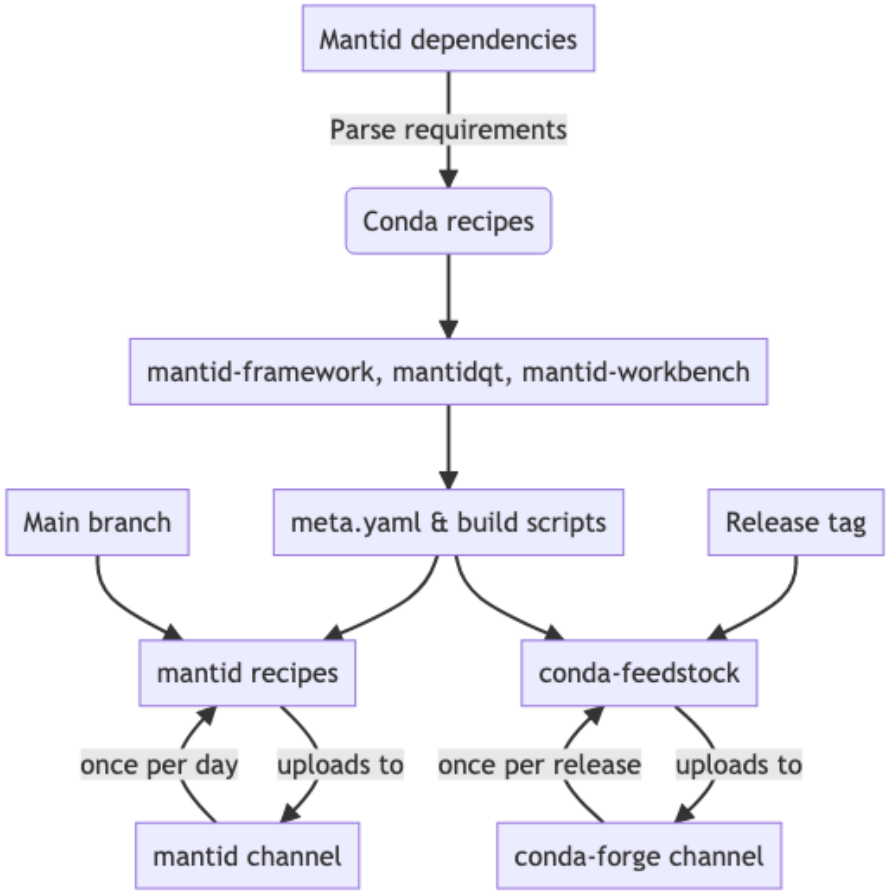
Conda build (nightly and release)



Conda forge

Uses conda smithy and conda build to create a package of your supplied source code. To submit a new package you fork the staged recipes repository. https://github.com/conda-forge/staged-recipes

Next, for your package you create a folder in the recipes folder. To build a c++ package this folder will typically contain:

- A build.sh file for osx and linux building
- A build.bat file for windows
- A meta.yaml file specifying your required environment.

The build scripts will typically contain make / cmake commands used to compile your project. E.g at is simplest level it could contain

```
mkdir build
cd build
```

```
cmake \
  ${CMAKE_ARGS} \
  -DCMAKE_BUILD_TYPE=Release \
  ..

ninja
ninja install
```

Note: CMAKE_ARGS is used on linux and osx to supply common arguments to all conda-forge packages, e.g install directory and openmp flags amongst others.

The meta.yaml file is used to provide conda-build the build, host and runtime requirements. For example, consider the requirements for mantid-framework,

```
requirements:
  build:
    - {{ compiler("c") }}
    - {{ compiler("cxx") }}
    - llvm-openmp  # [osx]
    - libgomp  # [linux]
    - ninja  # [unix]
    - cmake
    - {{ cdt('mesa-libgl-devel') }}  # [linux]
    - {{ cdt('mesa-dri-drivers') }}  # [linux]
    - {{ cdt('libxxf86vm') }}  # [linux]
    - {{ cdt('libx11-devel') }}  # [linux]
    - {{ cdt('xorg-x11-proto-devel') }}  # [linux]
  host:
    - boost {{ boost }}
    - eigen
    - gsl<=2.6
    - h5py
    - hdf5=1.10.*
    - jemalloc  # [unix]
    - jsoncpp
    - librdkafka
    - lib3mf
    - muparser
    - nexus
    - numpy {{ numpy }}
    - occt
    - python {{ python }}
    - poco
    - tbb-devel=2020.2.*
    - zlib
    - pip
  run:
    - {{ pin_compatible("boost", max_pin="x.x") }}
    - {{ pin_compatible("gsl", max_pin="x.x") }}
    - h5py
```

```
    - lib3mf
    - matplotlib-base
    - nexus
    - {{ pin_compatible("numpy", max_pin="x.x") }}
    - occt
    - python
    - python-dateutil
    - pyyaml
    - scipy
```

Separating out the build and host requirements allows cross compilation where host specifies your target system for package (i.e the system which the dependencies need to have been compiled for) whereas the build environment typically contains just the build tools, e.g the compiler and CMake.

The runtime requirements are needed as some build or host requirements section will impose a runtime requirement. Most commonly this is true for shared libraries (boost), which are required for linking at build time, and for resolving the link at run time.

To ensure we get compataible runtime, we can use pinning expressions in the runtime dependencies:

```
- {{ pin_compatible("gsl", max_pin="x.x") }}
```

With this example, if we built with gsl version 1.6.1 this pinning expression would evaluate to gsl>=1.6.1,<1.7.

Another way to include runtime dependencies is to define a run_exports section in the recipe, for example jsoncpp includes in its meta.yaml:,

```
build:
  number: 3
  run_exports:
    - {{ pin_subpackage('jsoncpp', max_pin='x.x.x') }}
```

This means if we include jsoncpp in our host requirements section, as with mantid, it will automatically be added to our runtime dependencies. This was developed to avoid people forgetting to include runtime dependencies in their package. THis example means if we built with jsoncpp 1.8.1 we would accept the following versions at runtime jsoncpp>=1.8.1,<1.8.2

This is very strict pinning, the reason for this can be seen by looking at the abi-laboratory entry for jsoncpp: https://abi-laboratory.pro/index.php?view=timeline&l=jsoncpp

Which shows some abi change between patch versions.

Alongside these requirements, sits several other required fields, these include:

- test:
  - Things in this section are run after the package has been built to test it is complete, e.g python imports. If a test fails the package is marked as broken.
- source:

    - Defines where to source the pacakge code from, e.g through a github link to a tar containing the source code - usually specified using a github tag.
- extra:
    - Package maintainers
- about
    - Package information and license.

Examples of the above fields are shown below:

```
test:
  imports:
    - mantid.kernel
    - mantid.geometry
    - mantid.api
    - mantid.simpleapi

build:
  number: 0
  run_exports:
    - {{ pin_subpackage('mantid-framework', max_pin='x.x.x') }}
```