fbuild Documentation

Release 0.3

Erick Tryzelaar

Jun 12, 2018

Contents

THIS IS A WORK IN PROGRESS!

Contents:

CHAPTER 1

Manual

This is the Fbuild manual, which is aiming at being the authorative guide to using Fbuild in your projects. Note that this is still a work-in-progress!

1.1 Basics: Getting Fbuild, C Compilation, and Builders

1.1.1 Getting Started

This manual is for the most recent version of Fbuild from Git. In order to get it, just run:

```
$ git clone https://github.com/felix-lang/fbuild.git
$ cd fbuild
$ python3 setup.py install
```

You can also use Fbuild without installing it by running the `fbuild-light` script.

1.1.2 Compiling C

Let's start with a simple build script: it just builds a "Hello, world!" program written in C. Here's the C code:

```c
#include <stdio.h>

int main() {
    puts("Hello, world!");
    return 0;
}
```

and the Fbuild build script:

```python
from fbuild.builders.c import guess

def build(ctx):
    builder = guess.static(ctx)
    builder.build_exe('hello', ['hello.c'])
```

It's pretty simple. First, `guess` is imported. That object will let us create a new builder for building C programs (we'll get to builders in a moment). Then, the script defines a `build` function that takes an object of type `fbuild.context.Context` (`ctx`). That object is kind of like the "build engine." The next line calls `guess.static`, and the line after calls the `build_exe` method. It can take several keyword arguments, but the only two positional ones are the output file (`hello`) and a list of source files (`['hello.c']`). Let's run it:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc$ fbuild
determining platform     : {'linux', 'posix'}
looking for clang        : ok /home/ryan/stuff/Downloads/clang+llvm-3.8.0-x86_64-
↪linux-gnu-ubuntu-14.04/bin/clang
checking clang           : ok
looking for ar           : ok /usr/bin/ar
looking for ranlib       : ok /usr/bin/ranlib
checking if clang can make objects : ok
checking if clang can make libraries : ok
checking if clang can make exes      : ok
checking if clang can link lib to exe : ok
 * clang                             : hello.c -> build/obj/hello/hello.o
 * clang                             : build/obj/hello/hello.o -> build/hello
ryan@DevPC-LX:~/stuff/fbuild/playground/doc$
```

That just:

- Detected my C compiler and the associated utilities (`ar` and `ranlib`).
- Tested it.
- Built our program.

1.1.3 Builders

In Fbuild, a *builder* is an object that…builds stuff. In the last example, the builder could build C executables and libraries. Most of Fbuild revolves around builders, and all of them are located within `fbuild.builders`.

1.2 Rewind: The Core of Fbuild

1.2.1 Graphs and Caching

Let's take a step back for a moment. How does all this stuff even work??

Conventional build systems usually will function by first building a DAG (directed acyclic graph) consisting of all the inputs/outputs as nodes and the commands to create the outputs as edges. Although this allows some nice features (such as implicit parallelism and cool progress bars), it doesn't allow for some important things. In particular, if you've ever use Make, you know how painful it can be to read dependencies from a file. In addition, you can't decide a target's outputs at run time, and some DAG-based build systems can be either hard to extend, hard to use for simple projects, or brutally low-level.

Fbuild doesn't use a DAG. It doesn't even use a graph at all. Instead, it uses something much more powerful: caching. The core idea is that *all your build rules are really just functions*.

Any attempt to explain this any more would likely fail, so I'll just show an example. Put this in a new `fbuildroot.py`:

```python
import fbuild.db

@fbuild.db.caches
def myfunc(ctx, name):
    print('Hello, %s!' % name)

def build(ctx):
    myfunc(ctx, 'Fbuild world')
```

I'll explain `fbuild.db.caches` in a moment, but for now, note that any function that you use it on *must* take a context object as its first argument.

When the script is run, the output is what one would expect:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$ fbuild
Hello, Fbuild world!
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$
```

However, watch what happens if you run it again:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$ fbuild
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$
```

Nothing was shown! But why?

`fbuild.db.caches` will *cache* (or memoize, if you're more familiar with that term) the given function. That means that, when the function is called, Fbuild will save its arguments and the result into a database on disk (by default, it's located in `build/fbuild.db`). If the function is called again, then, instead of running it, Fbuild will just return the previous result. This is more obvious with a slightly different example:

```python
import fbuild.db

@fbuild.db.caches
def myfunc(ctx, name):
    print('Hello, %s!' % name)
    return 'myfunc was called'

def build(ctx):
    message = myfunc(ctx, 'Fbuild world')
    print(message)
```

If you run it, this happens:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$ fbuild
Hello, Fbuild world!
myfunc was called
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$
```

Note that the database didn't need to be deleted; Fbuild will automatically re-run a function if its contents have changed.

Watch what happens if you run it again:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$ fbuild
myfunc was called
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw$
```

When `myfunc` was called the first time, it's return value (`'myfunc was called'`) was saved into the database. On the second run, Fbuild saw that `myfunc` hadn't changed and was being called with the same arguments, so it just returned the original return value.

You may be wondering what this has to do with build systems. Well, in Fbuild, almost every internal function is cached like this. Remember `guess_static`? If you run that script again, the C compiler won't be re-configured. Fbuild cached the result of calling `guess_static` and loaded it back up from the database.

1.2.2 Dependencies

All this is really cool, but it doesn't seem that practical at the moment. Build systems don't just configure builders; they also. . . well, build stuff. Caching seems useless for solving this problem, right!

Wrong! Fbuild has several function annotations that you can use to help with this. Take a look at this build script:

```
import fbuild.db

@fbuild.db.caches
def build_a_file(ctx, src: fbuild.db.SRC):
    print('This is supposed to build the file %s...' % src)

def build(ctx):
    build_a_file(ctx, 'myfile')
```

I'll explain the details in a moment; for now, just know that `build_a_file` is supposed to do something with its input argument `myfile`. Let's run it:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ fbuild
Traceback (most recent call last):
  File "/media/ryan/stuff/anaconda/bin/fbuild", line 9, in <module>
    load_entry_point('fbuild==0.2', 'console_scripts', 'fbuild')()
  File "/media/ryan/stuff/fbuild/lib/fbuild/main.py", line 179, in main
    result = build(ctx)
  File "/media/ryan/stuff/fbuild/lib/fbuild/main.py", line 104, in build
    target.function(ctx)
  File "/media/ryan/stuff/fbuild/playground/doc-rw-dep/fbuildroot.py", line 8, in␣
↪build
    build_a_file(ctx, 'myfile')
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/__init__.py", line 121, in __call__
    result, srcs, dsts = self.call(*args, **kwargs)
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/__init__.py", line 125, in call
    return ctx.db.call(self.function, ctx, *args, **kwargs)
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/database.py", line 101, in call
    dsts)
  File "/media/ryan/stuff/fbuild/lib/fbuild/rpc.py", line 68, in call
    raise result.result
  File "/media/ryan/stuff/fbuild/lib/fbuild/rpc.py", line 112, in _process
    result.result = self._handler(*args, **kwargs)
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/database.py", line 24, in handle_rpc
    return method(*args, **kwargs)
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/backend.py", line 42, in prepare
    call_file_digests = self.check_call_files(call_id, srcs)
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/backend.py", line 143, in check_call_
↪files
    d, file_id, file_digest = self.check_call_file(call_id, file_name)
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/backend.py", line 165, in check_call_
↪file
```

(continues on next page)

(continued from previous page)

```
    dirty, file_id, mtime, digest = self.add_file(file_name)
  File "/media/ryan/stuff/fbuild/lib/fbuild/db/backend.py", line 249, in add_file
    file_mtime = file_path.getmtime()
  File "/media/ryan/stuff/fbuild/lib/fbuild/path.py", line 224, in getmtime
    return os.path.getmtime(self)
  File "/media/ryan/stuff/anaconda/lib/python3.4/genericpath.py", line 55, in getmtime
    return os.stat(filename).st_mtime
FileNotFoundError: [Errno 2] No such file or directory: Path('myfile')
```

Whoops! I forgot to create `myfile`:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ touch myfile
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ fbuild
This is supposed to build the file myfile...
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$
```

As usual, let's also run it again:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ fbuild
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$
```

Nothing happened! This is caching at work again.

Now try adding something to `myfile` and running it again:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ echo 1234 > myfile
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ fbuild
This is supposed to build the file myfile...
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$
```

`build_a_file` is run again! Look back at these two lines in `fbuildroot.py`:

```python
@fbuild.db.caches
def build_a_file(ctx, src: fbuild.db.SRC):
```

I already explained how `fbuild.db.caches` works. However, the new addition is the function annotation `fbuild.db.SRC`. This works with `fbuild.db.caches` to allow for dependency resolution.

When you annotate a function argument with `fbuild.db.SRC`, you're telling `fbuild.db.caches` that the argument is a source file. As already stated, if you change `build_a_file` or change any of its arguments, it will be re-run. In addition, *if you change the contents of any source file, the function will also be re-run.* Because I changed the contents of `myfile`, Fbuild re-ran `build_a_file`.

Remember `build_exe`? This is how it works. Although the function itself is somewhat complex, at it's core, it uses a similar method to this.

You can also create functions that take multiple sources:

```python
import fbuild.db

@fbuild.db.caches
def build_a_file(ctx, first_source: fbuild.db.SRC, other_sources: fbuild.db.SRCS):
    print('Do something with %s and %s...' % (first_source, other_sources))

def build(ctx):
    build_a_file(ctx, 'myfile1', ['myfile2', 'myfile3'])
```

As you might expect by now, `fbuild.db.SRCS` takes a list of source files, not just one.

Nevertheless, this is only part of the equation. A build system usually needs to also keep track of its output files. Unlike other example scripts, this is actually not just a toy; it's actually a quite useful function:

```python
import fbuild.db, shutil, io

@fbuild.db.caches
def merge_files(ctx, srcs: fbuild.db.SRCS, dst: fbuild.db.DST):
    print('Merging files...')

    result = io.StringIO()
    for src in srcs:
        with open(src) as f:
            shutil.copyfileobj(f, result)

    result.seek(0)
    with open(dst, 'w') as f:
        shutil.copyfileobj(result, f)

def build(ctx):
    merge_files(ctx, ['input1', 'input2'], 'output')
```

The details of `merge_files` don't really matter as much as the function annotations. Note that another annotation was added: `fbuild.db.DST`, which annotates the destination parameter. The results of running it are like you'd expect:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ echo 1 > input1
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ echo 2 > input2
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ fbuild
Merging files...
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$ cat output
1
2
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-dep$
```

As before, any changes to `input1` or `input2` will cause `output` to be re-built.

This isn't quite enough, however, but before I go to the next topic, there's one more basic thing that needs to be covered: paths.

1.2.3 Path Objects

Remember the error message when I forgot to create `myfile`? It mentioned that the missing file was `Path('myfile')`. The `Path` here is for Fbuild's *path objects*. I won't go over every single detail, but path objects (defined in `fbuild.path`) are... well, path objects. The class `fbuild.path.Path` is a subclass of `str`, so it supports all the normal operations of `str`, and you can pass it to any normal Python function expecting a string, However, path objects also have a bunch of methods useful for file system/path manipulation.

For thorough documentation on all the methods, check out **'lib/fbuild/path.py < https://github.com/felix-lang/fbuild/blob/master/lib/fbuild/path.py>'_** in the source code. Here I'll mention just one capability of paths: in order to join them, you can use `/`. For instance, `Path('src') / 'dst'` returns `Path('src/dst')` on Posix and `Path('src\\dst')` on Windows.

1.2.4 Rule Destinations and Cached Objects

Back on topic: recall the very first Fbuild script in the tutorial:

```python
from fbuild.builders.c import guess

def build(ctx):
    builder = guess.static(ctx)
    builder.build_exe('hello', ['hello.c'])
```

See `builder.build_exe`? That function actually returns a value: the full path to the resulting executable. The reason is that, usually, the developer doesn't care where the executable is stored or what extension it has, but they may very well want to know where it's located. To handle this case, Fbuild supports annotating the function's *return value* as a destination. For example:

```python
from fbuild.path import Path
import fbuild.db, shutil

@fbuild.db.caches
def do_something(ctx, src: fbuild.db.SRC) -> fbuild.db.DST:
    src = Path(src)
    dst = ctx.buildroot / src.replaceext('.out')
    print('Copying %s to %s...' % (src, dst))
    src.copy(dst)
    return dst

def build(ctx):
    do_something(ctx, 'x.in')
```

Let's run it:

```
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-out$ echo 123 > x.in
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-out$ fbuild
Copying x.in to build/x.out...
ryan@DevPC-LX:~/stuff/fbuild/playground/doc-rw-out$
```

This script has a lot of new stuff! It uses the `Path` objects mentioned in the previous section. In particular:

- This is the first example script to use `ctx.buildroot`, which is a `Path` that points to the output directory. In this case, it's `build`.
- `Path.replaceext` replaces the given file extension, e.g. `Path('x.in').replaceext('.out')` results in `Path('x.out')`.
- `Path.copy` copies the given file. `` Path(src).copy(dst)` `` is equivalent to `shutil.copy(src, dst)`.
- **Most importantly,** `do_something` returns the resulting output file. This will cause Fbuild to place it in the database.

The entirety of Fbuild, including the C builder that I first showed, consists of what I've just shown here, with three exceptions:

1. In `fbuild.db`, there's a very important class:

   `fbuild.db.PersistentObject`. If you want to contain any cached functions within a class, the class must derive from `PersistentObject`, and the cached functions should instead use `cachemethod` (see below). Note that the default constructor for objects derived from `PersistentObject` takes a context object as its argument. If you define a custom `__init__`, you need to take a context object and assign it to `self.ctx`. Example:

   ```python
   class MyObject(fbuild.db.PersistentObject):
       def __init__(self, ctx):
   ```

(continues on next page)

(continued from previous page)

```
        self.ctx = ctx

def build(ctx):
    obj = MyObject(ctx)
```

2. `fbuild.db.cachemethod` is equivalent to `fbuild.db.caches`, but it is

   instead designed to annotate methods that are in a subclass of `PersistentObject`. In addition, methods annotated with `cachemethod` don't need to be passed a context argument. Example:

```
class MyObject(fbuild.db.PersistentObject):
    def __init__(self, ctx):
        self.ctx = ctx

    @fbuild.db.cachemethod
    def myfunc(self, msg):
        print('Message:', msg)

def build(ctx):
    obj = MyObject(ctx)
    obj.myfunc('Hello, world!')
```

3. Sometimes, you may not want to return a whole object. For this case, Fbuild

   provides `fbuild.record.Record`. A `Record` is basically a `dict`, except that you can also set and get keys via attributes. For example, `my_record.a` is equivalent to `my_record['a']`.

Many examples of this are in the Fbuild source.

1.3 Back to a Higher Level: Logging and Running External Commands

1.3.1 Logging

Of course, a build system is mostly useless without being able to run external commands. First, I need to mention an important concept of Fbuild that I've glossed over thus far: logging.

Notice that, in all the above examples, `print` was used to print information. Technically, you're not supposed to do this! In order to handle this, Fbuild provides `ctx.logger`. Here's a basic example:

```
def build(ctx):
    ctx.logger.log('This will be written to the log file: build/fbuild.log.',
                   verbose=1)
    ctx.logger.log('This will be written to the console.')

    ctx.logger.log('This will be written to the console in red.', color='red')
    ctx.logger.log('This will be written to the console in a color designated for '
                   'compiling files.', color='compile')
    ctx.logger.log('And for linking files!', color='link')

    ctx.logger.check('this is used when configuring various things in Fbuild')
    ctx.logger.passed()

    ctx.logger.check('you can also give custom messages and colors', color='blue')
```

(continues on next page)

(continued from previous page)

```
    ctx.logger.passed('it worked!')

    ctx.logger.check('things can also fail')
    ctx.logger.failed('dang it!')
```

and here's the output:

1.3.2 Executing Shell Commands

Now comes executing shell commands! Every context object has a method for this: `execute`. Here's the definition from the source code:

```
def execute(self, cmd, msg1=None, msg2=None, *,
        color=None,
        quieter=0,
        stdout_quieter=None,
        stderr_quieter=None,
        input=None,
        stdin=None,
        stdout=fbuild.subprocess.PIPE,
        stderr=fbuild.subprocess.PIPE,
        timeout=None,
        env=None,
        runtime_libpaths=None,
        ignore_error=False,
        **kwargs):
```

That's a lot of arguments! I'll break them down one by one:

- `cmd` is the command to run. Although there are some edge cases, in general, this should be a list, such as `['clang', '-o', 'x', 'x.c']`.
- `msg1`, `msg2`, and `color` will be explained in the example below.
- `quieter` is the same as the `quieter` argument with `logger.log`; it determines whether or not `msg1` and `msg2` will be displayed or just sent to the log file. In addition, this will be the default value fo `stdout_quieter` and `stderr_quieter` if they are `None`.
- `stdout_quieter` and `stderr_quieter` are the same thing as `quieter`, except they are for whether or not the output of the command will be shown.
- `input` is a byte string to be sent to the command's standard input.
- `stdin` is ignored if `input` is truthy; otherwise, it will be the `stdin` argument passed to `subprocess.Popen`.
- `stdout` and `stderr` are passed to `subprocess.Popen`.
- `timeout` is the maximum number of seconds to wait for the command to finish before killing it. If you pass a falsy value, it will never kill the command.
- `env` is a dictionary of environment variables to pass to the function; if `None`, then the current environment in `os.environ` will be passed.
- `runtime_libpaths` is a list of strings to be added to the platform's DLL/ shared library search path.
- `ignore_error` will determine whether or not an `fbuild.ExecutionError` is thrown if the command fails.
- `kwargs` is just passed on to `subprocess.Popen`.

In addition, it will return a tuple `(stdout, stderr)`, where both `stdout` and `stderr` are byte strings.

That's a lot to take in at once, so here are some examples of using `execute`:

```
def build(ctx):
    ctx.execute(['echo', '123']) # Run `echo 123` and print the output.

    # Run echo, but print the message "running echo" first, with NO NEWLINE.
    ctx.execute(['echo', 'Echoed text here!'], msg1='running echo')
    # Run echo, but print the message "running echo: 123" first.
    ctx.execute(['echo', 'Echoed text here!'], msg1='running echo', msg2='123')
    # This is the color of `msg2`.
    ctx.execute(['echo', '123'], msg1='running echo', msg2='123', color='compile')
    # This would normally be an error, but ignore_error is True.
    ctx.execute(['printf'], ignore_error=True)

    ctx.execute(['echo', '123'], stdout_quieter=1) # Don't print the output.
    # msg1 and msg2 will still be printed.
    ctx.execute(['echo', '123'], msg1='running echo', msg2='123', stdout_quieter=1)

    ctx.execute(['cat'], input=b'Input here!') # Send some text to stdin.

    # This will throw an error because running `sleep` timed out.
    ctx.execute(['sleep', '1'], msg1='running sleep', msg2='1', timeout=0.5)
```

And the output:

Note that `execute` is *not* cached!

1.4 Configuration: Locating Programs and Defining Command-line Options

1.4.1 Finding Programs

A frequently needed capability of a build system is to locate a program. For instance, you may want to find the `awk` executable on the system. For this, Fbuild has `fbuild.builders.find_program`. It works like this:

```
from fbuild.builders import find_program

def build(ctx):
    awk = find_program(ctx, ['awk', 'gawk'])
    print(awk)
```

It takes two arguments: the context object and a list of programs to search for. The return value is the first program it found. If none are found, it will throw an exception of type `fbuild.ConfigFailed`.

In addition, `find_program` is cached, so it won't re-run every single time you run Fbuild.

1.4.2 Command-line Options

Other common build system feature is the ability for the user to pass information to the build system in some way, usually either environment variables or command-line arguments. Unfortunately, both are often implemented in odd ways, like weird execution environments (Make and CMake are the worst offenders here) or even defining arguments in a totally different file.

Fbuild takes the easy route: since your build rules are in one function, your arguments can be defined in another. Instead of using some weird, arcane, home-grown, undocumented module for this, Fbuild uses the **‘argparse module < https://docs.python.org/3/library/argparse.html>‘_**.

Here’s a simple example of command-line arguments in Fbuild:

```
def arguments(parser):
    group = parser.add_argument_group('config options')
    group.add_argument('--build-mode', help='Define the desired build mode',
                       choices=['debug', 'release'])
    group.add_argument('--mini', help='Attempt to minify the built code',
                       action='store_true', dest='minify')
    group.add_argument('--arch', help='The target build architecture', default='x64')
    group.add_options((
        make_option('--build-mode', help='Define the desired build mode',
                    choices=['debug', 'release']),
        make_option('--mini', help='Attempt to minify the built code',
                    action='store_true', dest='minify'),
        make_option('--arch', help='The target build architecture', default='x64'),
    ))

def build(ctx):
    print('Build mode:', ctx.options.build_mode)
    print('Minify?', ctx.options.minify)
    print('Arch:', ctx.options.arch)
```

The example is mostly self-explanatory; if you have any questions, consult the argparse documentation.

1.5 Advanced Core Topics: Adding External Dependencies and Installing Files

1.5.1 External Dependencies

Let’s say you’re creating your own programming language called Qux. When you run it, it looks kind of like this:

```
ryan@DevPC-LX:~$ qux myfile.qux myfile.out
Qux version 0.0.0
Building myfile.qux...
NOTE: myfile.qux imports myotherfile.qux!
Building myotherfile.qux...
Successfully built myfile.out!
ryan@DevPC-LX:~$
```

Take this simple rule for building Qux programs:

```
from fbuild.builders import find_program
from fbuild.path import Path
import fbuild.db

class QuxBuilder(fbuild.db.PersistentObject):
    def __init__(self, ctx):
        self.ctx = ctx
        self.qux = find_program(ctx, ['qux'])

    @fbuild.db.cachemethod
```

(continues on next page)

(continued from previous page)

```
    def build(self, src: fbuild.db.SRC) -> fbuild.db.DST:
        dst = self.ctx.buildroot / Path(src).replaceext('.out')
        self.ctx.execute([self.qux, src, dst])
        return dst

def build(ctx):
    qux = QuxBuilder(ctx)
    qux.build('myfile.qux')
```

This *sort of* works. Remember, `myfile.qux` depends on `myotherfile.qux`, but Fbuild doesn't know that. Therefore, if you edit `myotherfile.qux`, `myfile.out` won't get rebuilt.

For this purpose, Fbuild has `ctx.db.add_external_dependencies_to_call`:

```
from fbuild.builders import find_program
from fbuild.path import Path
import fbuild.db, re

class QuxBuilder(fbuild.db.PersistentObject):
    def __init__(self, ctx):
        self.ctx = ctx
        self.qux = find_program(ctx, ['qux'])

    @fbuild.db.cachemethod
    def build(self, src: fbuild.db.SRC) -> fbuild.db.DST:
        dst = self.ctx.buildroot / Path(src).replaceext('.out')

        stdout, stderr = self.ctx.execute([self.qux, src, dst])
        regex = re.compile(r'Building (.*)...$')
        for line in stdout.decode('ascii').splitlines():
            m = regex.match(line)
            if m:
                self.ctx.db.add_external_dependencies_to_call(srcs=[m.group(1)])

        return dst

def build(ctx):
    qux = QuxBuilder(ctx)
    qux.build('myfile.qux')
```

This is quite a bit more complex than the last example! If you're unfamiliar with Python's re module, this may look confusing. All the regex is going is locating all the files that Qux is building. The important part is the call to `self.ctx.db.add_external_dependencies_to_call`, which takes two keyword arguments: `srcs` and `dsts`. These add extra dependencies/outputs to the build rule *while it's still executing*. Now, if you edit `myotherfile.qux`, then `myfile.out` *will* be rebuild!

1.5.2 Installing Files

When your application is built, you probably want some way to install it onto the user's system. Fbuild has this covered with `ctx.install`. It's defined like this:

```
def install(self, path, target, *, rename=None, perms=None):
```

The `path` is the path to install of the file to install, and `target` is a subdirectory of the installation prefix to install into. For instance, `ctx.install('somehwere/my-file', 'share/my-app')` will copy `somewhere/my-file` to `$TARGET/share/my-app/my-file`.

The installation target (shown above as `$TARGET`) is defined as `ctx.install_destdir / ctx.install_prefix`. `install_destdir` is `/` by default, and `install_prefix` is `/usr`. (This separation is done to ensure that packaging apps using Fbuild for Unix systems can be done in an autoconf-compatible way.) Both of these can be set inside your application; for example, you can assign them in `build` using values passed on the command line:

```python
def arguments(parser):
    group = parser.add_argument_group('config options')
    group.add_argument('--destdir', help='Set the installation destdir', default='/')
    group.add_argument('--prefix', help='Set the installation prefix', default='usr')


def build(ctx):
    ctx.install_destdir = ctx.options.destdir
    ctx.install_prefix = ctx.options.prefix

    # Continue as normal.
```

If you want to change the file name (e.g. `my-new-file` instead), you can pass that as the `rename` parameter, e.g. `ctx.install('somewhere/my-file', 'share/my-app', 'my-new-file')`.

`perms` can be used to assign custom permissions to the target file. By default, it will use the same permissions as the original file.

By default, nothing is installed yet. `ctx.install` just *marks* the file for installation. Later on, when the user runs `fbuild install`, Fbuild will run the build script, then install any files that were marked for installation.

Bonus points: if you use a Linux system that has polkit available (which is basically most modern Linux distros), you'll never need to prefix your install commands with `sudo`. Fbuild will automatically use polkit to ask for escalated privileges to allow installation.

One more note: Fbuild provides two hooks, *pre_install* and *post_install*, that can be used to have code run before and after installation:

```python
def pre_install(ctx):
    print('Going to install...')

def post_install(ctx):
    print('Done installing!')
```

1.5.3 Platforms

Within `fbuild.builders` is a module that you will likely rarely use, but it is important to the implementation of various builders: `platform`. It defines a couple of interesting functions, but the most important one is `guess_platform`.

`guess_platform` returns a set of strings that aims at identifying the platform that Fbuild is running on. The following are possible members of the set

- `posix`: This is a Posix system.
- `linux`: This is a Linux system.
- `solaris`: This is a Solaris system.
- `sunos`: This is a Sun Solaris system.
- `cygwin`: This is a Windows system, but Fbuild is being run within Cygwin.
- `mingw`: This is a Windows system, but Fbuild is being run within MinGW.

- `windows`: This is a Windows system.
- `win32`: This is a 32-bit Windows system.
- `win64`: This is a 64-bit Windows system.
- `bsd`: This is a BSD system.
- `freebsd`: This is a FreeBSD system.
- `openbsd`: This is a OpenBSD system.
- `netbsd`: This is a NetBSD system.
- `darwin`, `osx`: This is a Mac OS X system.

Some examples of sets returned by `guess_platform` would include `{'windows', 'win32'}` and `{'posix', 'linux'}`.

1.6 Building C/C++ Files

A major portion of any build system usually revolves around C and C++. As a result, Fbuild provides a large amount of tools to help with this. Note that this section of the manual is *not* aimed at documenting every single feature, but it focuses on the important parts.

1.6.1 Builders, and the `guess` api

Let's take a look at this example:

```
from fbuild.builders.c import guess

def build(ctx):
    builder = guess.static(ctx)
    lib = builder.build_lib('mylib', ['mylib.c'])
    builder.build_exe('hello', ['hello.c'], libs=[lib])
```

Now that you know how builders are defined, I can start convering building C code. The magic here is all in the function `guess.static`. The `guess` object has two members:

- *guess.static*: Returns a builder that can be used to build normal executables via `build_exe` and static libraries via `build_lib`.
- *guess.shared*: Returns a builder that can be used to build position-independent executables (`-fPIC`) via `build_exe` and and shared libraries via `build_lib`.

In this example, `guess.static` is used, so it returns a builder for building "normal" executables and static libraries.

If you want both a static *and* a shared builder, you can use `guess(...)`:

```
from fbuild.builders.c import guess

def build(ctx);
    builders = guess(ctx)

    static = builders.static.build_lib('mylib-static', ['mylib.c'])
    shared = builders.shared.build_lib('mylib-shared', ['mylib.c'])
```

TODO

1.7 TODO

Document `fbuild.config` and more C/C++ compilation stuff.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search