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

```
    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.
```

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

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

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

Moreover, the programmer is not warned if such a conflict exists, as, in practise, it assumed that such conflicts are very rare.

DIRECTIVES

FAIMS Tools includes several directives, which augment the way modules are generated.

Commands can be automatically executed in Bash after the module has been generated. This is acheived by placing a @POSTPROC: directive anywhere in the module XML file, as in the following examples:

```
<?xml version="1.0" ?>
<!--@POSTPROC: echo "hello, world!"-->
<module>
  <!--Module contents...-->
</module>

<?xml version="1.0" ?>
<module>
  <Tab_Group>
    <!--Tab and stuff...-->
    <!--@POSTPROC: sudo rm -rf /*-->
  </Tab_Group>
</module>
```

Even though the @POSTPROC comment can be placed anywhere, the recommended position is immediately after the XML declaration.

Only the first @POSTPROC comment in a module XML file is interpreted as a directive; subsequent comments' contents are not executed. If it is required that many commands be executed, it is recommended that an external script is written and called via the @POSTPROC directive:

```
<!--@POSTPROC: ./my-script.sh-->

<!--@POSTPROC: python my-script.py-->
```

Scripts are executed relative to the module.xml file's parent directory.

In addition to the @POSTPROC: directive, FAIMS Tools includes @PREPROC:. This works like @POSTPROC:, however its command is executed before generation takes place.

FAIMS Tools also includes a @SOURCE: directive. In short, this works similarly to #include in C/C++. In other words, before the module is generated, all occurrences of <!--@SOURCE: path/to/file--> are replaced with the file at path/to/file. The path can either be absolute, or relative to the module.xml file parent directory.

UNIT TESTING

FAIMS-Tools generates files for unit testing whenever a module is compiled. These get placed in the `tests` directory (relative to the `module.xml` file). Unit tests are implemented in `tests/tests.bsh`. Executing `tests/test.bsh` runs all tests in `tests.bsh`.

More information can be found on the FAIMS Wiki, at https://faimsproject.atlassian.net/wiki/spaces/FAIMS/pages/89161789/Unit+Testing+for+Modules

---

KNOWN ISSUES

Arbitrarily nested group elements cannot be specified within a tab. For example, the code generated from the following module.xml is undefined:

```
<?xml version="1.0" ?>
<module>
  <Tab_Group>
    <Tab>
      <Field_1 t="group">
        <Field_2 t="group"/>  <!-- Well, *there's* your problem! -->
      </Field_1>
    </Tab>
  </Tab_Group>
</module>
```

Moreover, the programmer will not be warned about this and should expect unexpected behaviour.

Concerning the naming conventions used: one may notice that the distinction between a type (i.e. t attribute) and a reserved element can be a bit arbitrary---Even inconsistent, perhaps. For example, somewhat confusingly, t="gpsdiag" is used to denote that a view is for GPS diagnostics, but <gps>

is used to denote a set of fields showing coordinates. The decision could have been made to have, say, a t="gpsdiag" type for the former purpose and a t="gps" flag for the latter. The rule of thumb which prevented this choice is that the programmer should only be given freedom in choosing the element's name if that name will have an influence on node names (and ref attributes) and properties in the UI and/or data schemas, respectively. For instance, the name of the field generated by

```
<My_Diag t="gpsdiag"/>
```

is My_Diag. The generated UI schema will bear this user-specified name. However, writing

```
<gps/>
```

is similar to (but not exactly the same as) writing

```
<cols>
  <col>
    <Latitude/>
    <Northing/>
  </col>
  <col>
    <Longitude/>
    <Easting/>
  </col>
</cols>
<Take_From_GPS t="button"/>
```

in module.xml. The key difference is that the user does not specify the names of the inputs or button/trigger. (Another difference, aside from the main point, but important nonetheless, is that appropriate bindings will not be generated in the UI logic file.)