# 1 How to Code Modules

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

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

## 1.1 Introduction to `Module.xml`

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

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

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

## 1.2 The Module Creation Process

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

**Quiz 1.1** Test your knowledge: a gentle descent into `module.xml`

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

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

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

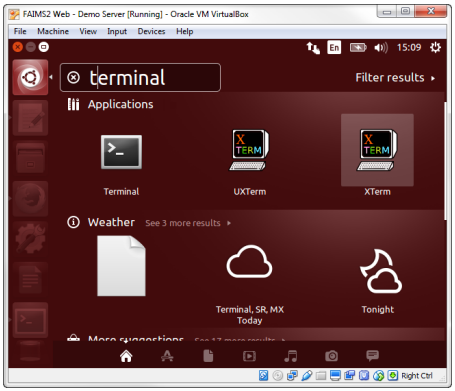


**Figure 1.1** How to open the terminal using Ubuntu.

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

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

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

Once you've finished coding, editing, and troubleshooting `module.xml` (and don't worry; we'll explain how to do all of that in the coming section), the instructions are complete and you're ready to generate your module. From the same directory you originally found `module.xml` in, run the commands in Codeblock 1.1 from the command line terminal. You won't have to type the rest of a command: just hit tab and the terminal will try to guess[1].

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

**Code 1.1** BASH shell commands to run the generator.

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

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

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

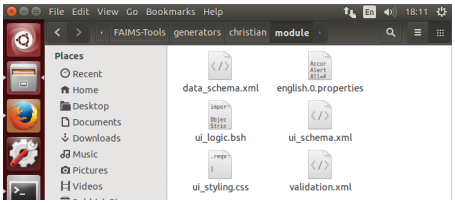


**Figure 1.2** The files generated by the generator.

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

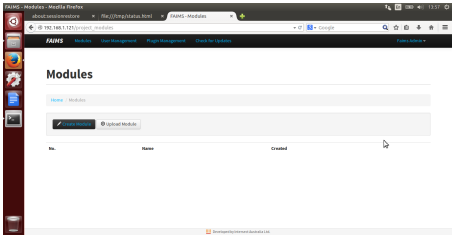


**Figure 1.3** Modules page on the FAIMS server.

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

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

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

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

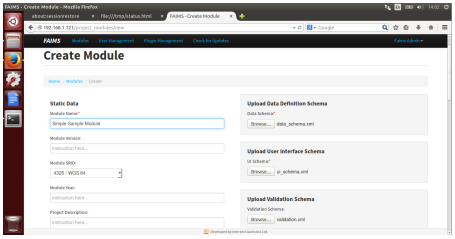


**Figure 1.4** Filling in the create module page.

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

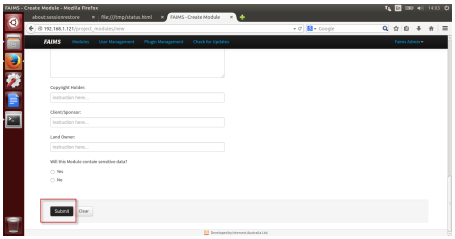


**Figure 1.5** The important submit button.

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

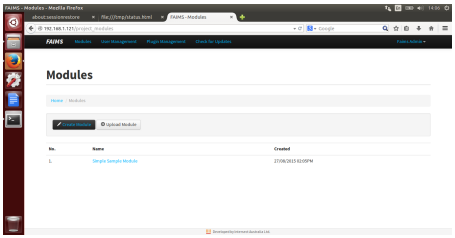


**Figure 1.6** Now the module appears.

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

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

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

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

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

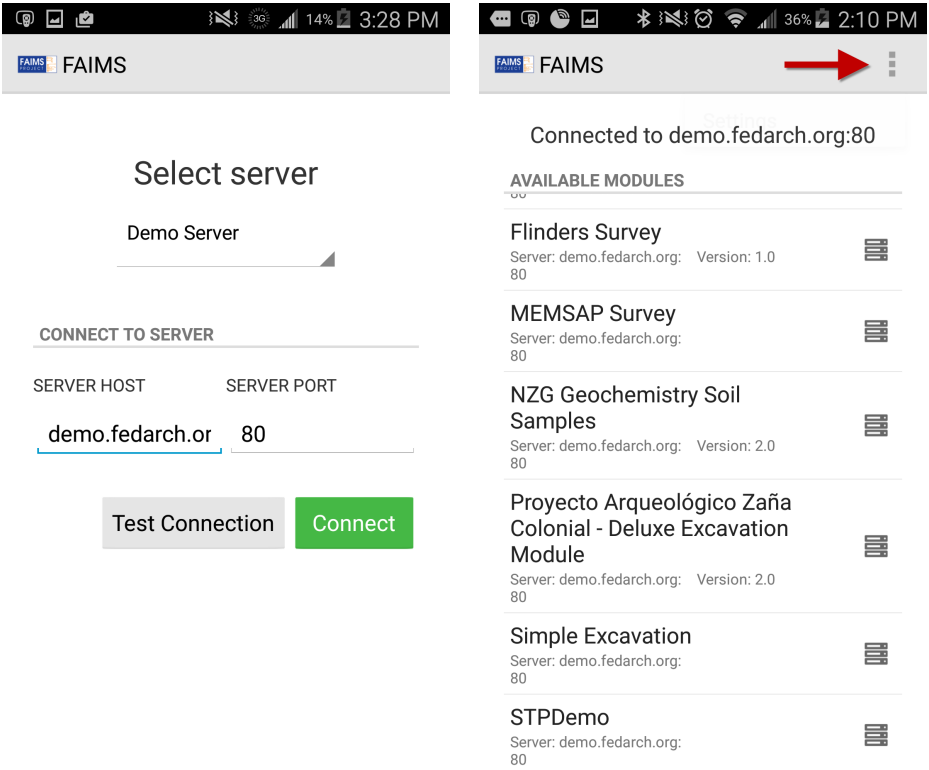


**Figure 1.7** Tap the three vertical dots to open "Select Server."

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

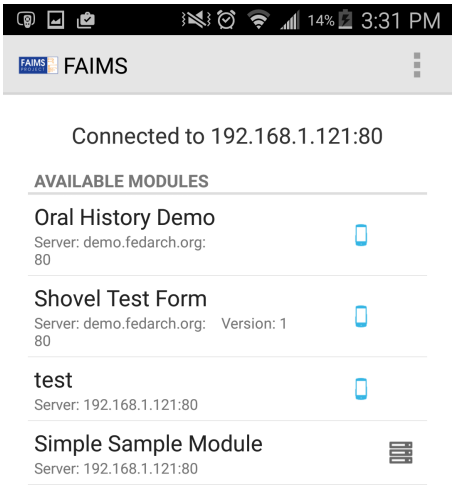


**Figure 1.8** Blue phone icons after the module is downloaded.

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

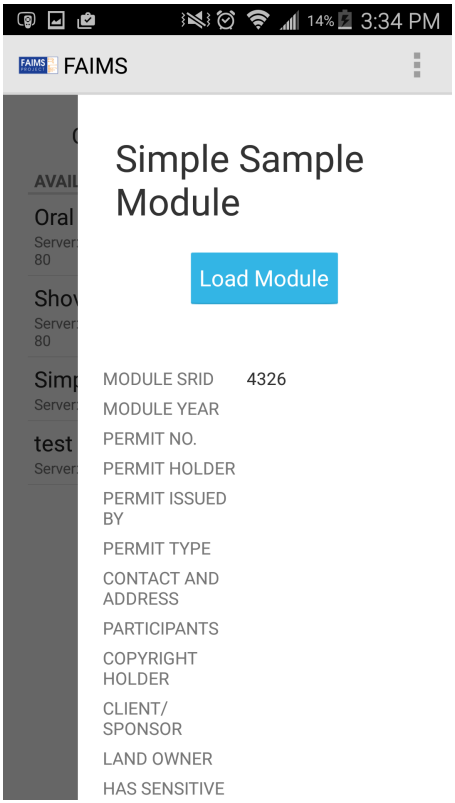


**Figure 1.9** Sidebar with module metadata.

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

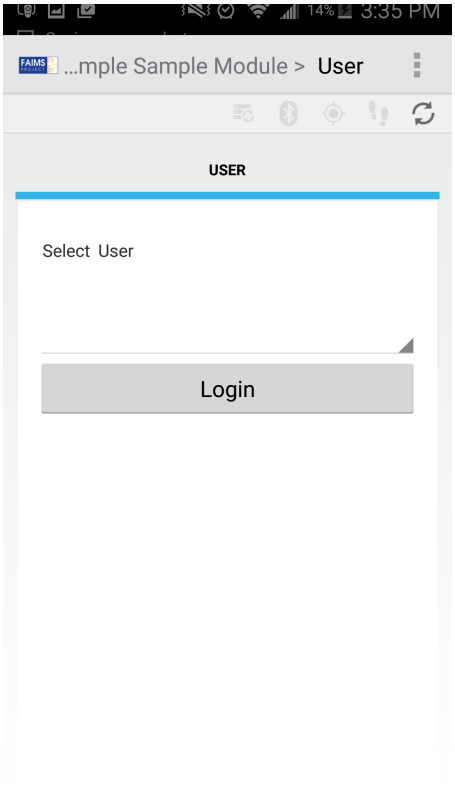


**Figure 1.10** User login menu.

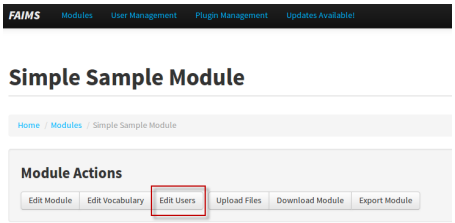


**Figure 1.11** Add or remove users throught the "Edit User" tab.

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

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

## 1.3 Understanding The Structure of XML Documents

Codeblock 1.2 is a very unusual description of a man named Richard. It's not code (it won't do anything if you plug it into FAIMS-Tools, because it contains no recognizable instructions), but it's written in a format very similar to what your module's code will look like. Take a minute and look it over and try to guess, just from the text and the way it's formatted, what Richard looks like.

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

**Code 1.2** An XML description of Richard.

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

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

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

Let's take that basic understanding and look at this section of actual code from the Oral History module's `module.xml` (Codeblock 1.3). You probably won't know what it means or does yet yet, but for now, just focus on the structure as we explain each part of it in detail.

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

**Code 1.3** Oral History's `module.xml`.

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

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

Tags contain information, and the information they contain is referred to as an *element*[2]. In our first example, the tags `<Richard>` and `</Richard>` defined the boundaries of the element Richard, and everything between them was contained within that element.

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

You may notice that there's two `</User>` tags above. You can probably guess from the way Codeblock 1.3 is indented that the very last `</User>` closing tag corresponds to the very first tag, `<User f="nodata">` (we'll explain later why this isn't closed by the tag, `</User f="nodata">`). However, it's worth mentioning that formatting doesn't really matter to FAIMS-Tools when it interprets how your code works; indenting when you start a new element is just a good practice for helping you keep track of your code. So in instance, and others like it: how does FAIMS-Tools decide which `</User>` closing tag belongs to which `<User>` opening tag?

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

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

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

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

You may have noticed that the XML Code Snippet 1.4 contains two tags which don't fall into either the *opening tag* or *closing tag* categories:

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

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

**Code 1.4** Empty element tags are an efficient way to create GUI elements with no additional elements inside them.

These are known as *empty-element tags*. Sometimes, an element is complete with only a single instruction. It's not creating something larger that has multiple qualities; it's just a single detail, such as a simple button the user can press. In these cases, it isn't really necessary to have both an opening and a closing tag, as there's nothing to put between them. The above tags are really just shorthand for the following in Code Snippet 1.5:

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

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

**Code 1.5**

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

**Quiz 1.2** Test your knowledge

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

## 1.4 Understanding The *FAIMS* XML Format

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

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

Consider for a moment the module in Figure 1.12 (left image). There are **GUI elements** (or, "useful parts you can interact with") which all belong to the "Recording Form" tab. If the user were to tap on a different tab, for instance the "Interview Details" tab, they would see a different set of GUI elements belonging to that different tab.

The right image in Figure 1.12 shows all the **tabs** which belong to a presently displayed **tab group** called "Form". Entering a different tab group would cause a different set of tabs to be displayed.

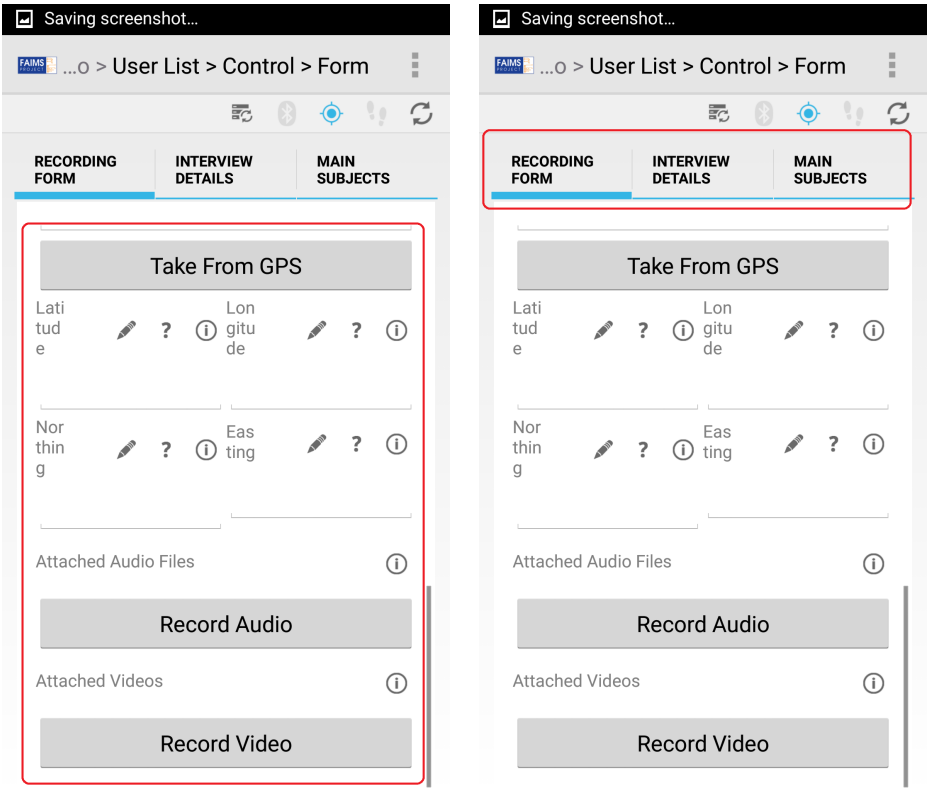


**Figure 1.12** GUI elements in a tab and tabs in a tab group.

Figure 1.13 shows two different tab groups—"Control" and "Form"—each containing their own sets of tabs.

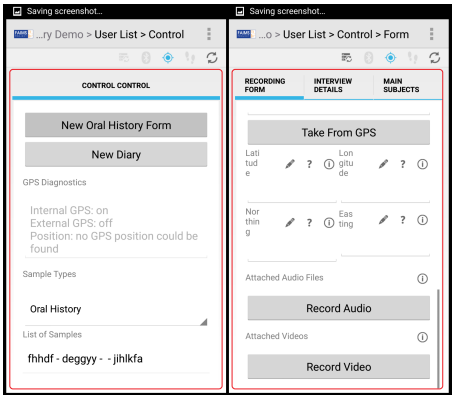


**Figure 1.13** "Control" and "Form" have different tabs and GUI elements.

So the hierarchy flows: your module has tab groups have tabs which have GUI elements. When you structure the elements of `module.xml`, it'll be contained exactly the same way. Your module element will contain tab group elements will contain tab elements which will contain GUI elements, many of which may be empty elements as they'll stand on their own without needing to contain anything else. Make sense so far? See what it looks like in practice in Figure 1.6.

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

**Code 1.6** The hierarchy of modules, tab groups, tabs, and GUI elements.

Remember that you're creating a version of `module.xml` that XML-Tools can read and interpret in creating your necessary files. Because XML-Tools understands the hierarchical structure we've just explained, it'll automatically understand whether something is a tab group or tab just from where it falls in the hierarchy. XML-Tools knows that `<Tab_Group>` is a tab group not because of its name, but because it appears straight away from the `<module>` tags without being contained in anything else.[3] Similarly, elements which appear directly within a tab group are automatically understood to be tabs. XML elements within tabs are interpreted as GUI elements.

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

**Code 1.7** Can you infer which elements are tab groups, tabs, and UI elements?

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

**Quiz 1.3** Test your knowledge

1. Review the structure in Codeblock 1.7.
2. Without knowing what's really inside the tags, can you figure out which elements are the tab groups, which are the tabs, and which are UI elements? If you have difficulty guessing that the "A" elements are tab groups, the "B" elements are tabs, and the "C" elements are the GUI, you may need to review the previous section.

## 1.5 Exploring a Simple Module

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

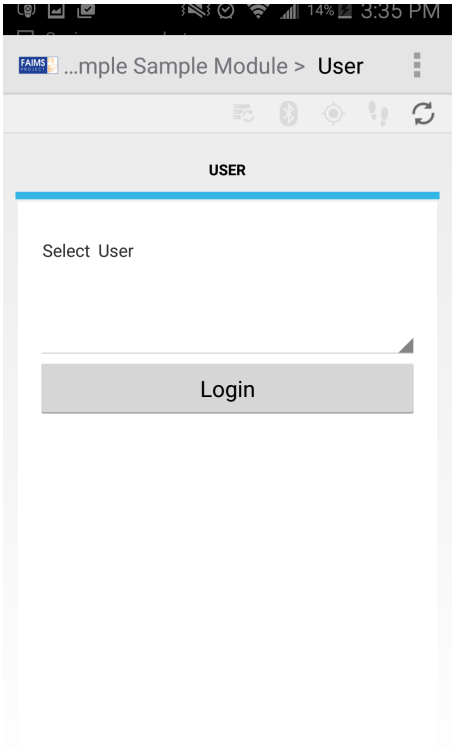


**Figure 1.14** A user login screen.

To create this first login screen (Figure 1.14), we used the XML code in Codeblock 1.8.

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

**Code 1.8** XML code for a user login screen.

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

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

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

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

So how do we determine what's in that drop down menu? In this case, by creating an attribute, `f="users"`, which in this context FAIMS understands to mean "get the list of usernames from the server and put them here."[4]

[4] **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)

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

So let's say we're using our module, we've just clicked the dropdown menu and chosen our user, then we've hit the button that says "Login." If you're using the finished sample module, you'll find yourself looking at the screen in Figure 1.15.

---

by selecting the null option and clicking the login button. If you want to prevent logging with the null user option (de facto avoiding the login), then you need to manually modify the `ui_logic.bsh` file after it's been generated or use `t=list`, following the guidelines below. Lists, unlike dropdowns, do not allow null elements so using a `t="list"` for login, as in the code below, prevents the problem of logging in with null user.
<User f="nodata"> <User f="noscroll"> <Select_User t="list" f="user" l="Control"/> </User> </User>

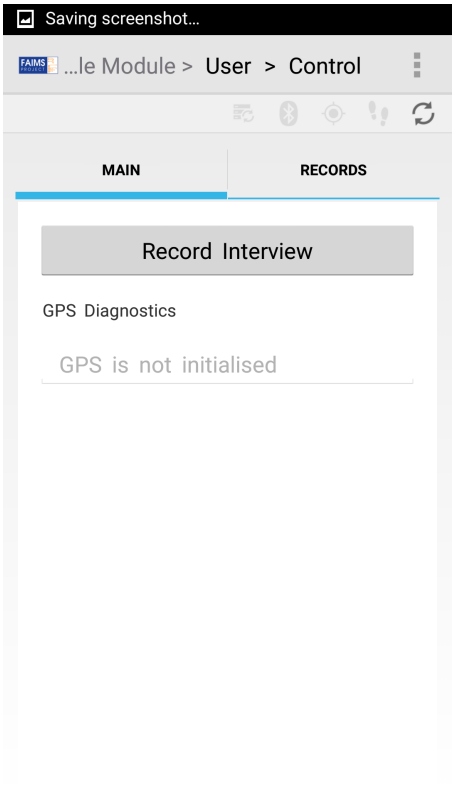


**Figure 1.15** The "main" tab.

Let's take a look at the code for this tab group here. As you review Codeblock 1.9, see if you can figure out which elements are tab groups, tabs, and GUI elements.

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

**Code 1.9** Matching XML to FAIMS elements.

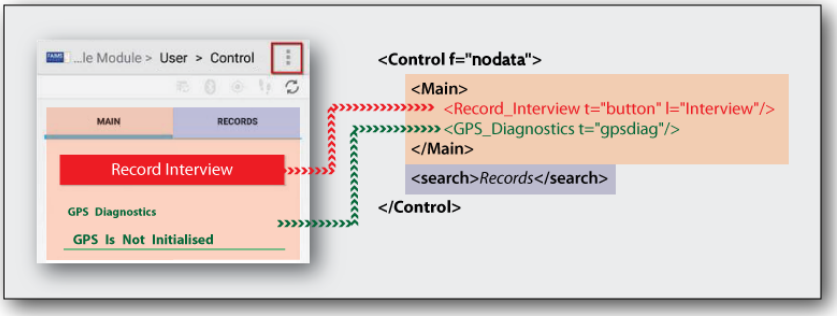


**Figure 1.16** How XML in `module.xml` describes different FAIMS Elements.

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

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

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

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

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

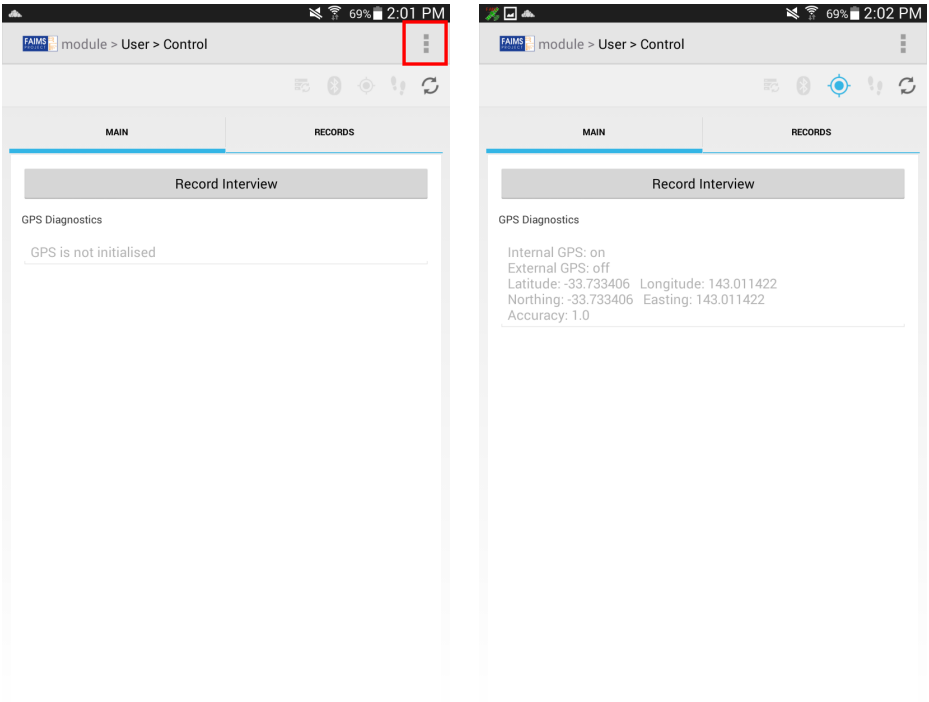


**Figure 1.17** Turn on internal GPS by tapping the dots in the red box.

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

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

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

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

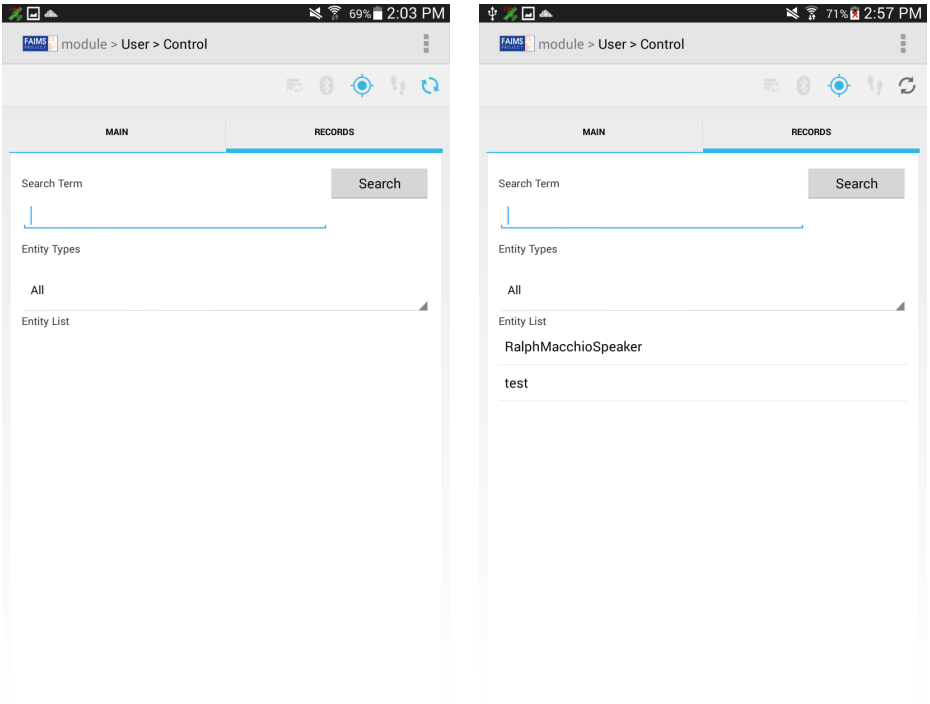


**Figure 1.18** The Search element before and after adding a few records.

Going back to the finished module, let's tap the "Main" tab and then click the "Record Interview" button, which, you may recall, will take us to "Interview." "Interview" looks like Figure 1.19:

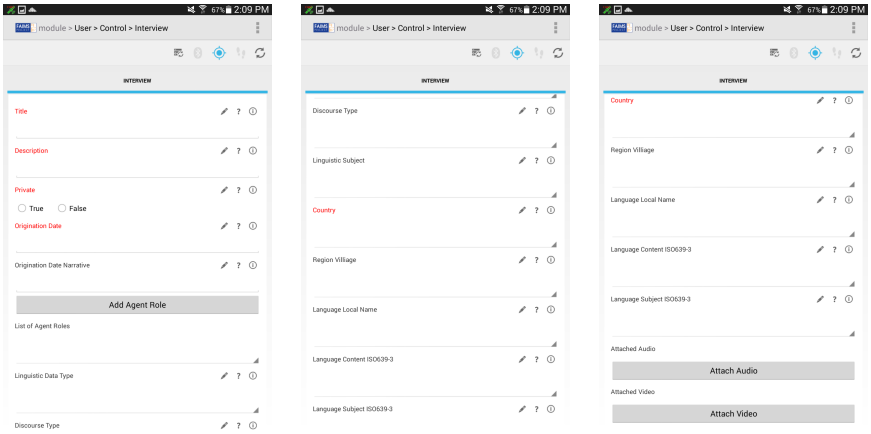


**Figure 1.19** Scroll to see all fields in the "Interview" tab.

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

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

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

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

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

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

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

The first element is `<Title>` (Codeblock 1.10).

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

**Code 1.10** Breaking down `<Title>`.

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

`<desc>` allows you to set a description that your users can access by tapping and holding for a few seconds on the info button as in Figure 1.20.

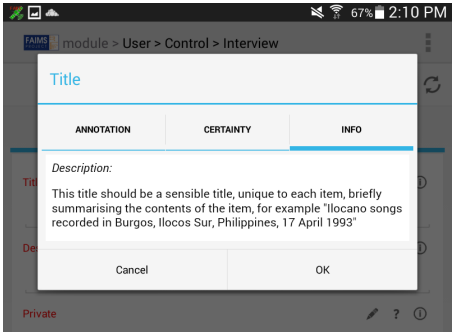


**Figure 1.20** A description pop-up appears.

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

and “False”). Review the code in Codeblock 1.11 and see if you can understand how all of this is accomplished.

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

**Code 1.11** XML for `<Private>`.

This next element (Codeblock 1.12), `<Add_Agent_Role>`, designates a button element (’t="button"’) that links to the tab group `Agent_Role`. This will allow users to register new Agent Roles.

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

**Code 1.12** XML for `<Add_Agent_Role>`.

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

Codeblock 1.13 designates a dropdown menu, `<List_of_Agent_Roles>`, which is populated with a list of `Agent_Role` records. Specifically, they will be the `Agent_Role` records which were saved using the button element above. The FAIMS-Tools knows these are the right records to display because the button and dropdown menu appear in the same tab group.

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

**Code 1.13** XML for `List_of_Agent_Roles`.

The final two element types used in this tab are the `t="audio"` and `t="video"` types (Codeblock 1.14). These two element types allow you to record audio and video files and attach them to your records.

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

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

**Code 1.14** XML for attaching audio and video.

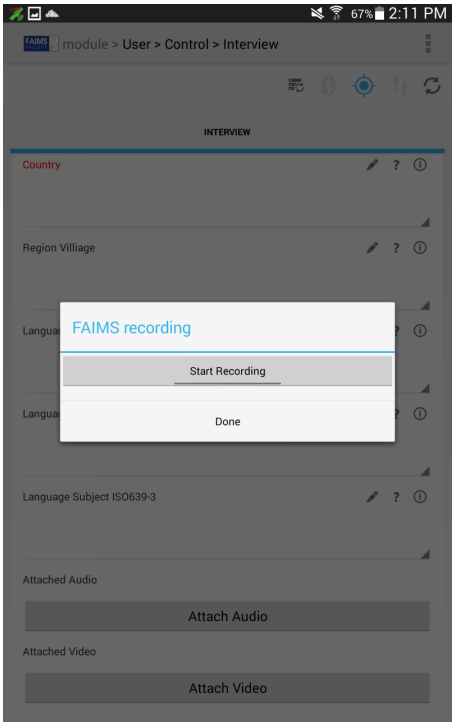


**Figure 1.21** With the audio UI element, you'll get a popup window that allows you to start and stop your audio recording.

The final elements are on the Agent Role tab group, and include a few element types we've already seen: two text input fields: `<First_Name>` and `<Last_Name>` both with flags that designate them as `ids` and a dropdown menu, `<Role>` which contains a few options and is also flagged as an `id` (Codeblock 1.15).

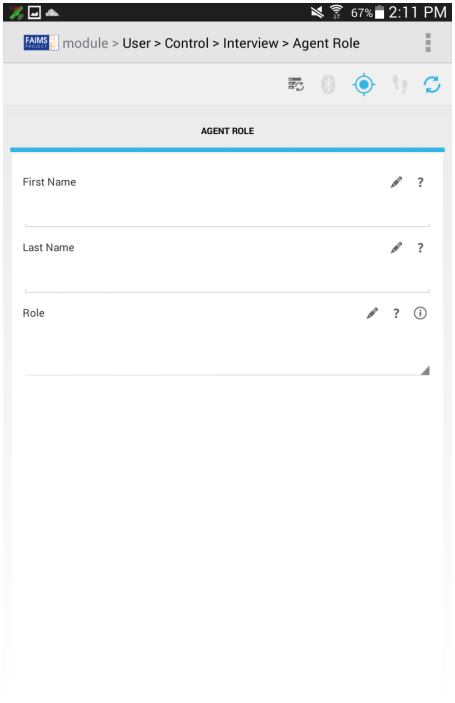


**Figure 1.22** "Agent Role" tab group appears when adding agent roles.

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

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

**Code 1.15** XML for “Agent Role” tab group.

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

Congratulations; you've just a simple module.

## 1.6 Iterating to Match the Oral History Module

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

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

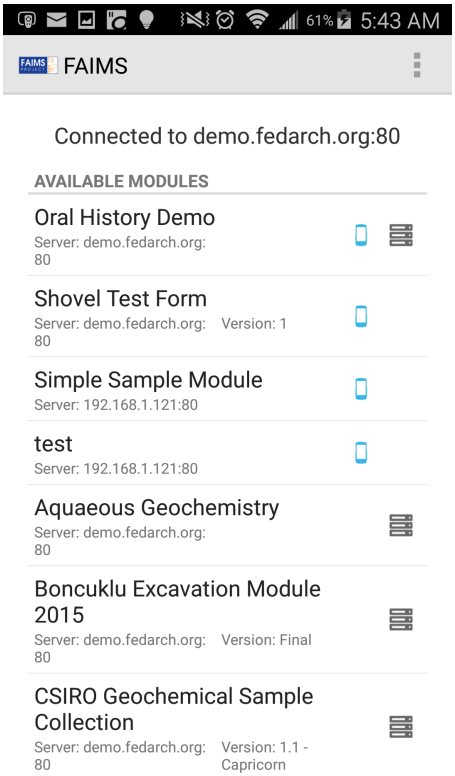


**Figure 1.23** Tap "Oral History Demo" to download.

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

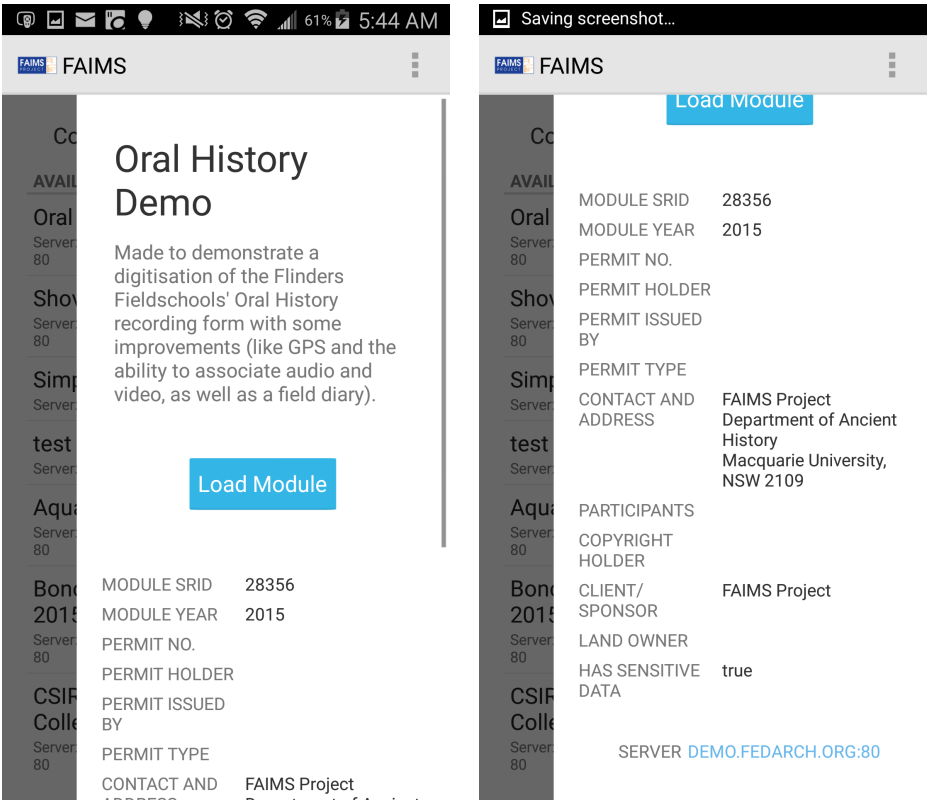


**Figure 1.24** More detailed metadata.

The first screen of the Oral History Demo (Figure 1.25) looks similar to our module.

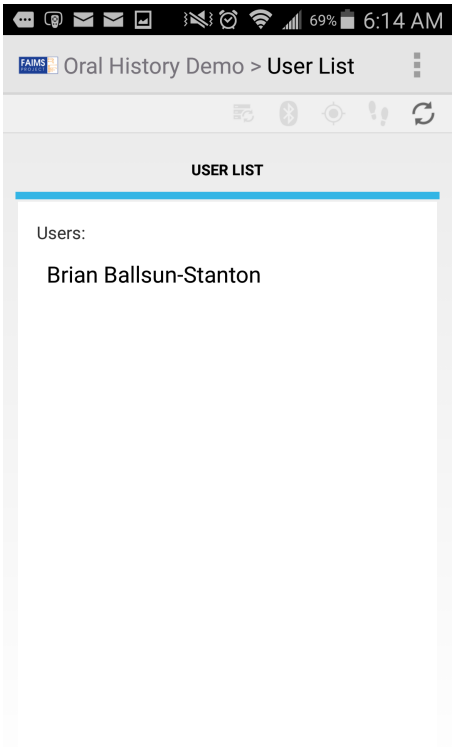


**Figure 1.25** Match your model to "Oral History Demo" by adding a User List.

In the same `module.xml` file we used for our version of Oral History update the code in Codeblock 1.16 to match Codeblock 1.17 (new code in **bold**).

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

**Code 1.16** Current code.

```
<User f="nodata">

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

</User>
```

**Code 1.17** This code replicates “Oral History Demo”.

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

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

Importantly, to download the updated version of the module, you must touch and hold the “Simple Sample Module” in the list of modules until the dialogue in Figure 1.26 is displayed.

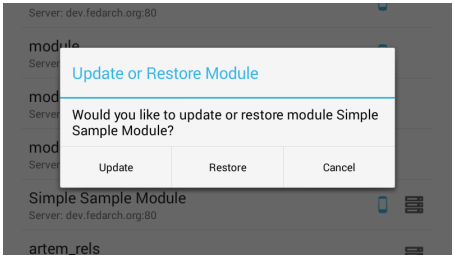


**Figure 1.26** Tap and hold “Simple Sample Module” to open this pop-up.

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

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

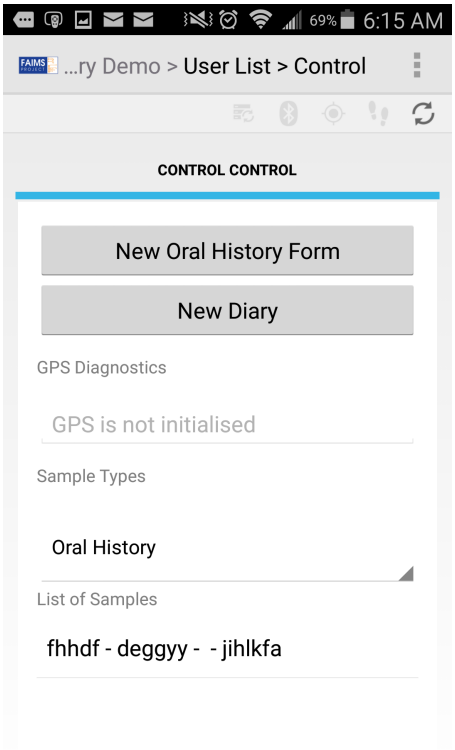


**Figure 1.27** "Oral History Demo" combines "Main" and "Record" into "Control".

On the screen in Figure 1.27, we need to condense a few elements from our original module. Change the code in Codeblock 1.18 to match the code in Codeblock 1.19.

Codeblock 1.19 replaces the "Main" and "Record" tab groups with a single one labelled "Control". (In a moment, we will define the "Form" and "Diary" tab groups linked to by the above buttons. We will also have to populate the "Sample Types" dropdown and "List of Samples" list by writing some additional code.)

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

**Code 1.18** Current code.

```
<Control f="nodata">

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

</Control>
```

**Code 1.19** Condense “Main” and “Record” into “Control”.

Save `module.xml` and run the generate.sh script again. Now use them to update your module again. You should see something similar to Figure 1.28 after selecting a user in the uploaded module.

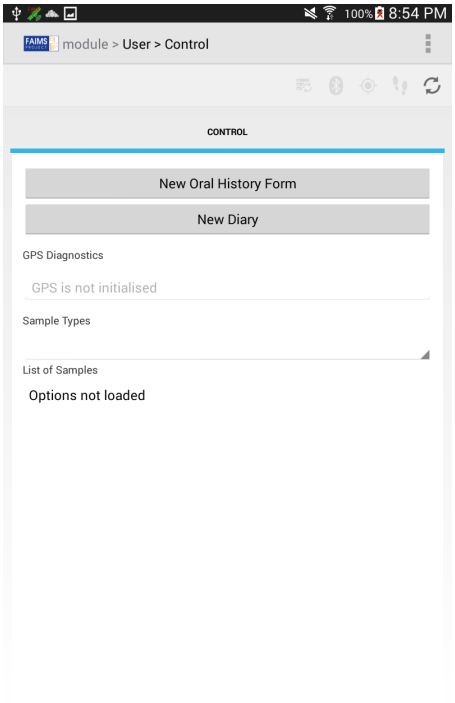


**Figure 1.28** "Control" tab in your updated module.

For the next step, we'll add new UI tab groups for each of our buttons: "Form" and "Diary" that we linked to in Codeblock 1.20.

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

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

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

**Code 1.20** Add “Form” and “Diary” tab groups.

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

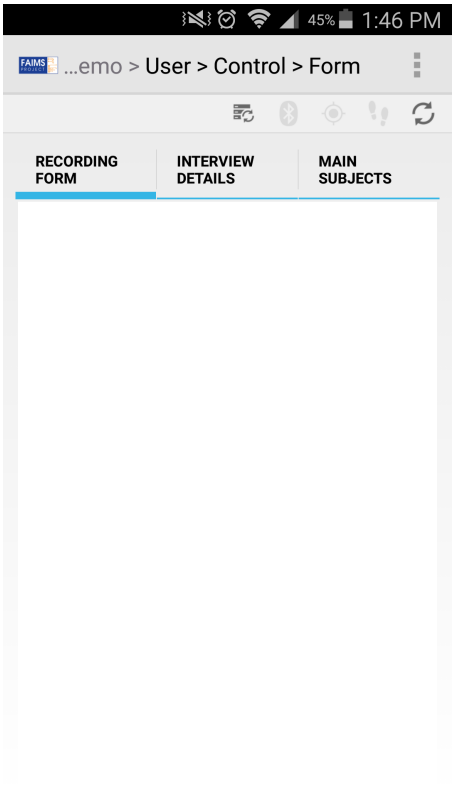


**Figure 1.29** Three newly created empty tabs.

Now we have the individual tabs for each section, but those tabs don't have any content just yet. For simple text fields, like "Birth Place" and "Parents' Birth Place" you can simply add a self-closing tag with the field's title as in Codeblock 1.21.

```
<BIRTH_PLACE/>
```

**Code 1.21** A self-closing "Birth Place" tag.

Also, non-alphabetical characters, like apostrophes are not allowed as tag names in XML, so the FAIMS-Tools would fail to generate a module which contains the code in Codeblock 1.22.

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

**Code 1.22** The apostrophe causes this to fail.

If such characters must be included, the solution is to firstly give the element a sensible name without an apostrophe or colon as in Codeblock 1.23.

```
<PARENTS_BIRTH_PLACE/>
```

**Code 1.23** First name the element without an apostrophe.

Then, to make FAIMS-Tools display the apostrophe and colon in the GUI, write them as the element's text as in Codeblock 1.24.

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

**Code 1.24** Write any special characters in the element's text.

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

Now, note that every tab group which you intend to save requires at least one *identifier*. In the original Oral History module, the identifiers were `PERSON` and `LANGUAGE_GROUP`. We can use the `f` attribute to denote that in our new module as in Codeblock 1.25.

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

**Code 1.25** Denoting an *identifier*.

For the GPS fields and their corresponding "Take From GPS" button, refer to Codeblock 1.26. `<gps>` is a special, self-contained shortcut tag that FAIMS will replace with several fields for the latitude, longitude, Easting, and Northing, as well as a button for inserting this data from GPS.

```
<gps/>
```

**Code 1.26** XML for GPS fields and button.

To add fields for attached files, you can use the `Audio` and `Video` tags, as in Codeblock 1.27. Simply set the type `t` to video or audio as required.

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

**Code 1.27** XML for attaching audio and visual files.

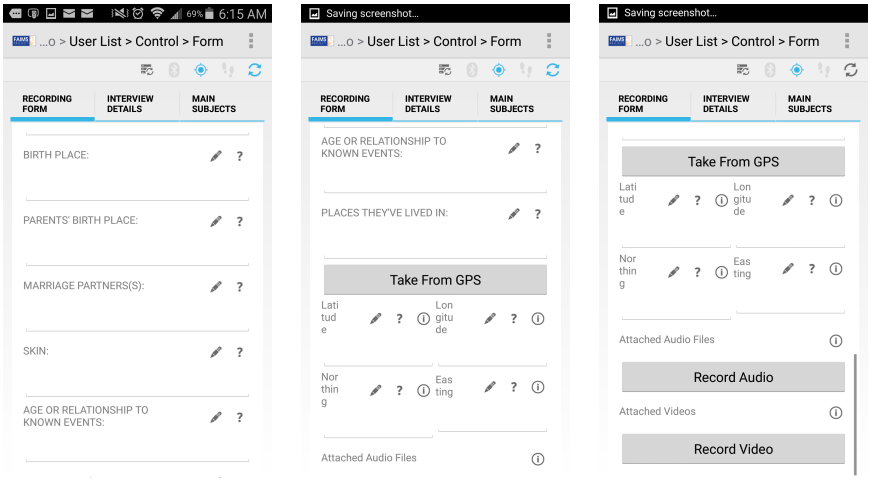


**Figure 1.30** Examples of simple text fields, GPS input, and attaching files.

To add the radio buttons, use the code in Codeblock 1.28.

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

**Code 1.28** XML for radio buttons.

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

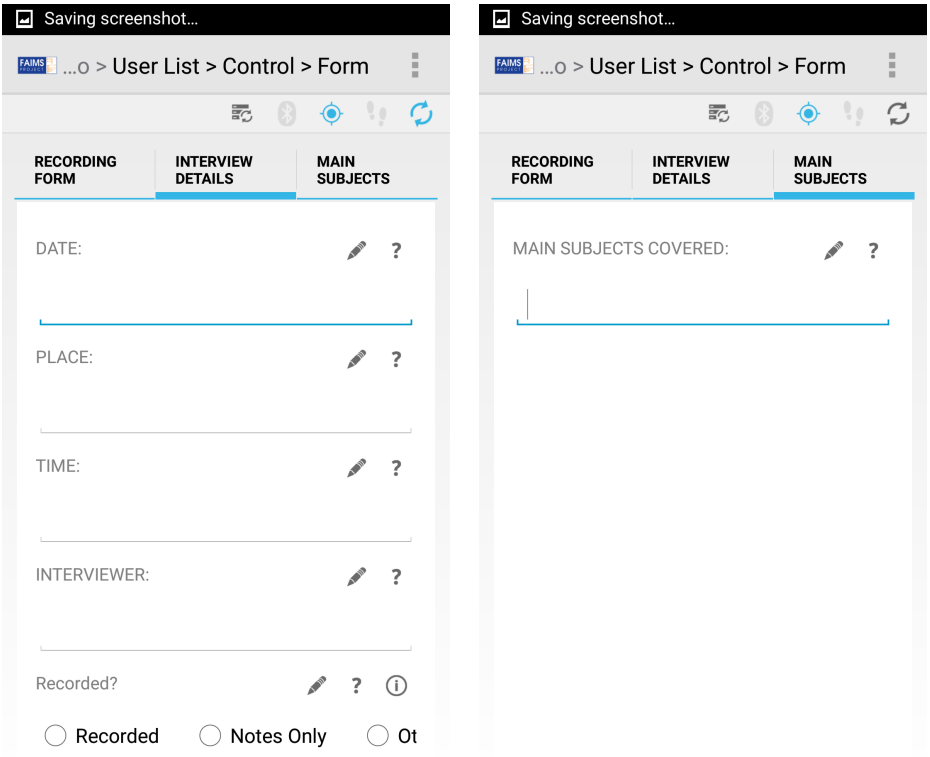


**Figure 1.31** Radio buttons in "Interview Details" and a simple text field in "Main Subjects".

The "Timestamp" and "Created By" values are not actually fields that we'll allow users to enter in manually, so we won't put in a data entry field. Instead, we'll make FAIMS set and update these fields when the record is created. Since there's no simple shortcut for that, for now we'll put in the code in Codeblock 1.29.

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

**Code 1.29** XML for “Timestamp” and “Created by”.

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

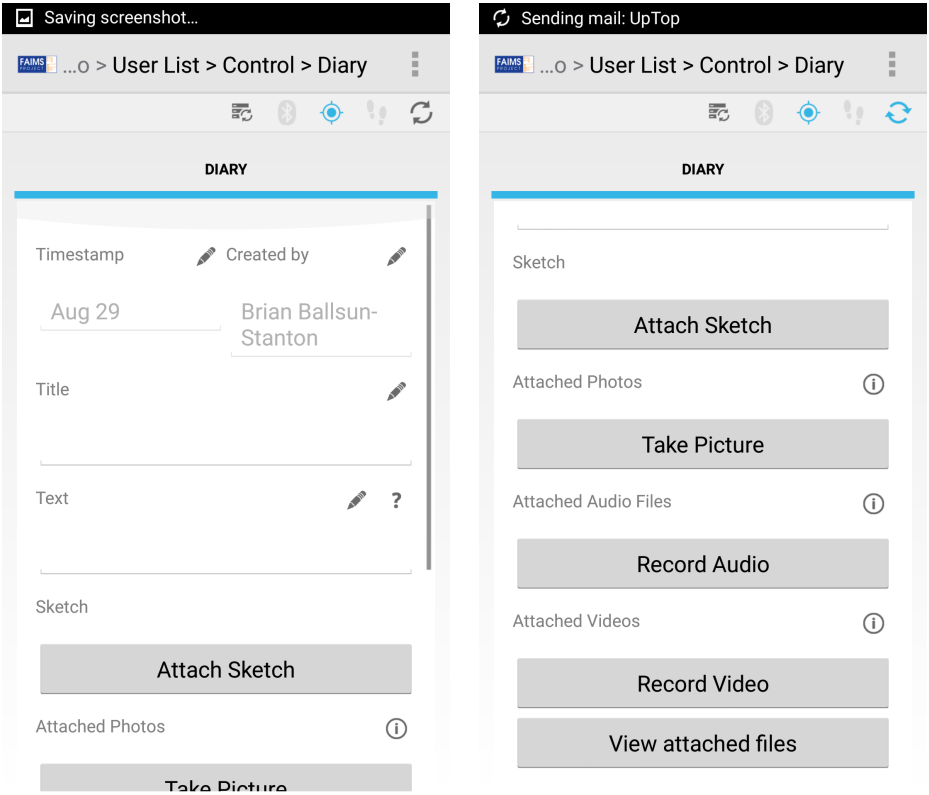


**Figure 1.32** "Timestamp" and “Created by” are automatically filled in by FAIMS.

## 1.7 Additional Features

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

### 1.7.1 Add a Picture Gallery

Remember when we told you how to add a tarball of images to your module? If you use the element type "picture," you can allow user selectable options to be displayed as these images. Take a look at the tab in Codeblock 1.30: Each picture is referenced by the tag, then labeled with the non-tagged text inside the element.

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

**Code 1.30** XML for a picture gallery.

### 1.7.2 Hierarchical Dropdown

"Hierarchical dropdown" is a fancy way of saying "selecting one option sometimes brings up more specific options." You do this very simply, by including `<opt>` tags inside of other `<opt>` tags. For example, Codeblock 1.31.

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

**Code 1.31** XML for hierarchical dropdowns.

### 1.7.3 Using the Translation File

### 1.7.4 Autonumbering

Basic autonumbering can be achieved using a combination of the `f="autonum"` flag and the `<autonum/>` tag. By flagging an input with `autonum`, one indicates to the FAIMS-Tools that the ID of the next created entity—the entity containing the flagged field—should be taken from the corresponding field generated using the `<autonum/>` tag. For instance the `Creatively_Named_ID` in Codeblock 1.32 will take its values from a field in Control which is generated by the use of the `<autonum/>` tag.

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

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

**Code 1.32** XML for autonumbering.

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

Multiple fields can be flagged as being autonumbered as in Codeblock 1.33.

### 1.7.5 Restricting Data Entry to Decimals for a Field

- Single flag to denote as a number field

### 1.7.6 Type Guessing for GUI Elements in FAIMS-Tools

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

If the XML tags do not contain a set of `<opts>` tags nor the `f="user"` flag, `t="input"` is assumed. For example, Codeblock 1.34.

If the XML tag is flagged with `f="user"`, `t="dropdown"` is assumed as in Codeblock 1.35.

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

**Code 1.33** XML for autonumbering multiple fields.

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

**Code 1.34** Without `<opts>` or `f="user"`, `t="input"` is assumed.

If the XML tags contain an `<opts>` element and no descendants with `p` attributes, `t="list"` is assumed. For example, Codeblock 1.36.

If the XML tags contain an `<opts>` element and one or more descendants with p attributes, `t="picture"` is assumed. For example, Codeblock 1.37.

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

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

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

**Code 1.35** When flagged with `f="user"`, `t="dropdown"` is assumed.

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

**Code 1.36** Tags containing `<opts>` with no descendants with `p` attributes are assumed to be lists.

can be hidden by adding `suppressWarnings="true"` to the opening `<module>` tag as in Codeblock 1.38.

### 1.7.7 Annotation and Certainty

### 1.7.8 Exporting Data

## 1.8 Advanced FAIMS Programming

### 1.8.1 `module.xml` Cheat Sheet

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

```
                    FAIMS TOOLS' MODULE AUTOGENERATOR

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

DEPENDENCIES AND SETUP

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

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

ALTERNATIVE SETUP (DOCKER)

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

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

QUICK START

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

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

**Code 1.37** Tags containing `<opts>` and one or more descendants with `p` attributes are assumed to be picture galleries.

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

**Code 1.38** XML to suppress warnings.

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

PARAMETERS AND USAGE

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

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

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

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

SCHEMA ATTRIBUTES

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

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

BINDINGS

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

FLAGS
```

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

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

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

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

---

TYPE GUESSING

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

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

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

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

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

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

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

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

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

---

TYPES

Types of GUI element:

- audio — `<select type="file" faims_sync=true/>` `<trigger/>`
- button — `<trigger/>`
- camera — `<select type="camera" faims_sync=true/>` `<trigger/>`
- checkbox — `<select/>`
- dropdown — `<select1/>`
- file — `<select type="file" faims_sync=true/>` `<trigger/>`
  File list with a button to add a file
- gpsdiag — `<input faims_read_only="true"/>...`
- group — `<group/>`
- input — `<input/>`
- list — `<select1 appearance="compact"/>`
- map — `<input faims_map="true"/>`
- picture — `<select1 type="image"/>`

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

RESERVED ELEMENT NAMES AND RECOMMENDED NAMING CONVENTIONS

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

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

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

INTENDED PURPOSE OF THE <rels> TAG

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

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

SEMANTICS OF <cols> TAGS

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

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

THE FAIMS TOOLS FORMAT-STRING SPECIFICATION

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

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

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

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

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

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

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

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

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

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

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

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

---

AUTONUMBERING

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

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

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

Multiple fields can be flagged as being autonumbered like so:

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

---

CHILD ENTITIES

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

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

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

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

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

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

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

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

---

LABELS

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

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

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

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

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

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

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

---

GENERATION OF THE ARCH16N FILE

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