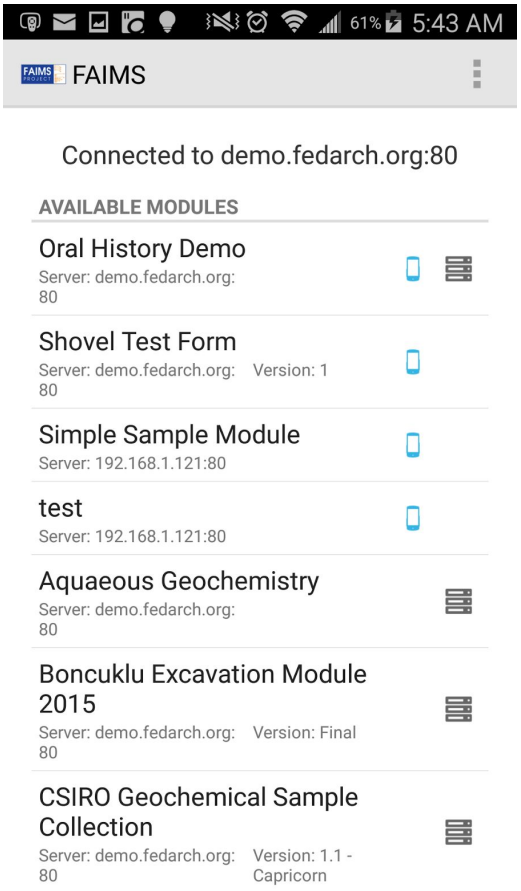

```
<Agent_Role>
  <desc>Enter participant name in the format Lastname, Firstname. Choose the
participant role from the closed vocabulary provided. Use the description field
to provide additional information on role or agents. Enter participant name in
the format Lastname, Firstname. Choose the participant 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>
```

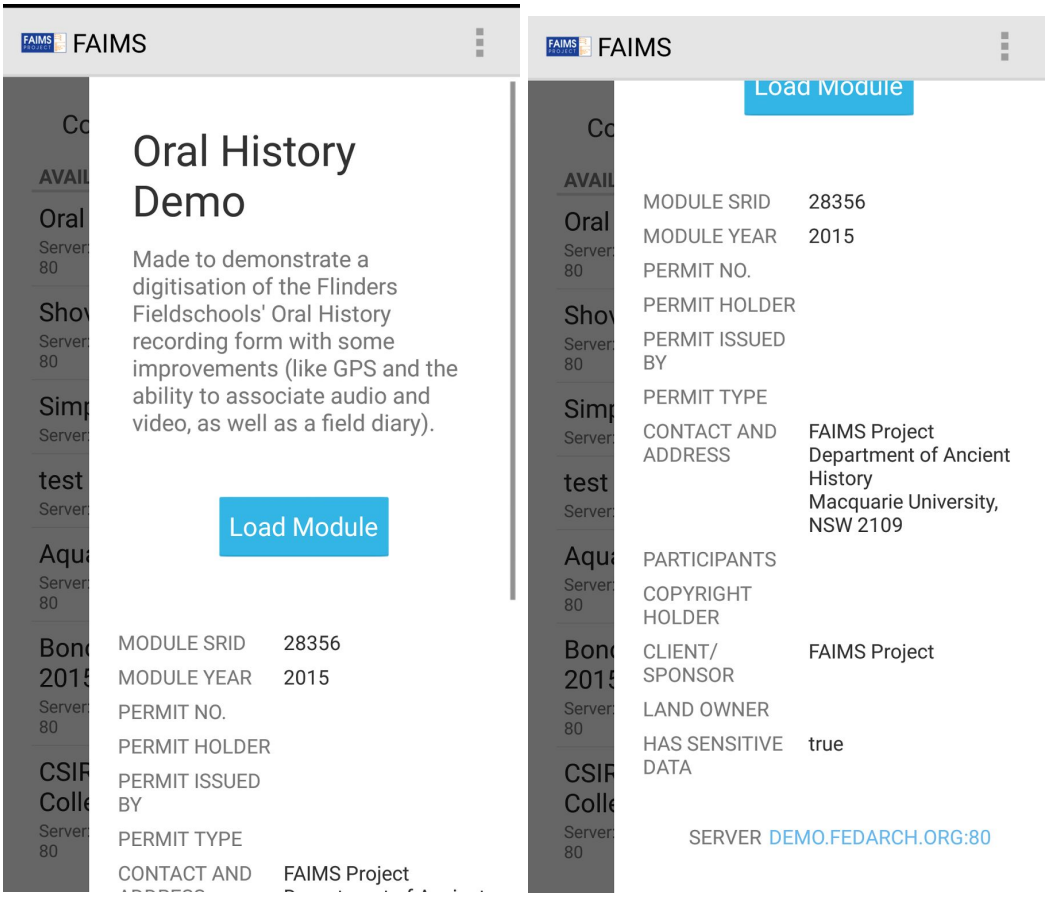
Iterating to Match the Oral History Module

We've got a good start to our module, but now we're going to begin modifying elements to transform our simple module into one that matches the functionality of the Oral History Demo module available from the FAIMS Demo Server.

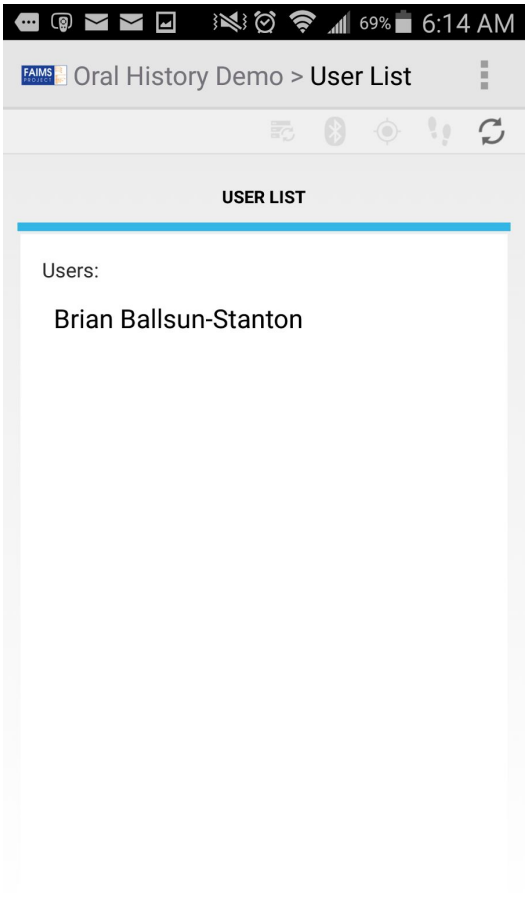
First, open up the FAIMS mobile and download the “Oral History Demo” from the FAIMS demo server (select "Demo Server" from the dropdown menu in the FAIMS setting menu).



The Oral History module has a bit more metadata information included than our module. You can update this information via the "Module" tab on your FAIMS server installation.



The first screen of the Oral History Demo looks similar to our module:



In our module.xml file update the following code:

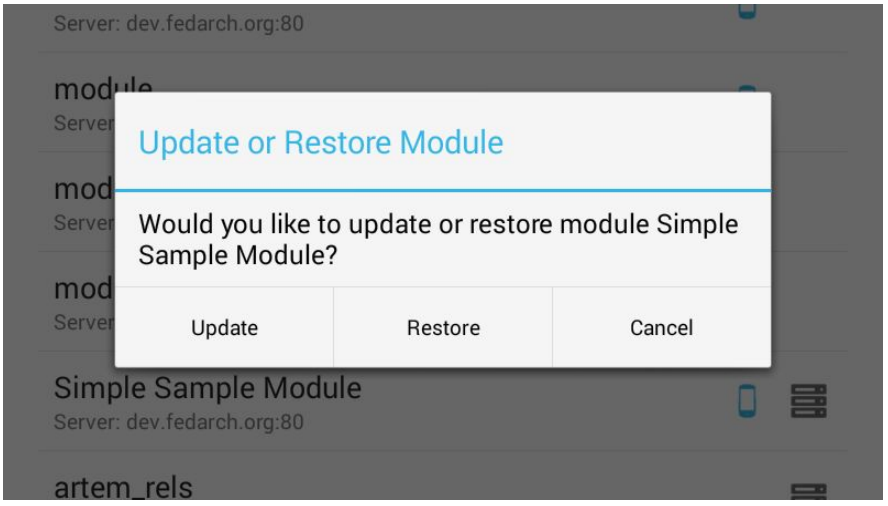
```
<User f="nodata">
```

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

to match this (new code in **bold**):

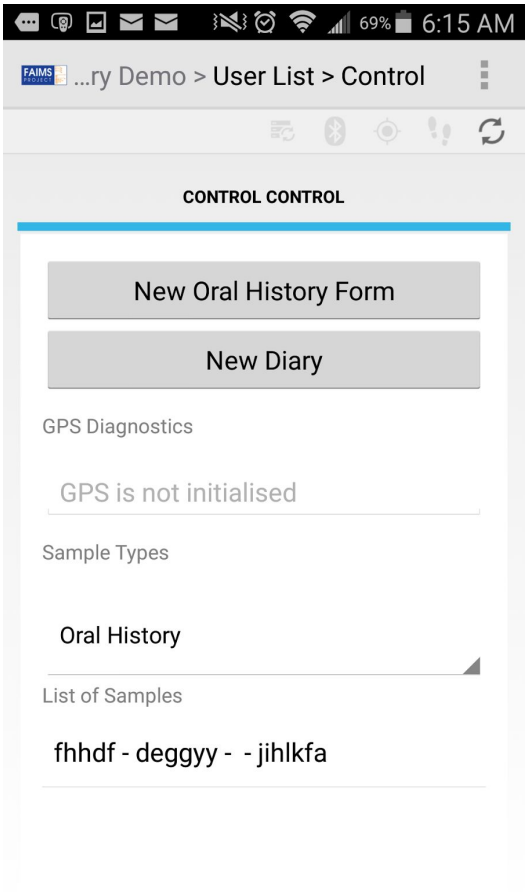
```
  <User f="nodata">
    <User_List>
      <Users t="list" f="user" l="Control">
        Users:
      </Users>
    </User_List>
  </User>
```

This replicates the “User List” tab illustrated in the above screenshot. Save your modified module.xml, then run "*./generate.sh*" in your Ubuntu terminal again, and reupload the module files through your server and download the 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 below dialogue is displayed:



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

Note: While we'll build a user interface that mimics the Oral History module, we'll actually have to create a new module later on the server after the design is complete so that all the functions work. Just updating the module does not allow you to modify the data schema and so our current module won't have all the data fields on the server that are present in the Oral History module. This separation of data from the way data are presented and from the logic used to format or query the data is an example of the [Model-View-Controller (MVC) software pattern](). This exercise will focus on making a "mock" front-end. While what we make with module.xml will look like the end product, none of the attributes will be correctly defined in the data schema. As such, we'll need to revisit this later to make sure all attributes are correctly defined and to plumb in the last bits of logic which allow for a fully customised and operational ~~battlestation~~module.



For this next screen (shown in the screenshot above), we need to condense a few elements from our original module. Change the following code:

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

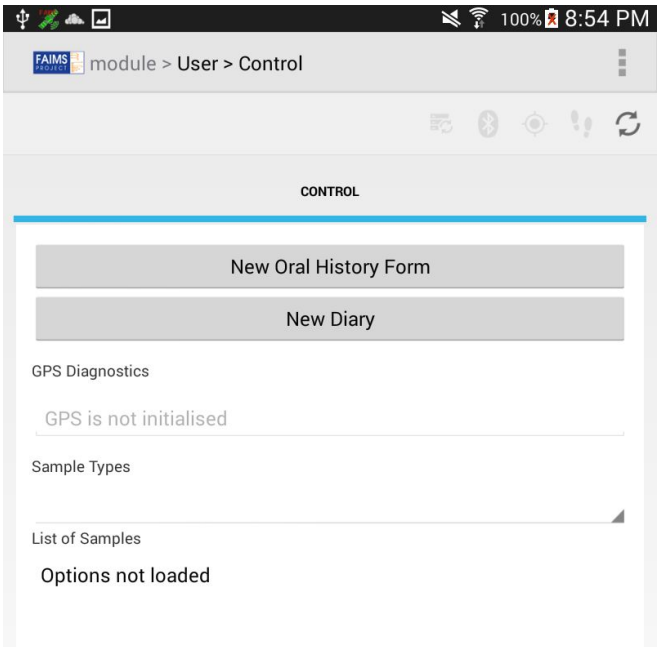
to match the following:

```
<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>
```

This code 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.)

Next, on your Ubuntu installation, save your module.xml file and rerun the generate.sh script. This will translate your changes into the separate FAIMS module files. On the FAIMS server, click on the module name and click the "Edit Module" button.

Here you can upload new versions of all the FAIMS module files, with the notable exception of the data_schema.xml file. Go ahead and upload your updated files. You should see something similar to the image below after selecting a user in the uploaded module.

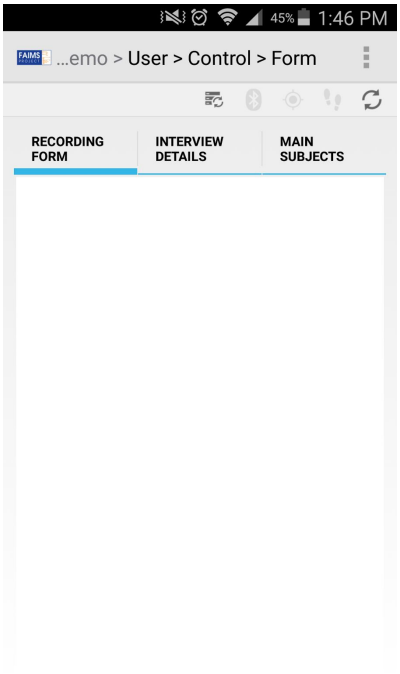


For the next step, we'll add new UI tab groups for each of our buttons: "Form" and "Diary" that we linked to in the last code block:

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

```
  </Main_Subjects>
</Form>


<Diary>
  <Diary>
  </Diary>
</Diary>
```

Make sure that the <Form> and <Diary> elements are written directly within the <module> so that the FAIMS-Tools correctly interprets them as tab groups. Now, under each of these new tab groups, we'll create opening and closing tags for each of the individual tabs. The "Form" 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 FAIMS creates the title for each tab, underscores will be replaced with spaces.



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:

```
<BIRTH_PLACE />
```

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 following:

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

If such characters must be included, the solution is to firstly give the element a sensible name without an apostrophe or colon:

```
<PARENTS_BIRTH_PLACE/>
```

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

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

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 the "PERSON" and "LANGUAGE_GROUP". We can use the "f" attribute to denote that in our new module:

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

For the GPS fields and their corresponding "Take From GPS" button, enter:

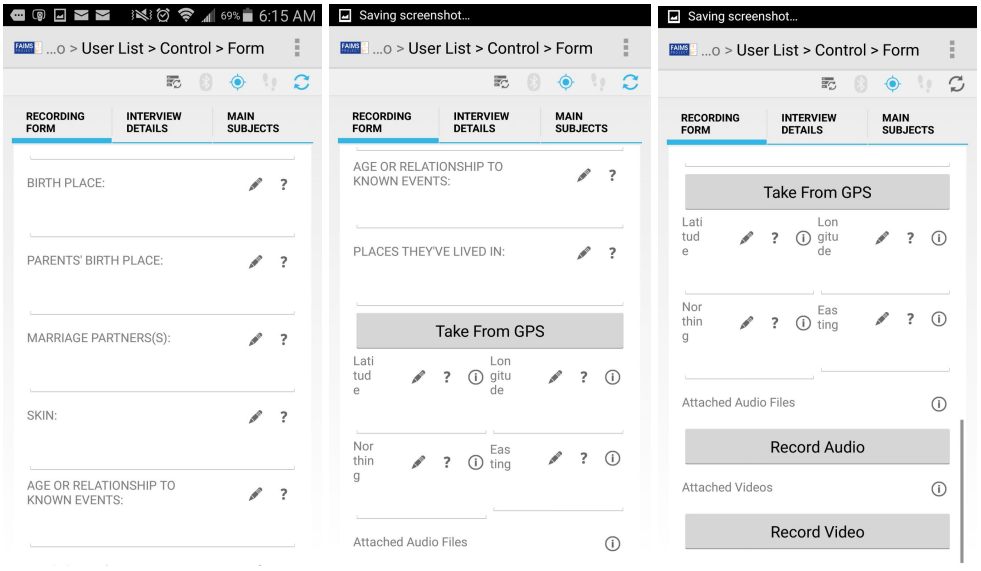
```
<gps />
```

`<gps>` is a special, 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.

To add fields for attached files, you can use the "Audio" and "Video" tags:

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

Simply set the type "t" to video or audio as required.

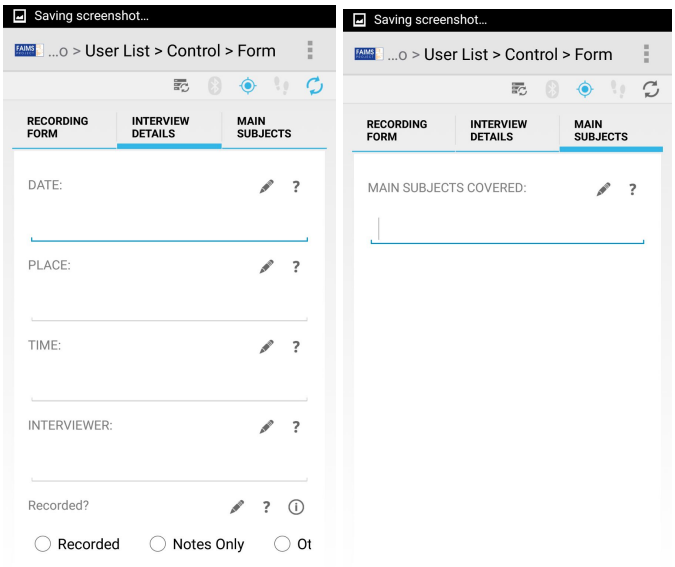


To add the radio buttons, use the code:

```
<Recorded t="radio">
  Recorded?
  <opts>
    <opt>Recorded</opt>
    <opt>Notes Only</opt>
```

```
    <opt>No</opt>
  </opts>
</Recorded>
```

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.

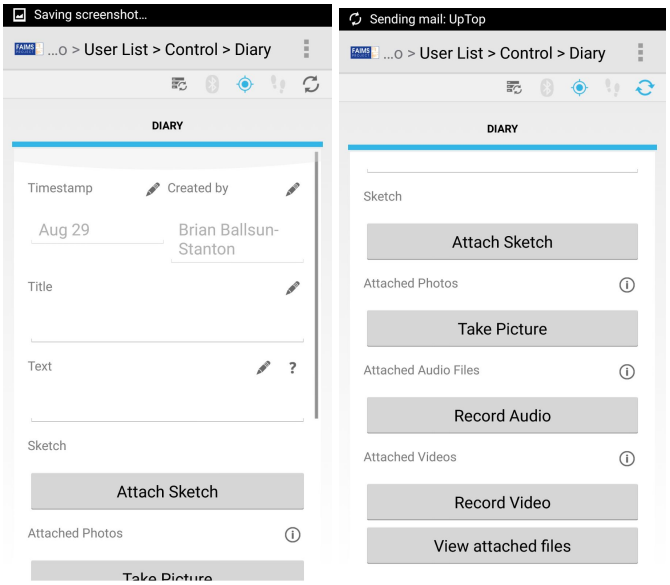


The "Timestamp" and "Created By" values are not actually fields that we'll allow users to enter. Instead, we'll make FAIMS set and update these fields when the record is created. For now, we'll put in the code:

<Timestamp/>

<Created_by/>

and then we'll need to do some work in our ui_logic.bsh file once it's generated to complete the fields.



Additional Features

Add a Picture Gallery

```
<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>
```

Hierarchical Dropdown

```
<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 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>
```

Using the Translation File

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 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>
```

Fetch Point from GPS
GPS functionality

Displaying Data as a Table in the App
Table views.

Human Readable Identifier Numbers

Restricting Data Entry to Decimals for a Field
-Single flag to denote as a number field

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. Example:

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

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

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

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

```
  <Element>                            <!-- This'll be a list -->
    <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>
```

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 can be hidden by adding suppressWarnings="true" to the opening <module> tag like so:

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

Annotation and Certainty

CSS and Validation files.

Exporting Data

Advanced FAIMS Programming

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/readme 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.

```
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')
  f                  Flags (See 'Flags')
  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.
  p                  In <opt> tags, equivalent to pictureURL attribute.
  suppressWarnings   Prevents warnings from being shown when equal to "true" and
                     present in the <module> tag. Does not suppress errors.
  t                  Type of GUI element (See 'Types'). If this attribute is
                     omitted from a view, t="input" is assumed.
```

```
BINDINGS

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

```
FLAGS

  hidden          Equivalent to faims_hidden="true".
  id              Equivalent to isIdentifier="true".
  noannotation    Equivalent to faims_annotation="false".
  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 ommits data schema entries.
  readonly        Equivalent to faims_read_only="true".
  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.
```

```
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:
    - <col>        One column in a <cols> tag
    - <cols>       Columns
    - <desc>       Description to put in the data schema
    - <module>
    - <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 the child of a <str>, contains <formatString> data.
    - <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. Doing so
  would make you a bad person.
```

```
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.
```

```
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 user 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.

The created arch16n entries are used in the generated UI and data schemas.

It should be carefully noted that replacing characters as described above can cause naming conflicts. For example, if the module.xml file contains the labels "I'm cool!" and "I'm cool?", the generated arch16n file will contain the following lines:

```
    I_m_cool_=I'm cool!
    I_m_cool_=I'm cool?
```

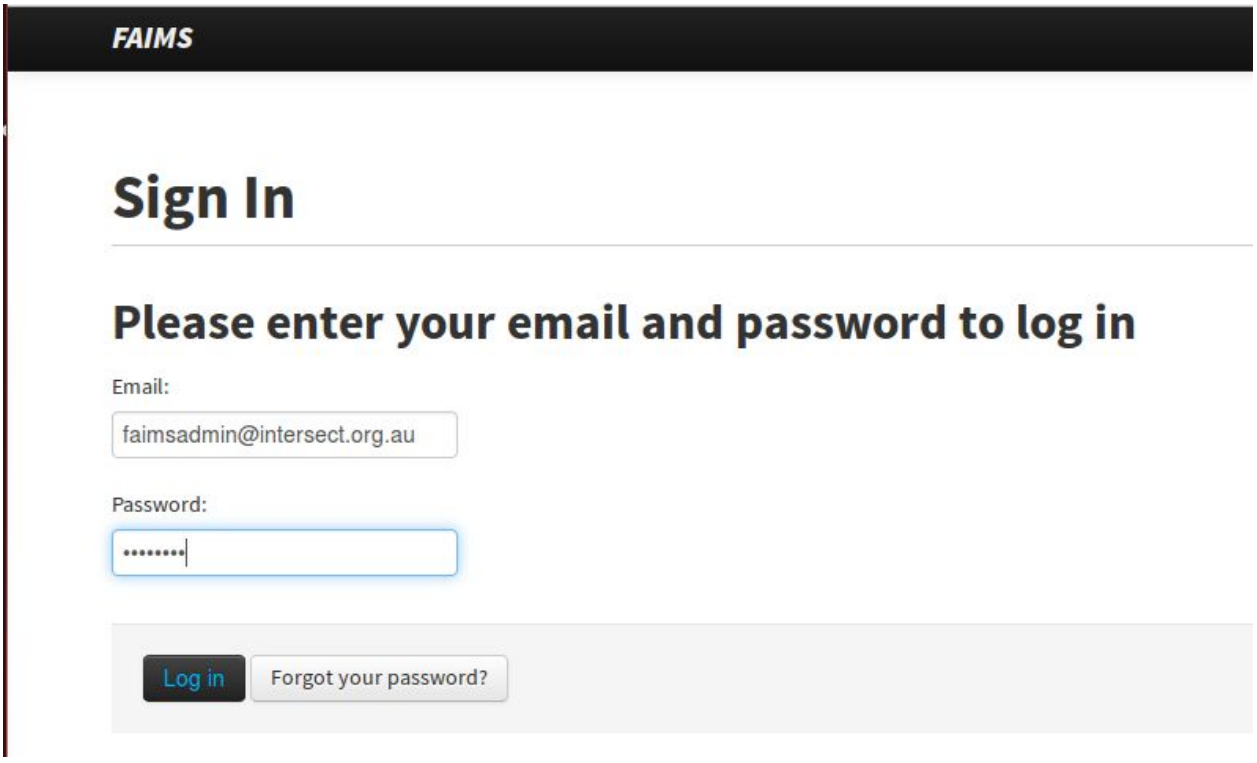
Moreover, the programmer is not warned if such a conflict exists, as, in practise, it assumed that such conflicts are very rare and checking for them in XSLT 1.0 violates Article 5 of 'The Universal Declaration of Human Rights'.

How to Deploy Modules

Opening the FAIMS Server Web Interface

Now that your FAIMS server is set up, you should be able to access it from any device on the same network as the server (and later, any device with a web browser connected to the internet). On a computer, navigate to the IP address that appears in the Firefox browser on your FAIMS Ubuntu installation.

Next, log into the FAIMS server using either the default admin account or an account that you set up (we recommend setting up individual accounts for each user). **Note:** The default FAIMS account is "`faimsadmin@intersect.org.au`", password "`Pass.123`".



Uploading Build Files to the FAIMS Server

1. Login to the Server
2. Click 'Show Modules'
3. Click 'Create Module'
4. Give your Module a name, and enter as much metadata as possible
5. Upload a Data Definition Schema (data_schema.xml).
6. Upload a User Interface Schema (ui_schema.xml)
7. Upload User Interface Logic (ui_logic.bsh)
8. Upload a Translation (Arch16n) file (faims.properties)

Let it combine and process them into a module, which will live on the server.

On your Android device(s), open the FAIMS app and connect to the server.

Download your module to your device and open the module.

Test Your Knowledge

Q1: Modules can be loaded by connecting your device to the computer via usb, then dragging the file over to the device with the file explorer. True or False?

Q2: FAIMS devices can sync together in the field, no server required. True or False?

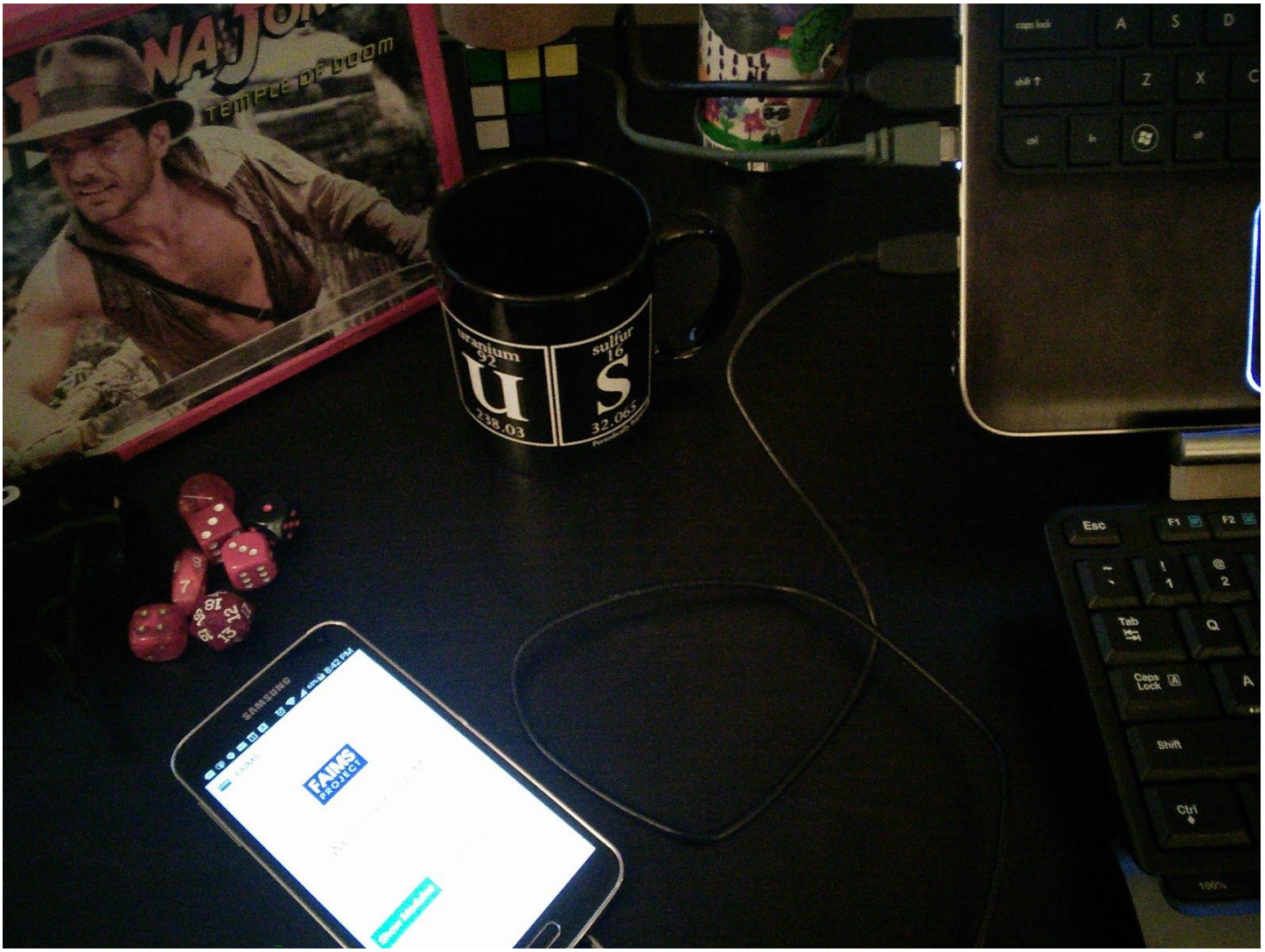
Exercise: Download the sample module here: https://github.com/FAIMS/SimpleExcavation. Next, transfer this module onto an attached Android device. List the steps and explain to a colleague how it works.

How to Debug Modules

When running your FAIMS modules from your Android device, you may encounter the occasional error where things don't work as expected, a dropdown or other UI element fails to function properly, or the module refuses to open. If you're lucky, you module programmer included an error message that identifies what error occurred, or even how to fix it. In many cases, these error messages won't appear as popup windows in the app, but can instead be accessed by monitoring the FAIMS app as it runs by connecting your Android device to your computer.

Connecting your Android Device to your Computer

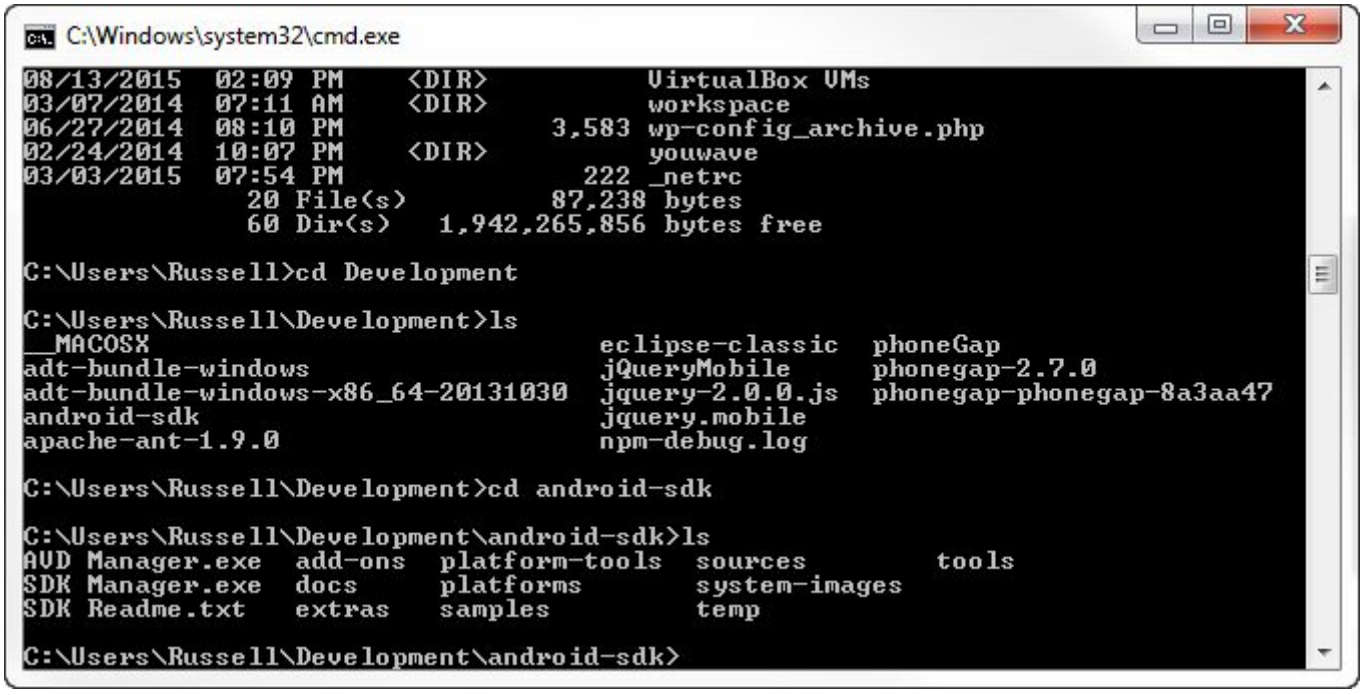
Up until now, we've worked mostly wirelessly, sending data back and forth from our FAIMS server to our Android device over a WiFi or cellular network. Now though, we'll break out one of the trusty USB cables that seem to litter offices and attics these days and physically connect an Android device to the laptop or desktop we've installed the Android SDK tools on.



Connect the USB cable to your device and connect the other end to your computer's USB port. Once completed, open up the FAIMS app on your device. The mug, lunch box, and dice are optional.

The Android Debug Monitor

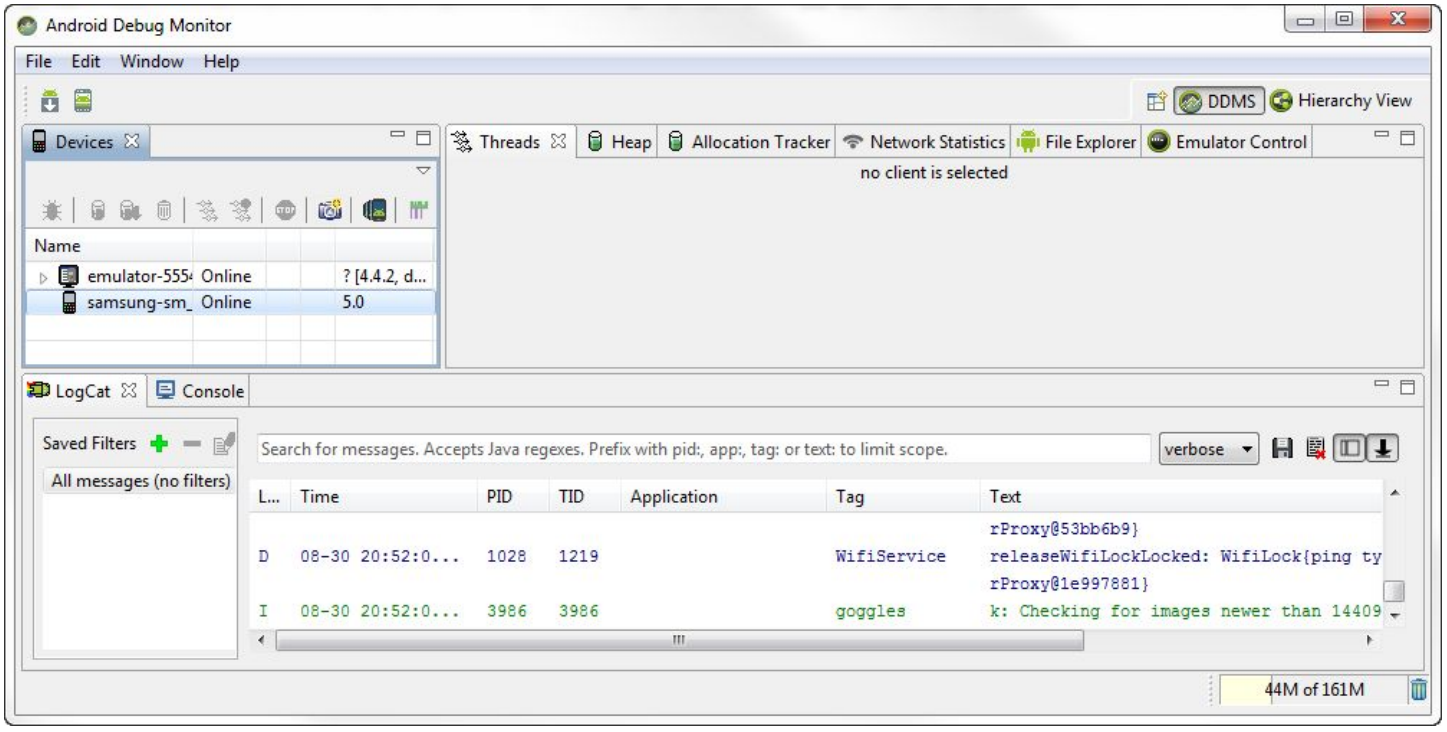
On your computer, open up the command window (on Windows, press the Windows key, and search for "cmd"). Using the change directory ("cd") and "ls" to list directory contents, navigate to where you installed the android-sdk.



While in this directory, type in:

monitor

and hit enter. Wait a few minutes and this command will open the Android Debug Monitor, a graphical program you can use to view messages and information from any attached Android devices and emulated devices.



The Android Debug Monitor show messages from your attached devices. Here we have two devices: an Android emulator and a physically attached Samsung phone.

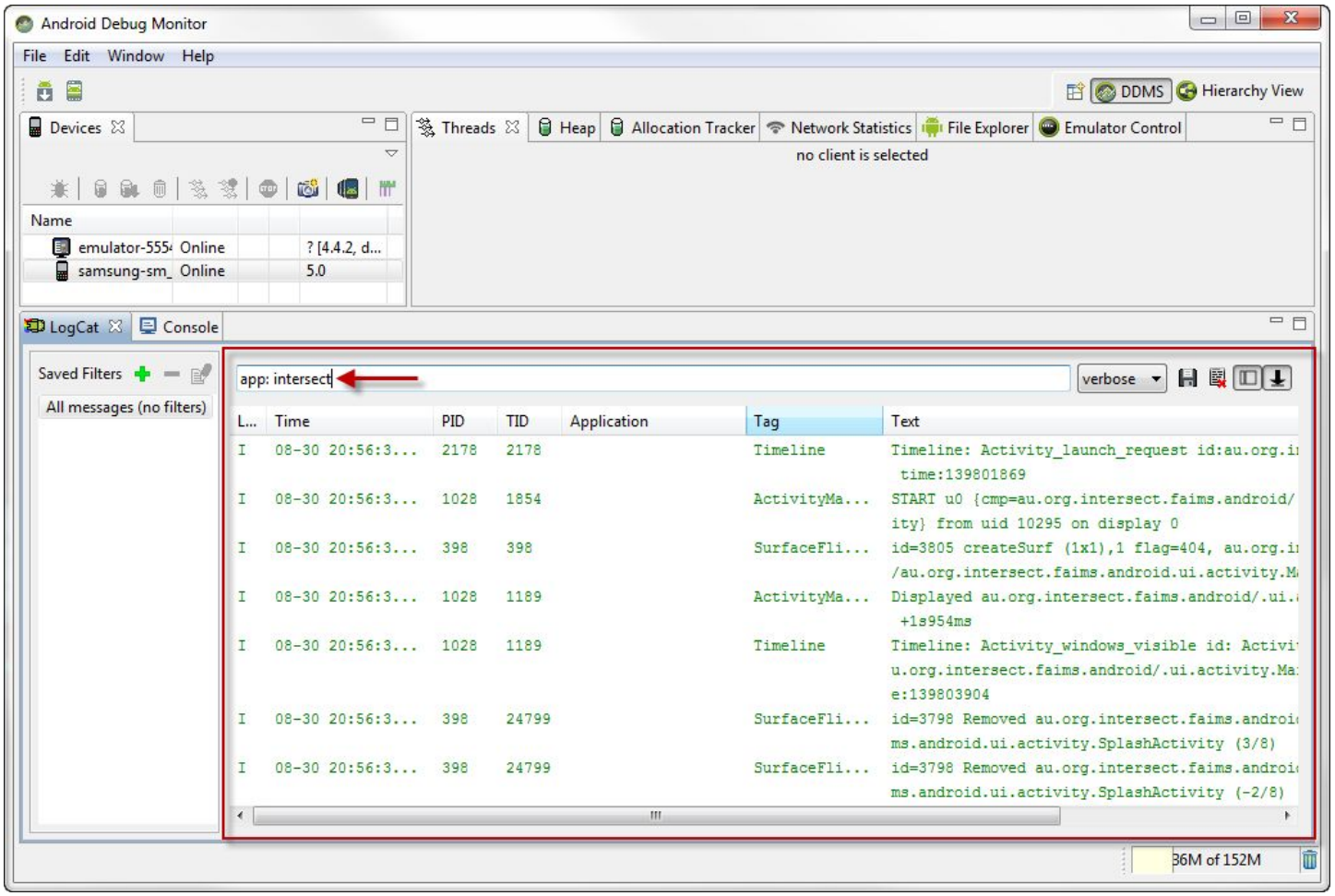
Filtering Monitor Messages by Application

The bottom portion of the Android Debug Monitor displays different messages from your applications running on your Android device. This is sometimes known as the "LogCat" window. When several applications are running, the window

will quickly fill up with messages and it may be hard to keep track of them all. Luckily, you can limit the displayed messages using the filter box. To see only messages from the FAIMS app, type in:

```
app: faims
```

and hit enter. Now only messages from the FAIMS app will appear in the window.



You can limit messages in the Android Debug Monitor by using the filter box to search for pid; app; tag, or text.

Identifying Log Messages

Logs are messages developers create to complete when apps complete certain actions or when special cases arise that the programmer anticipated. To search from log messages from applications, type:

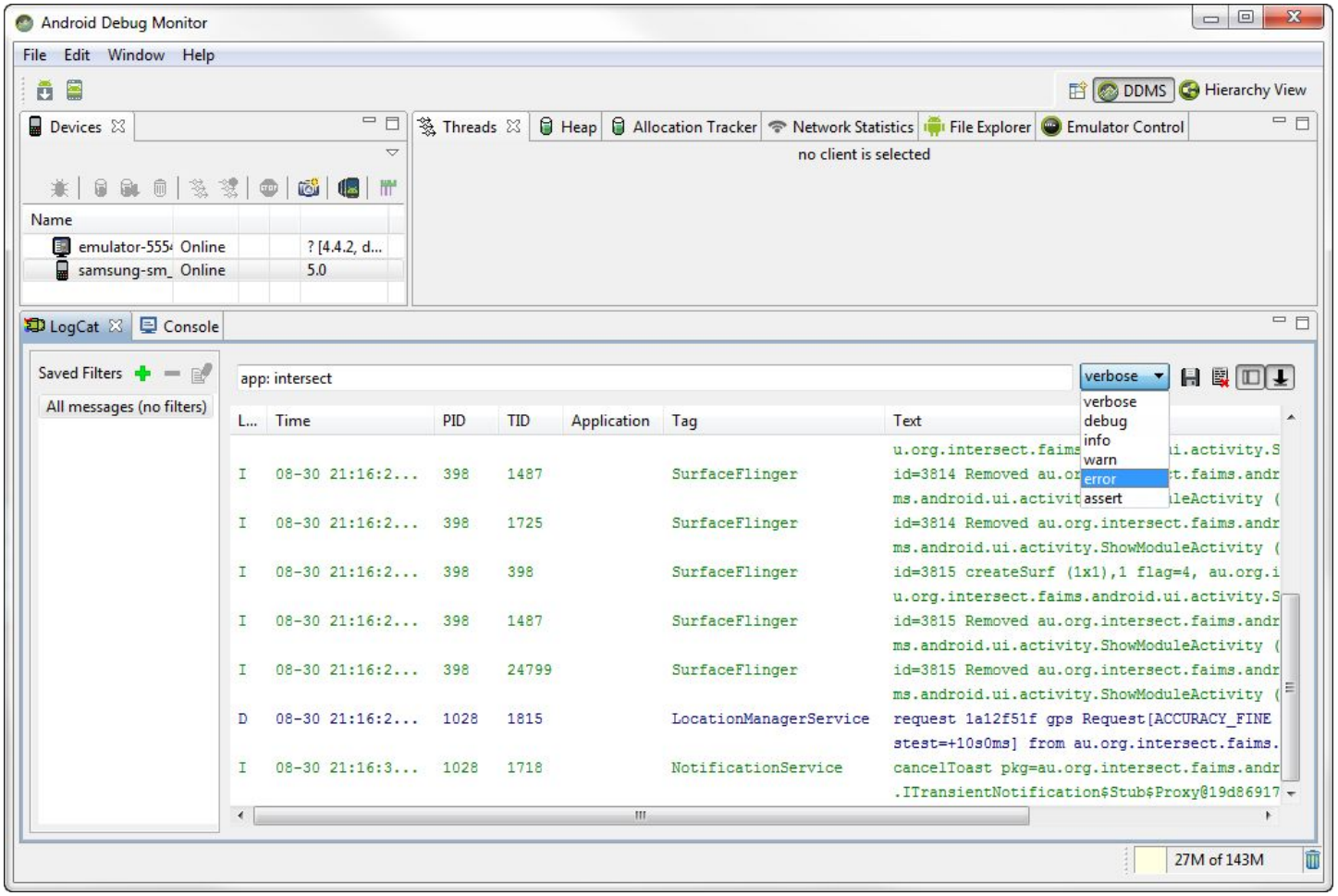
```
text: Log
```

into the LogCat search field.

Identifying Error Messages

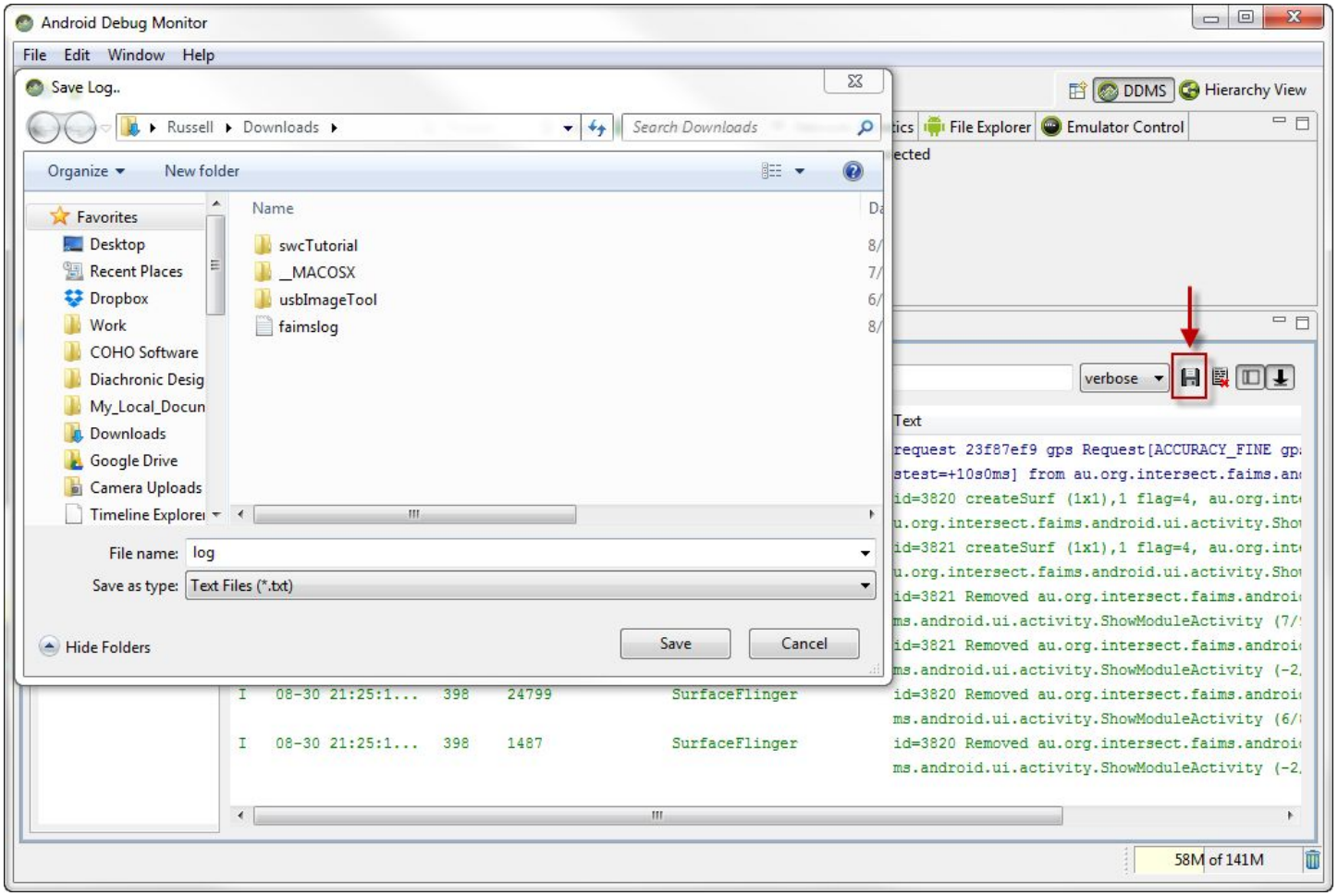
Errors differ from logs in that they are automatically sent out when a program encounters something that doesn't work properly. A FAIMS programmer may have specified a certain message to be shown when an error happens, but the Android operating system will also send out more general error messages when an app encounters a problem the programmer didn't specifically anticipate.

To further limit your LogCat output to show just error messages, you can click the dropdown menu to the right of the search field. By default, LogCat shows all messages with the "verbose" option. You can also select to show just debug, info, warn, or assert messages from the dropdown menu as well.



Now that you are able to display and identify log and error messages from the FAIMS app, you can use them to see when and where modules are having trouble. This information is invaluable when fixing bugs or communicating problems to support staff.

Finally, to save a copy of your LogCat output, click on the floppy disk icon to the right of the search field. This allows you to save a copy of the log as a text file that you can email or copy and paste to share log and error messages with the rest of your team.



Test Your Knowledge

Q1: You can monitor app messages from your Android device on your desktop using a wireless connection. True or False?

Q2: What is the difference between an error message and a log?

Exercise: Try searching the LogCat messages using a different filter, like "text: " or "tag: ." What messages appear or disappear depending on the filter? Can you find another way to filter out messages from just the "FAIMS" app?

FINIS

NOTES ON SECTION FOR DEBUGGING:

If there's a problem with the virtual machine, request assistance from FAIMs directly.

Cover soft-booting.

If they e-mail support, e-mail support.fedarch.org. The problem will be resolved *eventually.* Any sort of priority request can be obtained by purchasing a support package.