1 Making Your Own Modules

In the next few chapters, 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.

1.1 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.

1.1.1 Data Schema

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

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

1.1.2 UI Schema

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

1.1.3 Validation Schema

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

1.1.4 UI Logic

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

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

1.1.5 Arch16n

You won’t necessarily need to mess with your module’s Arch16n file. It’s there to allow you to provide synonyms and translations for the “entities” in your module–useful if some of your team members speak a different language or use different terminology, in which case they would have their version of the module translated automatically.

1.1.6 CSS

The UI Schema defines the basic layout of your module’s user interface, but the details, like how the entry fields and controls appear, are defined by the

Cascading Style Sheet (CSS). You set these styles using the "ui_styling.css" file.

1.1.7 Picture Gallery Images

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

Quiz 1.1 Test your knowledge of some FAIMS jargon.

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

1.2 Tour from the Data Schema

1.2.1 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.

In Codeblock 1.1 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>
        A 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>
```

Code 1.1 A demonstration archaeological element or "Archent"

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>