HELIX

Release 1.2.0

MIT Lincoln Laboratory

Jan 18, 2022

CONTENTS:

CHAPTER

ONE

GETTING STARTED

1.1 HELIX at a Glance

The problem of malware similarity (and, more broadly, software similarity) is very difficult to assess. Measuring relative performance of malware similarity solutions is very difficult without a large dataset of malware and high-quality ground truth about the software similarity among samples of the dataset and acquiring this ground truth for malware in the wild is nearly impossible at scale. Enter HELIX.

HELIX is a source code generation, mutation, and transformation framework primarily geared toward generating large, synthetic datasets of functional malware with known, measurable software similarity. HELIX primarily consists of three main primitives:

Blueprints Core project layouts including templated boilerplate and methods for generating and building artifacts from a set of Components and Transforms. For example, a C++ project build with CMake.

Components Small, configurable pieces of source code that represent a specific implementation of a specific functionality along with associated metadata. For example, a specific implementation of downloading a file from a given URL using the cURL library.

Transforms Modifications of either source code or a built artifact along with associated metadata. For example, the Linux binutil `strip` which removes debugging symbols from a compiled binary.

A HELIX build is made up of exactly one Blueprint, zero or more configured Components, and zero or more configured Transforms. The result of a HELIX build is one or more artifacts (for example, a compiled binary build from all of the Components and transformed by all of the Transforms) and a collection of metadata, aggregated from the included Blueprints, Components, and Transforms.

While developing individual Components and Transforms for HELIX is initially time-consuming, the effort scales well, as the more Components and Transforms are written for HELIX the larger dataset it is capable of generating.

1.2 Quick Install Guide

1.2.1 Prerequisites

HELIX is supported by Python (>=3.5) on both Windows and Linux and is installed with pip. Both Python and pip must be installed on your system before attempting to install HELIX.

1.2.2 Installation

To install HELIX from PyPI with pip, run:

```
pip install helix
```

1.2.3 Additional Dependencies

Some HELIX Blueprints, Components, and Transforms include additional, external dependencies which must be installed before they can be used in HELIX builds. To install all of these dependencies, use the `install` HELIX CLI command after installing HELIX with `pip`:

```
helix install
```

Note: Depending on your platform, the above command may require root/Administrator priveleges. Some dependencies may also need to be manually installed - these will be listed in the output of the above command.

1.2.4 Development

To set up a development environment, first clone the HELIX repo. Next, install additional, optional extensions for development and testing by running (from the root of the repo):

```
pip install .[development,testing]
```

1.3 Using the HELIX CLI

Once HELIX is installed, you can use the HELIX CLI by simply running:

```
helix <command>
```

You can list all of the installed Blueprints, Components, and Transforms with:

```
helix list
```

After installing the required dependencies (see *Additional Dependencies*) for Blueprints, Components, and Transforms, you can generate HELIX builds with the `build` command. For example, to build the `cmake-cpp` Blueprint with the `configuration-example` Component and the `strip` Transform and write the output to `./example`, run:

```
helix build blueprint cmake-cpp ./example \
    -c configuration-example:second_word=foo \
    -t strip
```

This should output a message listing the relevant metadata tags and the built artifacts (in this case, a single, UPX-packed binary that simply prints "hello foo").

Note: For more detail on the `build` command and additional examples, see *Building*.

1.4 Writing Your First Component

1.4.1 Building a Python Package

HELIX makes use of Python entrypoints to discover installed Blueprints, Components, and Transforms. Additional Blueprints, Components, and Transforms can be installed by bundling them in a Python package with an entrypoint in one of the following groups:

- `helix.blueprints`
- `helix.components`
- `helix.transforms`

The name of the entrypoint should correspond with the name of the Blueprint, Component, or Transform, and the object reference should refer to the class of its implementation.

To start, create a basic python package named `helix-example` by creating the following directory structure:

```
.
├── helix_example/
│   ├── __init__.py
└── setup.py
```

The `__init__.py` file should be blank and `setup.py` should consist of the following:

```
# setup.py

from setuptools import setup
from setuptools import find_packages

setup(
    name="helix-example",
    version="1.0.0",
    author="Your Name Here",
    author_email="you@your-domain",
    description="An example external HELIX package",
    url="http://your-domain",
    packages=find_packages(),
    python_requires=">=3.5",
    install_requires=[],
    include_package_data=True,
    zip_safe=False,
```

(continues on next page)

(continued from previous page)

```
    entry_points={
        "helix.blueprints": [],
        "helix.components": [],
        "helix.transforms": [],
        "helix.tests": []
    },
)
```

This is the basic layout of a Python package - in later sections, we will create Components and Transforms and register them as entrypoints. You can install the package with:

```
pip install .
```

Note: For ease of development, it can be useful to install the Python package in editable mode to avoid having to reinstall the package every time you make changes. You can do this by instead running:

```
pip install -e .
```

1.4.2 Writing the Component

Components are simply Python classes that implement the *Component* interface. To write a simple component, all you need to do is subclass this base class and implement the required abstract methods.

Let's start by adding a `components` directory to our Python package with an `example` module for our new Component. The package directory structure should look like the following:

```
.
├── helix_example/
│   ├── components/
│   │   ├── example.py
│   │   ├── __init__.py
│   ├── __init__.py
└── setup.py
```

Inside of `example.py` we'll create a simple Component by subclassing *Component*:

```
# example.py

from helix import component


class ExampleComponent(component.Component):
    """A simple example component."""

    name = "example-component"
    verbose_name = "Example Component"
    type = "example"
    version = "1.0.0"
    description = "A simple example component"
    date = "2020-10-20 12:00:00.000000"
```

(continues on next page)

(continued from previous page)

```
    tags = (("group", "example"),)

    blueprints = ["cmake-c", "cmake-cpp"]

    functions = [r"""
        #include <stdio.h>

        void ${hello_world}() {
            printf("hello world\n");
        }
    """]
    calls = {
        "main": [
            r'${hello_world}();'
        ]
    }
    globals = ["hello_world"]
```

We start by defining required metadata (*name*, *verbose_name*, *type*, etc.). Next, we need to define which Blueprints this Component is designed to work with - since we're writing code that could be compiled as either C or C++ code, we support both `CMakeCBlueprint` and `CMakeCBlueprint` by name. Next, we define a simple function `hello_world` that simply prints `"hello world"` by adding it to the the *functions* list for the Component. Note that the function name is surrounded in template parameters (`${...}`). These template parameters tell the build system how to finalize Components so that duplicate function names do create conflicts. Any template parameters like these that need to be deduplicated by the build system should be included in the *globals* property.

Finally, we'll add a single call at the `main` callsite (defined by the `cmake` Blueprints - see `helix.blueprints.CMakeCBlueprint.CALLSITE_MAIN`) which calls our `hello_world` function. *callsites* are defined by each individual Blueprint and provide a way for Components to invoke their functions. The `cmake` Blueprints' `main` callsite, as the name suggests, allows Components to call functions inside of the generated binary's `main` function. We can make use of this callsite by adding it to the *calls* property for the Component.

Note: Because the `printf` function is a part of the `stdio` library, we have to add an include that references it. We can simply add this to our function definition.

Our Component definition is now complete.

1.4.3 Registering the Component

To register the component so that HELIX can find it, we need to add an entrypoint in the group `helix.components` to our Python package's `setup.py`. Make the following change to `setup.py`:

```
# setup.py

...
entry_points={
    ...
    "helix.components": [
        "example-component = helix_example.components.example:ExampleComponent"
    ]
    ...
```

(continues on next page)

(continued from previous page)

```
}
...
```

Note: The `name` property of our new Component *must* match the name of the entrypoint.

To update the entrypoint list, reinstall the Python package (even if you installed it in editable mode):

```
pip install .
```

Check that our new Component is registered with the HELIX CLI:

```
helix list
```

The output should include our new example Component:

```
Available Components:
  ...
  Example Component (1.0.0) [example-component]
  ...
```

Finally, build a `cmake-cpp` Blueprint with our Component to make sure that it works:

```
helix build blueprint cmake-cpp ./example -c example-component
```

Run the generated artifact binary - it should simply print "hello world" and exit.

Note: While developing a new component, it can be useful to build in verbose mode (`-v/--verbose`) to see the full output of the build commands to assist in debugging.

1.4.4 Adding Configuration Options

Configuration options may be specified for Components in the *`options`* property. Make the following changes to the `ExampleComponent` class to define an optional configuration parameter `message` which will be printed to the console:

```
# example.py

from helix import utils

...

class ExampleComponent(component.Component):
    ...
    options = {"message": {"default": "hello world"}}
    ...

    # The following lines may be removed:

    # functions = [r"""
    #     #include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
    #     void ${hello_world}() {
    #         printf("hello world\n");
    #     }
    # """]
    # calls = {
    #     "main": [
    #         r'${hello_world}();'
    #     ]
    # }
    # globals = ["hello_world"]

    TEMPLATE = r"""
        #include <stdio.h>

        void ${hello_world}() {
            printf("${message}\n");
        }
    """

    def generate(self):
        function = utils.substitute(self.TEMPLATE, message=self.configuration["message"])

        self.functions = [function]
        self.calls = {
            "main": [
                r'${hello_world}();'
            ]
        }
        self.globals = ["hello_world"]
```

Components can choose to define their *functions*, *calls*, and *globals* properties inside of a *generate* method. This method is run after configuration parameters are parsed and these parameters are available in the *configuration* property as a dict and can be used in the *generate* method as above.

Reinstall the Python package (if not installed in editable mode) and then create a new HELIX build, supplying the new configuration parameter:

```
helix build blueprint cmake-cpp ./example -c example-component:message="goodbye world"
```

Run the generated artifact binary - it should now print "goodbye world" and exit.

1.4.5 Using External Template Files

Once a Component becomes relatively complex, it can be a good idea to move the templated function code belonging to the Component into its own file so that it is easier to track changes and so that syntax highlighting can be enabled for ease of development. HELIX includes a couple of utilities to help you do that. In this section, we'll move the source code for our `ExampleComponent` to an external `example.c` file.

To start, we'll need to configure our Python package so that it includes non-python files when it is compressed into its distributable form. To do this, add a file named `MANIFEST.in` to the root of your python package with the following contents:

```
# MANIFEST.in

recursive-include helix_example *.c
```

This tells the Python package manager that any files with the extension `.c` should be included with the package.

Next, write create a file in the same directory as `example.py` called `example.c`. The package directory structure should look like:

```
.
├── helix_example/
│   ├── components/
│   │   ├── example.py
│   │   ├── example.c
│   │   ├── __init__.py
│   ├── __init__.py
└── setup.py
```

Add the following content to `example.c`:

```
// example.c

#include <stdio.h>

void ${example}() {
    printf("${message}\n");
}
```

Finally, modify the `ExampleComponent` class in `example.py` as follows:

```
# example.py

class ExampleComponent(component.Component):
    ...

    # The following lines may be removed:

    # TEMPLATE = r"""
    #     #include <stdio.h>
    #
    #     void ${hello_world}() {
    #         printf("${message}\n");
    #     }
    # """

    def generate(self):
        ...

        template = utils.source(__name__, "example.c")

        ...

        function = utils.substitute(template, message=formatted)
```

(continues on next page)

(continued from previous page)

```
        ...
```

We make use of the *source* function here to fetch the source of the included template file, relative to the current package path.

You can now reinstall the package (if not installed in editable mode) and test these Component changes. The Component should function exactly the same, but the Python package is now a bit more maintainable.

1.4.6 Adding Dependencies

HELIX includes a dependency installation/management system for Blueprints, Components, and Transforms for managing external dependencies that cannot be installed with `pip`. Lets add a simple `apt` dependency to our Component - `cowsay` to improve the visual output of our printed message.

Note: From here on, this tutorial only works on a Linux platform. There are dependency types defined for Windows, however, and you can find examples of their use in HELIX source.

Add the following to the `ExampleComponent` class:

```
# example.py

from helix import utils

...

class ExampleComponent(component.Component):
    ...

    dependencies = [utils.LinuxAPTDependency("cowsay")]

    ...

    def generate(self):
        ...

        cowsay = utils.find("cowsay")
        output, _ = utils.run(
            "{} {}".format(cowsay, self.configuration["message"]), cwd="./"
        )
        formatted = repr(output.decode("utf-8")).replace("'", "")

        ...

        function = utils.substitute(self.TEMPLATE, message=formatted)
```

Reinstall the Python package (if not in editable mode) and install dependencies for our Component:

```
helix dependencies component example-component
```

Note: You may need to run the above command as root/Administrator to successfully install dependencies.

Finally, build the `cmake-cpp` Blueprint again with our updated Component:

```
helix build blueprint cmake-cpp ./example -c example-component
```

You should now get an output similar to the following when running the generated artifact binary:

```
 _____________
< hello world >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note: It's worth noting here that the binary generated by HELIX in this example does not actually make use of `cowsay`. Instead, `cowsay` is invoked during configuration of the Component, and the `cowsay` string is injected into the generated source code. A more advanced approach, left as an exercise for the reader, would be to invoke `cowsay` from the generated artifact instead (e.g., with a Linux `system` call written in C/C++).

1.4.7 Testing the Component

HELIX includes some minimal utilities for testing Components with the `unittest` framework. To write a unit test for our Component, add the following to the `example.py` module:

```
# example.py

from helix import tests

...

class ExampleComponentTests(tests.UnitTestCase, tests.ComponentTestCaseMixin):
    blueprint = "cmake-cpp"
    component = "example-component"
```

This will create a couple of simple unit tests from *TestCaseMixin*.

Note: When developing Components, at a minimum it is recommended to define the simple testing class above. This will introduce simple build tests as well as a test that ensures that your Component's templated globals are configured correctly (for more details, see *TestCaseMixin*).

To register this unit test with HELIX, add an entrypoint to the `helix.tests` group in the Python package's `setup.py` as follows:

```
# setup.py

...
entry_points={
    ...
    "helix.tests": {
```

(continues on next page)

(continued from previous page)

```
        "example-component = helix_example.components.example:ExampleComponentTests"
    }
    ...
}
...
```

Finally, to run unit tests for Blueprints, Components, and Transforms, run:

```
helix test unit
```

1.5 Writing Your First Transform

Prior to following this tutorial, you'll need to have a python package set up to extend HELIX (see *Building a Python Package*).

1.5.1 Writing the Transform

Similar to Components, Transforms are simply Python classes that implement the *Transform* interface.

Let's start by adding a `transforms` directory to our Python package with an `example` module for our new Transform. The package directory structure should look like the following:

```
.
├── helix_example/
│   ├── transforms/
│   │   ├── example.py
│   │   ├── __init__.py
│   ├── __init__.py
└── setup.py
```

Inside of `example.py` we'll create a simple Transform by subclassing *Transform*:

```
# example.py

import os
import shutil
import base64

from helix import transform


class ExampleTransform(transform.Transform):
    """A simple example transform."""

    name = "example-transform"
    verbose_name = "Example Transform"
    type = transform.Transform.TYPE_ARTIFACT
    version = "1.0.0"
    description = "A simple example transform"
    tags = (("group", "example"),)
```

(continues on next page)

(continued from previous page)

```
    def transform(self, source, destination):
        """Print the contents of the binary.

        This transform doesn't actually do anything, it is just a simple
        example that prints the contents of the input file, base64 encoded.
        """

        source = os.path.abspath(source)
        destination = os.path.abspath(destination)

        with open(source, "rb") as f:
            print(base64.b64encode(f.read()))

        shutil.copy(source, destination)
```

This simple transform just prints the base64 encoding of the built artifact and does not modify it at all. There are a couple of things to note:

1. The transform *type* is *TYPE_ARTIFACT* - this indicates that the transform should be applied to artifacts after the Blueprint is built, as opposed to *TYPE_SOURCE* which is applied to source files before they are built.

2. The required *transform* method must *always* write a resulting file from `source` to `destination`, even if it does not modify the contents.

1.5.2 Registering the Transform

Similar to Components, Transforms must be added to the entrypoint group `helix.transforms` in our Python package's `setup.py`. Make the following change to `setup.py`:

```
# setup.py

...
entry_points={
    ...
    "helix.transforms": [
        "example-transform = helix_example.transforms.example:ExampleTransform"
    ]
    ...
}
...
```

Note: Similar to Components, the `name` property of our new Component *must* match the name of the entrypoint.

To update the entrypoint list, reinstall the Python package (even if you installed it in editable mode):

```
pip install .
```

Check that our new Transform is registered with the HELIX CLI:

```
helix list
```

The output should include our new example Transform:

```
Available Transforms
    ...
    Example Transform (1.0.0) [example-transform]
    ...
```

Finally, build a `cmake-cpp` Blueprint with our Transform to make sure that it works:

```
helix build blueprint cmake-cpp ./example -t example-transform
```

Note: The generated binary will not do anything, but during generation you should see our Transform print the base64 encoding of the resulting artifact.

1.5.3 Adding Configuration Options

Adding and using configuration options to Transforms can be done in the same way as adding configuration options to Components (see *Adding Configuration Options*).

1.5.4 Adding Dependencies

Specifying Transform dependencies can be done in the same way as specifying Component dependencies (see *Adding Dependencies*).

1.5.5 Testing the Transform

Writing unit tests for Transforms can be done in the same way as writing unit tests for Components (see *Testing the Component*).

1.6 Developing a New Blueprint

Blueprints are at the core of how HELIX generates a build. A properly written Blueprint is generic, reusable, and flexible - designed to work well with many or few Components in many different configurations. Good Blueprints do not define program behavior other than some minimal control flow to set up callsites - that is left to the Components integrated with a Blueprint in a build.

Most developers will not need to write a Blueprint for HELIX - making use of the existing Blueprints built in to HELIX is sufficient for most usecases. However, HELIX is flexible enough to support practically any programming language and any build system - developers simply need to write a Blueprint for their target platform. To demonstrate this, this tutorial will guide you through the process of writing a new Blueprint to support Python.

Prior to following this tutorial, you'll need to have a python package set up to extend HELIX (see *Building a Python Package*).

1.6.1 Writing the Blueprint

Similar to Components and Transforms, Blueprints are simply Python classes that implement the *Blueprint* interface.

Let's start by adding a `blueprints` directory to our Python package with a `python` module for our new Blueprint. The package directory structure should look like the following:

```
.
├── helix_example/
│   ├── blueprints/
│   │   ├── python.py
│   │   ├── __init__.py
│   ├── __init__.py
└── setup.py
```

Inside of `python.py` we'll create a simple Blueprint by subclassing *Blueprint*:

```
# python.py

import os

from helix import blueprint
from helix import utils


class ExamplePythonBlueprint(blueprint.Blueprint):
    """An example Python blueprint."""

    name = "example-python"
    verbose_name = "Example Python Blueprint"
    type = "python"
    version = "1.0.0"
    description = "A simple Python blueprint"

    CALLSITE_STARTUP = "startup"
    """Called at program startup.

    Calls at this callsite are called once and expected to return.
    """

    callsites = [CALLSITE_STARTUP]

    TEMPLATE = """${functions}

if __name__ == "__main__":
    ${startup}
"""

    def filename(self, directory):
        """Generate a build file name in the given directory.

        Args:
            directory (str): The path to the build directory.
```

(continues on next page)

(continued from previous page)

```
        Returns:
            The file path of the build file.
        """

        return os.path.join(directory, "{}.py".format(self.build_name))

    def generate(self, directory):
        functions = "\n".join(self.functions)

        startup = self.calls.pop(self.CALLSITE_STARTUP, [])
        startup = "\n    ".join(startup) or "pass"

        source = utils.substitute(self.TEMPLATE, functions=functions, startup=startup)

        with open(self.filename(directory), "w") as f:
            f.write(source)

    def compile(self, directory, options):
        """Nothing to do here.

        Python is an interpreted language, so we don't really need to do
        anything in the ``compile()`` step. We still need to pass the build
        artifacts to the output, however.
        """

        return [self.filename(directory)]
```

Blueprints have three main components:

1. A set of *callsites* - where components may register calls to functions that they define.

2. A *generate* method which generates source code from the collection of Components provided, using the *functions* and *calls* properties of the Blueprint class. These properties aggregate functions and calls provided by all of the included Components. Source code is written to the given directory path and a list of source files is returned.

3. A *compile* method which compiles the given directory of source files and returns a list of build artifacts. In this case, since Python is not compiled, this simply returns the path to the generated Python file.

This simple Python Blueprint defines a single callsite called `startup` and generates a single python file in the target directory with all included functions and calls to `startup` functions in `__main__`.

Note that the Blueprint does not need to be concerned with how or when Transforms are applied. HELIX will apply Transforms automatically during build based on their type - Blueprints simply need to know how to generate valid source code from Components and compile that source code into build artifacts.

Note: It's generally good practice to define callsite names as constants on the Blueprint class for easier use by Components (e.g., `CALLSITE_STARTUP`).

1.6.2 Registering the Blueprint

Similar to Components and Transforms, Blueprints must be added to the entrypoint group `helix.blueprints` in our Python package's `setup.py`. Make the following change to `setup.py`:

```
# setup.py

...
entry_points={
    ...
    "helix.blueprints": [
        "example-python = helix_example.blueprints.python:ExamplePythonBlueprint"
    ]
    ...
}
...
```

Note: Similar to Components and Transforms, the `name` property of our new Blueprint *must* match the name of the entrypoint.

To update the entrypoint list, reinstall the Python package (even if you installed it in editable mode):

```
pip install .
```

Check that our new Blueprint is registered with the HELIX CLI:

```
Available Blueprints
    ...
    Example Python Blueprint (1.0.0) [example-python]
    ...
```

Finally, build an empty `example-python` Blueprint to make sure that it works:

```
helix build blueprint example-python ./example
```

Take a look at the generated Python script - it's not particularly interesting right now but we'll add a Component for our new Blueprint next.

1.6.3 Writing a Component for the Blueprint

Let's create a minimal Component to test our new Blueprint in much the same way we created our first Component in *Writing Your First Component*.

First, let's add a new module to the `components` directory of our Python package called `python` to house our new Component. The package directory structure should look like the following:

```
.
├── helix_example/
    ├── components/
        ├── example.py
        ├── python.py
        ├── __init__.py
```

(continues on next page)

(continued from previous page)

```
    ├── __init__.py
└── setup.py
```

Inside of `python.py` we'll create a simple Component:

```
# python.py

from helix import component


class ExamplePythonComponent(component.Component):
    """An example Python component."""

    name = "example-python-component"
    verbose_name = "Example Python Component"
    type = "example"
    version = "1.0.0"
    description = "An example Python component"
    date = "2020-10-20 12:00:00.000000"
    tags = (("group", "example"),)

    blueprints = ["example-python"]

    functions = [
        """def ${example}():
    print("hello world")
"""
    ]
    calls = {"startup": ["${example}()"]}
    globals = ["example"]
```

This is a very simple Component that defines one function that prints "hello world" and registers a call to it at the `startup` callsite.

Next, we need to register the new Component with the `helix.components` entrypoint group. Make the following change to `setup.py`:

```
# setup.py

...
entry_points={
    ...
    "helix.components": [
        ...
        "example-python-component = helix_example.components.
↪python:ExamplePythonComponent",
        ...
    ]
    ...
}
...
```

To update the entrypoint list, resintall the Python package (even if you installed it in editable mode):

```
pip install .
```

Check that the new Component is registerd with the HELIX CLI:

```
helix list
```

The output should include the new Component:

```
Available Components:
    ...
    Example Python Component (1.0.0) [example-python-component]
    ...
```

Now we can test our new Blueprint with the new Component:

```
helix build blueprint example-python ./example -c example-python-component
```

The generated Python script should simply print "hello world" and exit.

1.6.4 Adding Another Callsite

Blueprints are not limited to exposing only a single, trivial callsite. Blueprints can evoke very sophistocated behavior from their Components by exposing multiple different types of callsites. To demonstrate this, let's add another callsite to our Blueprint called `loop` which is called repeatedly inside of a loop defined in the Blueprint.

Make the following changes to the Blueprint:

```
# blueprints/python.py

...

Class ExamplePythonBlueprint(blueprint.Blueprint):
    ...

    CALLSITE_LOOP = "loop"
    """Called every five seconds, indefinitely.

    Calls this callsite repeatedly, inside of a loop, until the program is
    terminated.
    """

    callsites = [CALLSITE_STARTUP, CALLSITE_LOOP]

    ...

    TEMPLATE = """import time

${functions}

if __name__ == "__main__":
    ${startup}

    while True:
```

(continues on next page)

(continued from previous page)

```
        ${loop}

        time.sleep(5)
"""

def generate(self, directory):
    ...

    loop = self.calls.pop(self.CALLSITE_LOOP, [])
    loop = "\n        ".join(loop) or "break"

    ...

    source = utils.substitute(
        self.TEMPLATE, functions=functions, startup=startup, loop=loop
    )

    ...
```

Note that we've chosen to set the `loop` template parameter to `break` if no calls are registered at that callsite. This makes our Blueprint more flexible - if no calls are registered for the `loop` callsite the Blueprint will simply break out of its infinte loop.

Next, let's update the simple testing Component for this Blueprint to make use of the new callsite. Make the following changes to the Component:

```
# components/python.py

...

class ExamplePythonComponent(component.Component):
    ...

    functions = [
        ...
        """from datetime import datetime

def ${now}():
    print(datetime.now())
""",
        ...
    ]

    ...

    calls = {
        ...
        "loop": ["${now}()"],
        ...
    }

    ...
```

(continues on next page)

(continued from previous page)

```
    globals = ["example", "now"]
```

This adds a single function which prints the current date and time and adds a call at the `loop` callsite to that new function.

After reinstalling the python package (if not installed in editable mode), we can now create a new build with our updated Blueprint and Component:

```
helix build blueprint example-python ./example -c example-python-component
```

You should now have a Python script that prints "hello world" once and then repeatedly prints the current time every five seconds indefinitely.

Note: Blueprint Flexibility: When developing new Blueprints, it can be tempting to add a lot of project structure and even some core program functionality to Blueprints by implementing various callsites. A best practice is to limit the functionality inside of a Blueprint to only control flow and ensure that all callsites are optional. Remember: callsites must be able to support *zero or more* calls from components. A generic Blueprint is a reusble Blueprint.

1.6.5 Adding Dependencies

Specifying Blueprint dependencies can be done in the same way as specifying Component dependencies (see *Adding Dependencies*).

1.6.6 Testing the Blueprint

Writing unit tests for Transforms can be done in the same way as writing unit tests for Components (see *Testing the Component*).

1.7 Further Reading & Next Steps

- The complete python package created through the course of this tutorial is included in the `examples/` directory of the HELIX repo for your reference.
- Read through the detailed *Using HELIX* section for more information on using specific parts of HELIX.
- Browsing the source code of built-in HELIX Blueprints, Components, and Transforms can be a good way to learn about HELIX conventions, tagging, and design.
- Check out the *Reference* documentation for more detail on any of the interfaces provided by HELIX, a listing of the included Blueprints, Components, and Transforms, and details on all of the CLI commands.

CHAPTER

TWO

USING HELIX

2.1 Building

There are a couple of supported ways to build a Blueprint and a collection of Components and Transforms with the HELIX CLI. Namely, you can specify a configuration via command line arguments or you can generate a build from a JSON configuration file.

2.1.1 Building with CLI Arguments

A simple example (given elsewhere in this documentation) builds the `cmake-cpp` Blueprint with the `configuration-example` Component and the `strip` Transform and writes the output to `./example`:

```
helix build blueprint cmake-cpp ./example \
    -c configuration-example:second_word=foo \
    -t strip
```

It is also possible to specify multiple components and transforms to generate an arbitrarily complex build. For example, using some of HELIX's built-in Components and Transforms, we can generate a build that:

1. Downloads a remove file (https://www.google.com/).
2. Compresses the file using `zlib`.
3. Encrypts the compressed file with `aes` and a fixed key.
4. Deletes the unencrypted file.
5. Deletes the uncompressed file.
6. Timestomps the compressed, encrypted download.

We can then strip and UPX compress this build, all using only built-in HELIX Components and Transforms:

```
helix build blueprint cmake-cpp example \
    -c linux-libcurl-remote-file-copy:url=https://www.google.com/,output=test.txt \
       linux-zlib-compress-data-compressed:input=test.txt,output=test.txt.gz \
       linux-openssl-aes-encrypt-data-encrypted:input=test.txt.gz,output=test.txt.gz.enc,
↪key=abcdefghijklmnopqrstuvwxyzabcdef \
       linux-remove-file-deletion:path=test.txt \
       linux-remove-file-deletion:path=test.txt.gz \
       linux-utime-timestomp:path=test.txt.gz.enc,timestamp="2010-01-01 12:00:00" \
    -t strip upx
```

2.1.2 Building from JSON Configuration

The HELIX CLI can also take configuration from a JSON file. For example, we can build the following configuration:

```
{
    "name": "example",
    "blueprint": {"name": "cmake-cpp"},
    "components": [
        {
            "name": "minimal-example",
            "configuration": {}
        },
        {
            "name": "configuration-example",
            "configuration": {
                "second_word": "example"
            }
        }
    ],
    "transforms": [
        {
            "name": "replace-example",
            "configuration": {
                "old": "hello",
                "new": "goodbye"
            }
        },
        {
            "name": "strip"
        }
    ]
}
```

by running the following command:

```
helix build json configuration.json ./example
```

2.2 Testing

HELIX includes three different categories of tests:

system Unit tests for HELIX's core system functionality - these are typically only run by core HELIX developers.

unit Unit tests for individual Blueprints, Components, and Transforms - these are written by internal and external developers and can be used to test custom Blueprints, Components, and Transforms as well as those built into HELIX.

integration Integration tests for running multiple sets of Blueprints, Components, and Transforms together to ensure they work in concert.

These test suites can be run with the the `test` CLI command. For example, to run all unit tests for Blueprints, Components, and Transforms, run:

```
helix test unit
```

2.3 Using the Python API

HELIX Blueprints, Components, and Transforms can also be used directly by instantiating and manipulating the relevant classes directly in Python. This can make scripting with HELIX somewhat easier.

For example, the following script generates a simple HELIX build with a few simple Components and Transforms:

```
from helix import utils

output = "./example"

MinimalExampleComponent = utils.load("helix.components", "minimal-example")
minimal = MinimalExampleComponent()
minimal.generate()
minimal.finalize()

ConfigurationExampleComponent = utils.load("helix.components", "configuration-example")
config = ConfigurationExampleComponent()
config.configure(second_word="example")
config.generate()
config.finalize()

components = [minimal, config]

ReplaceExampleTransform = utils.load("helix.transforms", "replace-example")
replace = ReplaceExampleTransform()
replace.configure(old="hello", new="goodbye")

StripTransform = utils.load("helix.transforms", "strip")
strip = StripTransform()

transforms = [replace, strip]

CMakeCppBlueprint = utils.load("helix.blueprints", "cmake-cpp")
blueprint = CMakeCppBlueprint("example", components, transforms)
artifacts = blueprint.build(output)

for artifact in artifacts:
    print(artifact)
```

CHAPTER

THREE

REFERENCE

3.1 Blueprints

class helix.blueprint.**Blueprint**(*build_name*, *components=None*, *transforms=None*, **args*, ***kwargs*)

A common base class for all blueprints.

abstract property callsites

A list of valid callsites which Components may use.

A good practice is to define these as constants on the Blueprint class as well so they may be referenced directly by components and don't rely on matched strings.

sane()

Ensure the current list of components is sane.

This method is optional and allows the developer to define additional constraints on the components supported by this Blueprint. This method should raise exceptions if the build is not sane.

property tags

Returns the union of all tags involved in this blueprint.

Simply aggregates and dedups all tags on Components and Transforms.

property functions

Aggregate all functions from included Components.

property calls

Aggregate all calls from included Components.

abstract generate(*directory*)

Generates a source directory from an iterable of components.

The components passed to this class will be finalized prior to calling this so that their source code is readily available. This function is responsible for writing configured components to the target output directory.

Parameters **directory** (*str*) – A directory to write the resulting generated source code - you may assume that this directory already exists and is writable.

Returns A list of generated source files.

Note: Component order matters here - this is the order in which calls and includes will be inserted into the generated source so if this is important you should make sure the order of components passed to this function is what you want.

---

transform(*type*, *targets*)

Applies all Transforms of a given type.

Parameters

- **type** (*str*) – The type of transform to apply.
- **targets** (*str*) – A list of targets to transform.

abstract compile(*directory*, *options*)
Compiles the target directory.

Compiles a directory generated with `generate`, applies any artifact transformations, and returns a list of build artifacts.

Parameters

- **directory** (*str*) – A directory with generated source code.
- **options** (*dict*) – An optional dictionary of additional build options that should be respected by this function. This will contain things like `stdout`, `stderr` and `propagate` for display options.

Returns A list of built artifacts.

Note: In practice this may frequently just be a bunch of calls to `os.system` to invoke the target build system of this Blueprint.

---

build(*directory*, *options=None*)
Fully builds this Blueprint.

Generates code from Components, applies source Transforms, compiles code, and applies artifact Transforms.

Parameters

- **directory** (*str*) – A directory to write the resulting generated source code.
- **options** (*dict*) – An optional dictionary of additional build options that should be used by `compile`.

Returns A list of built and transformed artifacts.

dependencies = []
A list of external dependencies which must be installed.

This list is optional, but must consist of zero or more instances of `Dependency` (or subclasses) which may be installed by users.

abstract property description

classmethod install(*verbose=False*)
Install all dependencies.

This reruns dependency installation regardless of if it has been installed already. This should be safe because of the requirement that dependency installation methods be repeatable but can be inefficient. Users can check the `installed()` method to determine if the installation should be rerun.

Parameters **verbose** (*bool*) – Verbosity.

classmethod installed()
Check if all dependencies have been installed.

Returns `True` if all dependencies have been installed already, and `False` otherwise.

abstract property name
A simple name.

Names should consist of *[a-z-]*. Make use of the optional `verbose_name` property if you would like a pretty-printable version of this name.

classmethod string()

abstract property type
A short type descriptor.

This should be set to some constant (a short string) defined in some common location.

abstract property verbose_name
Optional verbose name for pretty printing.

By default this will just return `name`.

abstract property version
A version number.

This should be a string using Semantic Versioning.

3.2 Components

class helix.component.**Component**(**args*, ***kwargs*)
A common base class for all components.

A component represents a single, configurable unit of functionality. This may be as simple as a single function or as complex as an entire command and control library.

DATE_FORMAT = '%Y-%m-%d %H:%M:%S.%f'

abstract property date
A relevant date.

Possibly the initial publication date of this sample or compile timestamp. This does not need to be exact but may be useful for some applications of the final dataset and will be included.

Note: The date should be in a string format parsable with `DATE_FORMAT`

---

abstract property blueprints
A list of the Blueprints supported by this component.

This should be a list of the string names of supported Blueprint and must match the corresponding `Blueprint.name` property.

functions = []
A list of functions included in this Component.

This list of functions may be defined either in the Component class (for relatively simple components) or may be populated by the implementation of the `generate()` method.

calls = {}
A dictionary of callsites (defined by Blueprints) to call strings.

This dictionary of calls may be defined either in the Component class (for relatively simple components) or may be populated by the implementation of the `generate()` method.

`globals = []`
A list of template parameter names that must be globally unique.

These template parameters are present in the `functions` and `calls` parameters and will be substituted for a globally unique value when this Component is finalized. This list should encompass all of the parts of this Component that must be globally unique (e.g., function names, global variable names). A good test to ensure that all of the globally unique parameters have been templated is to try to build a Blueprint with two instances of this component.

`property generated`
Indicates if this Component has been generated yet.

Note: This assumes that the Component has been generated if functions or calls have been set. This means that components which define a generate method should leave these parameters empty and set them inside of the `generate()` method only. A hybrid aproach of partially setting these parameters and then updating them inside of the `generate()` method is not advised.

`property configuration`
Parsed configuration parameters.

This is populated by the `configure` method, controlled by the `options` parameter. If this has not yet been configured, this raises a `NotConfigured` exception.

`configure`(***kwargs*)
Parse configuration data passed as kwargs.

Configuration values taken from `options` will be passed to this function as kwargs. This function is responsible for parsing and storing those configuration options and storing them in `configuration`.

Parameters **`**kwargs`** – Arbitrary keyword arguments corresponding to fields in `options`

Note: Although it is possible to pass values of varying types to this method, it is recommended that code which makes use of configuration parameters assumed that they are a string, since configuration parameters parsed from command line arguments and configuration files will be passed as strings.

`property configured`
Indicates if this has been configured.

This attribute is set `True` by the `configure` method. Components are reconfigurable, so `configure` may still be called if `configured` is `True`.

A Configurable with no configuration options is considered to be already configured.

`dependencies = []`
A list of external dependencies which must be installed.

This list is optional, but must consist of zero or more instances of `Dependency` (or subclasses) which may be installed by users.

`abstract property description`

`generate`()
Generate configured source code for this component.

This function is responsible for any necessary code generation. `configure` will be called before generate so this function may draw from any instance attributes (set by `configure`) for configuration parameters. This function may be used to populate/modify the `functions` and `calls` properties of this Component.

Note: Defining this method is optional - relatively simple components may find it easy enough to simply define all of the data they may need directly on the class.

classmethod install(*verbose=False*)
Install all dependencies.

This reruns dependency installation regardless of if it has been installed already. This should be safe because of the requirement that dependency installation methods be repeatable but can be inefficient. Users can check the `installed()` method to determine if the installation should be rerun.

Parameters **verbose** (*bool*) – Verbosity.

classmethod installed()
Check if all dependencies have been installed.

Returns `True` if all dependencies have been installed already, and `False` otherwise.

abstract property name
A simple name.

Names should consist of *[a-z-]*. Make use of the optional `verbose_name` property if you would like a pretty-printable version of this name.

options = {}
A dictionary of configurable options and default values.

This defines which options which may be edited with a call to `configure`.

Example

A subclass with both required and default parameters may be defined as:

```
options = {
    "server": {},
    "port": {"default": 1337}
}
```

classmethod string()

tags = ()
An optional iterable of human-readable tag tuples.

Tags may represent family or component groupings and are fairly loosely defined.

Example

An `APT29/SEDINT` sample may be defined as `(("family", "APT29"), ("sample", "SEDINT"))`

abstract property type
A short type descriptor.

This should be set to some constant (a short string) defined in some common location.

validate_configuration()
Custom option validation code.

This optional method may be implemented on subclasses to provide custom configuration validation and is called by the `configure` method. If invalid configuration is detected, this method should raise

ConfigurationError. Parsed configuration values are stored in configuration by the time this is called.

abstract property verbose_name
Optional verbose name for pretty printing.

By default this will just return name.

abstract property version
A version number.

This should be a string using Semantic Versioning.

finalized = False

finalize()
Make this Component unique.

Uses the globals list to generate and insert globally unique values into the functions and calls properties to prepare this Component to be used by a Blueprint.

3.3 Transforms

class helix.transform.**Transform**(**args*, ***kwargs*)
A common base class for all transforms.

TYPE_SOURCE = 'source'
A source-to-source transformation.

Setting Transform.type to TYPE_SOURCE indicates to Blueprints that this Transform should be fed source files and should produce modified source files.

TYPE_ARTIFACT = 'artifact'
An artifact-to-artifact transformation.

Setting Transform.type to TYPE_ARTIFACT indicates to Blueprints that this Transform should be fed built artifacts and should produce modified built artifacts.

property configuration
Parsed configuration parameters.

This is populated by the configure method, controlled by the options parameter. If this has not yet been configured, this raises a NotConfigured exception.

configure(***kwargs*)
Parse configuration data passed as kwargs.

Configuration values taken from options will be passed to this function as kwargs. This function is responsible for parsing and storing those configuration options and storing them in configuration.

Parameters ****kwargs** – Arbitrary keyword arguments corresponding to fields in options

---

Note: Although it is possible to pass values of varying types to this method, it is recommended that code which makes use of configuration parameters assumed that they are a string, since configuration parameters parsed from command line arguments and configuration files will be passed as strings.

---

property configured
Indicates if this has been configured.

This attribute is set `True` by the `configure` method. Components are reconfigurable, so `configure` may still be called if `configured` is `True`.

A Configurable with no configuration options is considered to be already configured.

dependencies = []
A list of external dependencies which must be installed.

This list is optional, but must consist of zero or more instances of `Dependency` (or subclasses) which may be installed by users.

abstract property description

classmethod install(*verbose=False*)
Install all dependencies.

This reruns dependency installation regardless of if it has been installed already. This should be safe because of the requirement that dependency installation methods be repeatable but can be inefficient. Users can check the `installed()` method to determine if the installation should be rerun.

Parameters **verbose** (*bool*) – Verbosity.

classmethod installed()
Check if all dependencies have been installed.

Returns `True` if all dependencies have been installed already, and `False` otherwise.

abstract property name
A simple name.

Names should consist of *[a-z-]*. Make use of the optional `verbose_name` property if you would like a pretty-printable version of this name.

options = {}
A dictionary of configurable options and default values.

This defines which options which may be edited with a call to `configure`.

Example

A subclass with both required and default parameters may be defined as:

```
options = {
    "server": {},
    "port": {"default": 1337}
}
```

classmethod string()

supported(*source*)
Check if the given source is supported.

This method is optional - the default behavior is to assume all sources are suppored.

Parameters **source** (*str*) – The source material for this tranformation.

Returns `True` if the target is supported by this transform, `False` otherwise.

tags = ()
An optional iterable of human-readable tag tuples.

Tags may represent family or component groupings and are fairly loosely defined.

Example

An APT29/SEDINT sample may be defined as `(("family", "APT29"), ("sample", "SEDINT"))`

abstract property type
A short type descriptor.

This should be set to some constant (a short string) defined in some common location.

validate_configuration()
Custom option validation code.

This optional method may be implemented on subclasses to provide custom configuration validation and is called by the `configure` method. If invalid configuration is detected, this method should raise `ConfigurationError`. Parsed configuration values are stored in `configuration` by the time this is called.

abstract property verbose_name
Optional verbose name for pretty printing.

By default this will just return `name`.

abstract property version
A version number.

This should be a string using Semantic Versioning.

abstract transform(*source*, *destination*)
Apply this transform.

Applies this transform to the given `source` and writes the output to `destination`. If this transform expects configuration, this method should raise exceptions if `configured` is `False` when this is called.

Parameters

- **source** (*str*) – The source material for this tranformation.
- **destination** (*str*) – The destination to write the output of this transformation.

3.4 Dependencies

class helix.utils.**Dependency**
A base class for dependency definitions.

This defines the interface for all dependency installers.

abstract install(*verbose=False*)
Install this dependency.

Raise a `DependencyInstallationFailure` on error.

Parameters **verbose** (*boolean*) – Verbosity.

abstract installed()
Check if this dependency is installed.

Returns `True` if this has already been installed, and `False` otherwise.

string()
The string representation of this dependency.

Example

The name of the package to install or the path to a script to run.

3.4.1 Included Dependencies

class helix.utils.**ManualPATHDependency**(*name*, *help=None*)
A manually installed dependency that resides in the PATH.

This is a minor extension of `ManualDependency` which adds additional functionality to check if target dependency has been installed and resides in the PATH.

Note: `name` must match the target binary to find in the PATH or this will not work.

---

installed()
Check the PATH for the existence of `name`.

class helix.utils.**ManualWindowsRegistryDependency**(**args*, *registry*, *key*, ***kwargs*)
A manually installed dependency that sets a Windows registry key.

This checks for the existence of a Windows registry key to verify the installation status of a dependency.

Parameters

- **name** (*str*) – The name of this dependency.
- **help** (*str*) – A short description of how to install this dependency manually. Optional, but recommended.
- **key** (*str*) – The full path to the key to check.

installed()
Check the registry for the existence of `key`.

This uses the built-in `winreg` API to check for the existence of the key.

class helix.utils.**WindowsChocolateyDependency**(*package*)
A dependency that may be installed via the Chocolatey package manager.

Parameters **package** (*str*) – The package name.

Note: These sorts of dependencies require that the plaform under which this Blueprint/Component/Transform is run includes Chocolatey and assume that it is already installed (i.e., this will not work on Linux).

Note: These dependencies require Administrator permissions to install successfully.

---

install(*verbose=False*)
Install the package with Chocolatey.

installed()
Check if the package is installed with Chocolately.

string()
The string representation of this dependency.

Example

The name of the package to install or the path to a script to run.

class helix.utils.**LinuxAPTDependency**(*package*)
A dependency that may be installed via the Advanced Package Tool (APT).

Parameters **package** (*str*) – The package name.

Note: These sorts of dependencies require that the plaform under which this Blueprint/Component/Transform is run includes APT and assume that it is already installed (i.e., this will not work on Windows).

Note: These dependencies require root permissions to install successfully.

---

install(*verbose=False*)
Install the package with APT.

installed()
Check if the package is installed with dpkg.

string()
The string representation of this dependency.

Example

The name of the package to install or the path to a script to run.

3.5 Testing

class helix.tests.**UnitTestCase**(*methodName='runTest'*)
The base class for all Blueprint, Component, and Transform tests.

Provides utilities for safely generating builds during unit tests.

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

build(*configuration*)
Build a given configuration in the current working directory.

Tests may make use of the build artifacts to determine success/failure by using `utils.run` as necessary.

Parameters **configuration** – A build configuration dictionary.

Returns A list of build artifacts from the successful build.

class helix.tests.**BlueprintTestCaseMixin**
Provides testing utilities and minimal tests for Blueprints.

Note: Requires a base class of `UnitTestCase`.

---

abstract property blueprint
The Blueprint name.

test_build_success()
Tests that this Blueprint builds successfully.

Builds this Blueprint on its own with no Components or Transforms. As long as no errors are raised in building, this test will pass.

class helix.tests.**ComponentTestCaseMixin**
Provides testing utilities and minimal tests for Components.

Note: Requires a base class of `UnitTestCase`.

---

abstract property blueprint
The Blueprint that this Component should be built with.

abstract property component
The Component name.

configuration = {}
Optional configuration parameters for building.

If the Component requires configuration for a basic build, specify an example configuration here.

test_build_success()
Tests that this Component builds successfully.

Builds this Component on its own with the configured Blueprint and nothing else. As long as no errors are raised in building, this test will pass.

test_duplicate_build()
Tests that duplicating this Component builds successfully.

Attempts a minimal build with only this Component, duplicated. This is a useful test to determine if the Component has been parameterized successfully. If not, the build will likely fail with errors complaining about duplicated functions or globals.

class helix.tests.**TransformTestCaseMixin**
Provides testing utilities and minimal tests for Transforms.

Note: Requires a base class of `UnitTestCase`.

---

abstract property blueprint
The Blueprint that this Transform should be built with.

abstract property transform
The Transform name.

configuration = {}
Optional configuration parameters for building.

If the Transform requires configuration for a basic build, specify an example configuration here.

test_build_success()
Tests that this Transform builds successfully.

Builds this Transform on its own with the configured Blueprint and nothing else. As long as no errors are raised in building, this test will pass.

3.6 Utilities

helix.utils.**source**(*package*, *resource*)
Fetch the content of a specific file in this package.

Parameters

- **package** (*str*) – The package path relative to which the resource exists. In most cases you probably want to supply `__name__` to search relative to the current code.
- **resource** (*str*) – The resource filename to load.

Returns The content of the package resource as a string.

helix.utils.**substitute**(*template*, *safe=True*, ***kwargs*)
Substitute parameters in a template string.

Template substitution makes use of Python string templates (described in PEP 292).

Parameters

- **template** (*str*) – The template string.
- **safe** (*bool*) – If `True`, missing parameters will be ignored and some template parameters may remain in the return value.
- ****kwargs** – Template parameters.

Returns The substituted template string.

helix.utils.**find**(*name*, *environment=None*, *guess=None*)
Finds a particular binary on this system.

Attempts to find the binary given by `name`, first checking the value of the environment variable named `environment` (if provided), then by checking the system path, then finally checking hardcoded paths in `guess` (if provided). This function is cross-platform compatible - it works on Windows, Linux, and Mac. If there are spaces in the path found, this function will wrap its return value in double quotes.

Parameters

- **name** (*str*) – Binary name.
- **environment** (*str*) – An optional environment variable to check.
- **guess** (*iterable*) – An optional list of hardcoded paths to check.

Returns A string with the absolute path to the binary if found, otherwise `None`.

helix.utils.**run**(*cmd*, *cwd=None*, *exception=None*, *propagate=False*, *stdout=None*, *stderr=None*)
Run the given command as a subprocess.

This function caputres `stdout` and `stderr` by default and returns them, and raises the given exception if the process fails.

Parameters

- **cmd** (*str*) – The command to run.
- **cwd** (*str*) – The working directory in which to run the command.
- **exception** – An exception to raise if the command fails.
- **propagate** (*bool*) – If `True`, command output will be written to stdout and stderr of the current process, otherwise command output is captured and returned (and written to `stdout` and `stderr` if provided). Default: `False`.

- **stdout** (*file*) – An open file-like object or fileno where stdout should be written or None.
- **stderr** (*file*) – An open file-like object or fileno where stderr should be written or None.

Returns Output to stdout and stderr as binary strings.

Note: Without adding sufficient complexity (additional threads) output cannot be both captured and printed stdout/stderr of the current process in real time. If this is called with `propagate=True`, then no output will be returned or written to the provided `stdout`/`stderr` arguments.

---

helix.build.**build**(*configuration*, *output*, *options=None*)

Build a given configuration.

Builds the given `configuration` dictionary using `output` as a working directory.

Parameters

- **configuration** – A dictionary describing blueprint, components, and transforms to use for this build.
- **output** (*str*) – The path to the output directory.
- **options** (*dict*) – An optional dictionary of additional options to be passed to the build command.

Returns A list of build artifact paths.

Example

Example configuration dictionary:

```
{
    "name": "test",
    "blueprint": {"name": "cmake-cpp"},
    "components": [
        {"name": "minimal-example"},
        {
            "name": "configuration-example",
            "configuration": { "second_word": "world" }
        }
        {"class": components.MinimalExampleComponent}
    ],
    "transforms": [
        ...
    ]
}
```

Configuration parameters are passed via the `configuration` key. Note that Blueprints, Components and Transforms may be specified by either `name` or directly by `class`.

3.7 CLI Commands

class helix.management.commands.install.**Command**

Install external dependencies.

```
usage: helix install [-h] [-c] [-f] [-v] {blueprints,components,transforms} ...

positional arguments:
  {blueprints,components,transforms}
    blueprints          install dependencies for blueprints
    components          install dependencies for components
    transforms          install dependencies for transforms

optional arguments:
  -h, --help            show this help message and exit
  -c, --check           check the installation status without installing anything
  -f, --fail-fast       stop installing if there are any failures
  -v, --verbose         verbose mode
```

class helix.management.commands.list.**Command**

Print details about blueprints, components, and transforms.

```
usage: helix list [-h] [-s SEARCH] [-d] [-y] [-t] [-c] [-r] [-v] [-vv] [-vvv]
↪{blueprints,components,transforms} ...

positional arguments:
  {blueprints,components,transforms}
        blueprints          list blueprints
        components          list components
        transforms          list transforms

optional arguments:
  -h, --help            show this help message and exit
  -s SEARCH, --search SEARCH
                                                search with a given string
  -d, --description     include description
  -y, --type            include type
  -t, --tags            include tags
  -c, --configuration   include configuration parameters
  -r, --dependencies    include dependencies
  -v, --verbose         include description and tags
  -vv, --very-verbose   verbose plus configuration
  -vvv, --very-very-verbose
                                                very verbose plus type and␣
↪dependencies
```

class helix.management.commands.build.**Command**

Build a blueprint with a set of components and transforms.

```
usage: helix build [-h] {blueprint,json} ...

positional arguments:
  {blueprint,json}
```

(continues on next page)

(continued from previous page)

```
    blueprint       manually specify blueprint, components, and transforms
    json            build from a json configuration file

optional arguments:
  -h, --help        show this help message and exit
```

class helix.management.commands.test.**Command**

Run tests.

```
usage: helix test [-h] {unit,system,integration} ...

positional arguments:
  {unit,system,integration}
    unit                unit tests for blueprints, components, or transforms
    system              unit tests for core helix functionality
    integration         integration tests for combinations of blueprints,
                        components, and transforms

optional arguments:
  -h, --help            show this help message and exit
```

CHAPTER

FOUR

ABOUT

4.1 Disclaimer

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

- Subject to FAR 52.227-11 – Patent Rights – Ownership by the Contractor (May 2014)
- SPDX-License-Identifier: MIT

This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

4.1.1 License

INDEX