Galaxy IUC Standards and Best Practices Documentation

Release 0.1

Galaxy IUC

Mar 29, 2024

CONTENTS

With the extensive efforts developers from the IUC have put into developing tools over the years, we have tried to come up with some best practices that we would recommend to the community for their use.

Contents:

CHAPTER

ONE

BEST PRACTICES FOR CREATING GALAXY TOOLS

This page is written by experienced tool developers and contains information about what practices in Galaxy tool development tend to be the most successful ones.

1.1 Why you might read this

If you need to maintain existing tools or develop new tools for Galaxy, this programmer oriented guide, contributed by the community, will detail current collected best practice guidelines for building and maintaining automatically installing reproducible Galaxy tools.

1.2 Definitions

Follows a short summary of the key parts when it comes to Galaxy Tools.

- ToolShed, The Galaxy AppStore - companion Galaxy web server for tools. E.g. https://toolshed.g2.bx.psu.edu or localhost:9009 for Mercurial based source code management and automated installation of all components described below through any Galaxy admin interface.
- Galaxy Tool - An application specific, XML defined interface and associated documentation exposing any command line application as a form-driven Galaxy tool - e.g. BWA or bamtools. Ideally, as a shareable tool shed repository, supporting automated Galaxy installation with revision/version specific control of dependency binaries for reproducible analyses.
- Tool Dependency Package - Tool Shed tools in the Tool Dependency Packages category whose names start with `package_` such as `package_samtools_1_0_0`, automate the installation of a specific version of some command line application software that other tools depend on. Each dependency package may be shared by many tools and are only available to users through tool forms that populate command lines and execute them.
- Datatypes - Galaxy has a flexible and extensible internal representation for specialised data formats such as fasta sequences, fastq short read data or tabular text persisted in Galaxy histories. Tools and packages can extend Galaxy by installing new datatypes when needed.
- DataManagers - Large scale scientific analyses often involve local copies of canonical reference data such as reference genomes and application specific index files for annotation and mapping in genomics. In most cases these are rapidly evolving and a constant drain on highly skilled resources to keep up to date manually. Data Managers can be built and shared to automate reference data maintenance by the local Galaxy administrators. Data Manager repositories should start with `data_manager_`.

1.3 Tools and Tool Development

1.3.1 Tools

Before you start writing a new tool please search the Main Tool Shed (MTS) and the Test Tool Shed (TTS) because it's possible that someone has already created a wrapper for the same third party executable you are looking for. Consider announcing your tool project on galaxy-dev to see if anyone has already created a wrapper.

Wrapping tools with subcommands

Subcommands can be represented as separate tools or a conditional. Separate tools are preferable in particular if the different subcommands need different amounts of resources (because this simplifies the allocation of resources for Galaxy admins).

Tool versions

Tool versions are mandatory to enable reproducibility. Version is an attribute of the XML tool element, e.g.

```
<tool id="rgTF" name="Tool Factory" version="1.11">
```

and should be incremented with each change of the wrapper that is released to the Tool Shed (except for cosmetic modifications).

The value should follow the PEP 440 specification.

If the Galaxy tool is a wrapper for an underlying tool, we recommend to:

- define a `@TOOL_VERSION@` macro token, which you should also re-use in the corresponding `<requirement>` element;
- optionally, define a `@VERSION_SUFFIX@` macro token, which may be placed either in the tool wrapper or in a shared macro file. This should be set to an integer number:
  - starting at 0 for the first wrapper release of each version of the underlying tool;
  - to be increased by 1 whenever you update the wrapper without changing the underlying `@TOOL_VERSION@`.
- set the tool `version` attribute to `@TOOL_VERSION@+galaxy@VERSION_SUFFIX@`. If it is preferred not to use a `@VERSION_SUFFIX@` token (e.g. to allow bumping the version only for a specific tool in a suite), the tool `version` attribute should be simply set to `@TOOL_VERSION@+galaxyN`, where N is an integer following the same rules as `@VERSION_SUFFIX@`.

If instead the Galaxy tool cannot be identified with a single underlying tool, the `+galaxy@VERSION_SUFFIX@` local version identifier should be omitted, and any version value can be used, as long as it respects the PEP 440 specification.

For tools whose wrapper version is (for historical reasons) already greater than the version of the underlying tool, only the minor version number shall be increased if this is likely to bring the two version in sync in a reasonable time.

Tool ids

Should be meaningful and unique also in a larger context. If your tool is called `grep` try to prefix that name with something meaningful. Objective is to make it easier for Galaxy admins to identify a tool based on the short ID. Otherwise they would need to use the long `toolshed/xx/` id.

Some simple rules for generating tool IDs:

- Tool IDs should contain only `[a-z0-9_-]`.
- Multiple words should be separated by underscore or dashes
- Suite tools should prefix their ids with the suite name. E.g. `bedtools_*`

Tool Names

Names are important! Names are how users and admins find your tools. Names should strive to be unique within a suite of tools, and may wish to include the suite name if it is a well known suite. Some instructional examples:

- Cufflinks, Cuffdiff, Cuffmerge are in a suite together.
- The vsearch suite contains tools with names like "VSearch Alignment", "VSearch Clustering", etc.

In the cufflinks example, everyone knows the functionality of the cufflinks command, and can easily guess as the use of a tool named "cuffdiff" in their tool panel.

With VSearch however, a tool named "Alignment" would not be useful, as users would have a hard time finding it and gathering context about its functionality. With the VSearch prefix, once a user learns what one VSearch tool does, they can quickly apply that to the other available VSearch tools.

Tool profile

Tools should define a recent profile, i.e. a profile not older than 1 year.

Tool Descriptions

Tool names are not your only tool for making your tool discoverable to end users, and conveying information regarding the functionality of said tool. Tool descriptions are displayed directly after the tool name and generally conform to a "sentence" like structure.

- `bowtie2` is a short read aligner
- `Cuffmerge` merges together several Cufflinks assemblies
- `NCBI BLAST+ database info` shows BLAST database information from blastdbcmd

In the above examples the tool name is rendered in fixed width text, and the rest is the tool description.

Tool cross-references (bio.tools)

It is important to cross-link Galaxy tools to best practice metadata in registries and catalogs such as bio.tools.

bio.tools is a global portal for bioinformatics resources that helps researchers to find, understand, compare and select resources suitable for their work.

You can link a Galaxy tool to its bio.tools entry by adding a `xref` snippet to the tool wrapper: i.e. modifying the `*.xml` file. Below is an example `xml` for Racon

Please reference only the tool which is wrapped, not the dependencies.

If a bio.tools entry does not exist, you should add an entry as follows:

- Create an account on bio.tools.
- Add a new tool, completing as much metadata as possible / available.
- Ensure that you add at least one EDAM `Topic` and one EDAM `Operation` to the entry (see below).

Tool annotations (EDAM)

The bio.tools registry makes use of the EDAM ontology to annotate tools with `Data`, `Format`, `Operation` and `Topic` terms.

EDAM terms are used to give a description of the tool's scientific domain and the functionalities it provides. The EDAM browser makes the process of picking EDAM terms much easier.

Once a bio.tools identifier is added as an `xref` in the tool wrapper, the easiest way to improve the available EDAM annotations is to update the bio.tools entry metadata.

Alternatively, you can add EDAM annotations directly to a Galaxy tool by adding the edam_topics and edam_operations elements to the tool XML file. This approach is not recommended though, as it risks duplicating the tool annotation work in multiple places.

When picking EDAM terms, avoid "root terms" such as `Topic` and `Operation`, and pick the most specific terms available. If you feel like the terms required to describe the tool are missing, do not hesitate to ask for new terms.

More detailed guidelines to pick EDAM terms are also available in the [bio.tools curators guide](https://biotools.readthedocs.io/en/latest/curators_guide.html#edamannotations) and the [EDAM ontology users guide](https://edamontologydocs.readthedocs.io/en/latest/users_guide.html#picking-concepts).

Parameter name, argument and help

The `argument` attribute of `<param>` should include the long form of the underlying tool parameter, e.g. `argument="--max"`. This is automatically displayed inside the parameter help and is useful to give the user the chance to go to the original documentation and map the Galaxy UI element to the actual parameter. It also makes debugging easier if the user is talking to non-Galaxy developers.

When `argument` is specified, the `name` attribute becomes optional and, if not included, is derived from `argument` by stripping any leading dashes and replacing internal dashes by underscores (the later since release 19.09). This derived name can be used inside the `<command>` element to refer to the parameter value as you would normally do with the `name` attribute. Note that if the automatically generated name violates the rules for valid Cheetah placeholders (i.e. consist of alphanumeric characters or underscore and must not start with a digit) you should specify a valid `name` attribute for the parameter.

Tests

All Galaxy Tools should include functional tests. In their simplest form, you provide sample input files and expected output files for given parameter values. Where the output file is not entirely reproducible you can make assertions about the output file contents.

Testing error conditions is also important. Recent development now allows tests say if the test should fail, and to make assertions about the tool's stdout and stderr text (e.g. check expected summary text or warning messages appear). See planemo docs for more information.

When tools contain output filters, tests should be included that verify this filtering occurs. See planemo docs for more information.

Data parameters

If a compressed version of a datatype (e.g. `fasta.gz` or `fastqsanger.bz2`) is supported by Galaxy, then data parameters should accept both the compressed and the uncompressed datatypes. The tool should internally decompress the dataset if the underlying tool cannot handle the compressed formats natively.

Booleans

`truevalue` and `falsevalue` attributes of `<param>` should contain the underlying tool parameter. This makes it really easy to reference the param name in the Cheetah `<command>` section.

```
<command>
...
$strict
...
</command>
<inputs>
    ...
    <param name="strict" truevalue="--enable-strict" falsevalue="">
```

Boolean should not be used as a conditional for other options. For dynamic options, please use a `select` input type as described in the Dynamic Options section below.

Dynamic Options

Options that are conditionally hidden (using the `<conditional>` element) should use a `select` param type and not a `boolean`. The user may not expect a boolean checkbox to change the content of a form.

To create an "Advanced options" section which is normally hidden and the user can expand, a `<section>` element can be used instead of a `<conditional>`. Beware that parameters inside a hidden section still have a value set, which is used when creating the job command, while in a "closed" conditional the non-visible parameters don't have a value.

Command tag

The command tag is one of the most important parts of the tool, next to the user-facing options. It should be highly legible.

Command Formatting

The command tag should be started and finished by a CDATA tag, allowing direct use of characters like the ampersand (&) without needing XML escaping (`&`).

```
<![CDATA[ your lines of Cheetah here ]]>
```

Wikipedia has more on CDATA

All Cheetah variables for text parameters, input and output files must be single-quoted, e.g. `'${var_name}'`.

For composite datatypes the recommended attribute to access the associated directory name differs for inputs (e.g. `$input.extra_files_path`) versus outputs (e.g. `$output.files_path`). This difference is historical, and it is hoped this will be harmonised in a future Galaxy release.

If you need to execute more than one shell command, concatenate them with a double ampersand (&&), so that an error in a command will abort the execution of the following ones.

Exit Code Detection

Unless the tool has special requirements, you should take advantage of the exit code detection provided by Galaxy, in lieu of using the `<stdio/>` tags. This can be done by adding a `detect_errors` tag to your `<command />` block like so:

```
<command detect_errors="aggressive">
...
</command>
```

This will automatically fail the tool if the exit code is non-zero, or if the phrases `error:` or `exception:` appear in STDERR.

Help tag

The help tag should be started and finished by a CDATA tag.

```
<![CDATA[ your lines of restructuredText here ]]>
```

http://en.wikipedia.org/wiki/CDATA

Inside the help tag you should describe the functionality of your tool. The help tag is to the `help=""` attribute as a man page is to the `--help` flag. The help tag should cover the tools functionality, use cases, and even known issues in detail. The help tag is a good place to provide examples of how to run the tool and discuss specific subcases that your users might be interested in.

Including Images

If you have produced images detailing how your tool works (e.g. bedtools), it might be nice for those images to be included in the Galaxy tool documentation!

Images should be placed in a subdirectory, `./static/images/`, and referenced in your tool help as `.. image:: my-picture.png`. This can be seen in the IUC's wrappers, such as the one for the bedtools slop command.

Generating Indices

Occasionally data needs to be indexed (e.g. bam, fasta) files. When data is indexed, those indices should be generated in the current working directory rather than alongside the input dataset. This is part of the tool contract, you can read from your inputs, but only write to your outputs and CWD.

It's convenient to do something like:

```
ln -sfn "${input_fasta}" tmp.fa;
```

before data processing in order to be able to easily generate the indices without attempting to write to a (possibly) read-only data source.

Datatypes

For now, the recommended practice is to push new datatypes to the Galaxy repository.

Data Managers

TODO

Coding Style

- 4 spaces indent
- Order of XML elements:
  - description
  - macros
  - edam_topics
  - edam_operations
  - xrefs
  - [parallelism]
  - requirements
  - [code]
  - stdio
  - version_command
  - command
  - environment_variables

  - configfiles
  - inputs
  - request_param_translation
  - outputs
  - tests
  - help
  - citations
- Cheetah code should also be indented and mainly PEP8 conformant
- XML elements should normally have all attributes on a single line for easier searchability, but for large XML elements the `label` and `help` attributes can be on a new line.
- param names should be readable and understandable, e.g. using the long option name of the wrapped tool
- Order of parameter attributes:
  - name
  - argument
  - type
  - format
  - min | truevalue
  - max | falsevalue
  - value | checked
  - optional
  - label
  - help
- Python code should be Python3-compatible and PEP8 conformant. Imports should follow the smarkets style.

1.3.2 ToolShed Readiness Checklist

The process from writing a tool to getting it into a ToolShed can be long and arduous and confusing. This checklist should assist in making sure you have done everything required for a great, easy to use Galaxy Tool!

Before ToolShed

- A GitHub repository should exist for your wrappers, either one you own, or perhaps you are contributing to the IUC's repository.
- A tool directory should exist for the specific set of tools or related functionality you are wrapping.
- Check Bioconda for available packages required for the tool you are wrapping. If they do not exist, you may need to create them. The IUC will be happy to help you with doing this.
- Planemo should be installed (`pip install -U planemo`)
- You will need to have credentials to access your ToolShed (either the Main ToolShed, or your local Galactic ToolShed).

Creating the Tool Wrapper (XML File)

- Review the IUC's Best Practices for Tools.
- Consult the Galaxy Tool XML File schema.
- Create your tool wrapper with a command like `planemo tool_init --id 'tool_name' --name 'Tool description'`.
- Alternatively, you could copy and modify an existing IUC wrapper.
- Give your tool an appropriate `ID` and `name` by consulting the IUC's Best Practices for Tools. The `ID` is usually the same as the name of the tool XML file and directory.
- Define a Tool Version for the wrapper. If it is the first wrapper, is recommended to use the same version as the tool in the requirement.
- Add a short tool Description.
- Fill in the Requirements section with the conda package name and version number for the tool and its dependencies.
- Add the Version Command that specifies the command to get the tool's version.
- Add the Command section. The command to run the tool must be within CDATA tags, written in Cheetah and conform to PEP 8. You should add Exit Code detection and **use single quotes** for all Input and Output parameters of type `data`, `data_collection` and `text`.
- Supply at least one Input with a description of parameters. Add Validators to user input fields.
- Supply at least one Output with a description of parameters. For Output that is optionally created, use Filters.
- Supply at least one Test. The primary output is a good choice for testing. Don't forget the use of `sim_size` if variable data is included.
- Add a Help section written in valid reStructuredText within CDATA tags.
- Add a Citation section with a citation for the tool, preferably a DOI.
- **If your tool uses built-in data:**
  - Provide the comment-only `tool-data/data_table_name.loc.sample` file
  - Provide the comment-only `tool_data_table_conf.xml.sample` file
- Check that the XML elements and parameters attributes are in the Order specified in the Best Practices.
- If you have a collection of related tools you can try to avoid duplicating XML by using a Macros XML file
- Use 4 spaces for indentation.

Testing Your Tool

- Fill the `test-data` directory with at least one input file and the expected output file.
- It is strongly encouraged that you use small test data sets, ideally under 1 Mb. Every Galaxy instance that downloads your tool will have to download an entire copy of the test data. If the sum of your test-data files is larger than that, consider use of `contains` and test for a small subset of the output, see the CWPair2 example.
- **If your tool uses tool-data:**
  - Provide a `tool_data_table_conf.xml.test` file, which is an uncommented version of `tool_data_table_conf.xml.sample` containing the path to the loc file for testing: `<file path="${__HERE__}/test-data/data_table_name.loc" />` (Please note the use of `${__HERE__}` to indicate the directory where the tool is).

  - Provide the .loc file: test-data/data_table_name.loc
  - For a good example of how to test parameters from data tables, please see the Bowtie example.

- Check your tool XML with planemo lint.
- Run functional tool tests in a local Galaxy with planemo test.
- Serve the tool on a local Galaxy instance for manual verification that everything looks as expected with the planemo serve command.

Uploading Your Tool to a ToolShed

- Ensure you have a .shed.yml file with the appropriate contents.
- Check the .shed.yml with planemo shed_lint.
- Create the remote repository with planemo shed_create --shed_target [toolshed|your_local_shed:9000].

Adding Your Tool to the IUC Repository

- Create an issue on IUC GitHub, tracking your progress and ensuring that no one else is working on the same tool.
- Fork IUC GitHub on your GitHub account.
- Clone the corresponding repository git clone https://github.com/<YOUR_NAME>/tools-iuc tools-iuc
- Within that folder, create a corresponding branch with git checkout -b $branch_name. You might name it after the tool.
- After you have tested your tool and are completely happy with it (per previous sections of this document), add your tool and all associated data, then Commit the changes with git commit -m "I changed X, Y, and Z". Finally push your changes to github with git push origin $branch_name.
- Go to the IUC's Repository and click on 'Compare & Pull Request'.
- Add a comment describing what the tool and any extra information that might be needed (E.g. "I had some trouble with the data tables, can someone please double check them").
- The IUC will review your tool for inclusion.
- Note that any Python code submitted to IUC must conform to PEP 8, in order to pass the flake8 Travis CI testing.

1.4 Packages

1.4.1 Packages

Before you start writing a new tool please search the Main Tool Shed (MTS) and the Test Tool Shed (TTS) because it's possible that someone has already created an installer for the same third party executable you are looking for. Consider announcing your packaging project on galaxy-dev to see if anyone has already created a wrapper.

Packaging software is something of a more advanced topic, and due to the complexities of the syntax, somewhat harder to validate.

Downloads

Packages generally must download one or more files from the internet in order to function. We require checksums on all of our package downloads from multiple reasons:

- Download integrity.
- Insecure transport methods, like `http://` and `ftp://`
- The packages come from the untrusted internet, we don't know if anyone has modified the software in transit. This software is installed directly to large university clusters. We must make an effort to ensure that what is being installed is what the user actually asked for, and not a version of `bowtie` that has been modified unexpectedly.

The checksums take the form of sha256sums attached as attributes to `<action type="download_by_url">` and other elements, e.g.

```
<action type="download_by_url" sha256sum="ab060325...">
    http://mbio-serv2.mbioekol.lu.se/ARAGORN/Downloads/aragorn1.2.36.tgz
</action>
```

This XML snippet will cause the file `aragorn1.2.36.tgz` to be downloaded, and to be validated. If the sha256sums match, then the package installs. Otherwise, it fails immediately.

1.5 Repository Layout

1.5.1 Github Repositories

Most tool developers are on GitHub, and have chosen to lay out their repositories in a structure similar to the following:

```
tools-iuc/
├── data_managers
│   └── data_manager_NAME/...
├── LICENSE
├── packages/
│   └── package_NAME_VERSION/tool_dependencies.xml
├── README.rst
├── suites/
│   └── suite_name/...
└── tools/
    └── NAME
        ├── macros.xml
        ├── my_tool.xml
        ├── CHANGELOG.md
        ├── other_tool.xml
        ├── test-data/
        └── tool_dependencies.xml
```

The highest level directory contains only a few folders for the major types of Galaxy repositories; tools, packages, data managers, and sometimes visualizations and datatypes.

ToolShed Repositories

A Github repository may correspond to any number of published ToolShed repositories.

Every unrestricted tool shed repository should contain a README file - named either `README` or `README.txt` (if plain text) or `README.rst` (if reStructuredText). A reStructuredText `README.rst` is generally preferred. For a good example of such a file - please see Peter Cock's NCBI Blast+ Suite's README.rst.

The Tool Shed recognizes many more types of README files than this - but these are not encouraged and may be deprecated in the future. Markdown is not supported by the Tool Shed at this time and so `README.md` are not recognized at all.

Package Repositories

These may only contain a `tool_dependencies.xml` file

Suites

The Toolshed offers the concept of a suite which is simply a meta-package listing several other packages. For example the `suite_hmmer_3` provides a package that depends on all of the individual `hmmer_.*` packages, defined by a `repository_dependencies.xml` file:

```
<repositories description="HMMER v3 HMM based sequence alignment and database search␣
↪tools">
  <repository changeset_revision="ddda6eae7b23" name="hmmer_hmmemit" owner="iuc"␣
↪toolshed="https://testtoolshed.g2.bx.psu.edu" />
  <repository changeset_revision="5ec773098cb9" name="hmmer_hmmconvert" owner="iuc"␣
↪toolshed="https://testtoolshed.g2.bx.psu.edu" />
  ...
```

Manually curated suites are most commonly used to package together related pieces of software by different groups, when that functionality all serves a common purpose.

Suites should NOT be used for a single set of highly related tools from the same group, like the hmmer example above, or bedtools. Instead, a suite can be automatically created for those sets of tools by Planemo.

Tools

Tool often contain:

- Tool XML files
- `macros.xml` file for use in keeping tools DRY
- `test-data/` directory, because all tools need test data
- `tool-data/` directory, for things like `*.loc` files
- `tool_dependencies.xml` file for specifying associated packages
- `CHANGELOG.md` file for tracking the history of features over time in your tool

1.6 .shed.yml

1.6.1 Toolshed Yaml File

The `.shed.yml` file provides a way for developers using the awesome planemo to easily push their tools to toolshed repositories.

```yaml
name: package_aragorn_1_2_36
owner: iuc
description: Contains a tool dependency definition that downloads and extract version
  1.2.36 of Aragorn.
homepage_url: http://mbio-serv2.mbioekol.lu.se/ARAGORN/
long_description: |
    ARAGORN, tRNA (and tmRNA) detection.

    http://www.ncbi.nlm.nih.gov/pubmed/14704338
remote_repository_url: https://github.com/galaxyproject/tools-iuc/tree/master/packages/
↪package_aragorn_1_2_36
type: tool_dependency_definition
categories:
- Tool Dependency Packages
```

Parameter	Value
name	This is the package or tool's name. It is usually the name of the folder that contains the `.shed.yml` file. This should be package_$name_$version for packages, and $name for tools
owner	Your toolshed username
description	A short description of the package or tool set
home-page_url	This value is currently under debate, but we recommend reading over #1.
long_descript	A longer README type description of the package, as tool dependencies do no currently support README files.
re-mote_reposit	This should be the path to the folder in your github repository, on the branch you create releases from (usually master). This will eventually be used with the toolshed for update hooks.
type	The repository type, one of `unrestricted`, `tool_dependency_definition`, or `repository_suite_definition`
categories	Toolshed categories that are relevant to the tool or package.

A note on the `name` attribute: as periods are disallowed in repository names, we recommend replacing periods in the version number with underscores.

Repository Type	Recommended Name	Examples
Data Managers	`data_manager_$name`	`data_manager_bowtie2`
Packages	`package_$name_$version`	`package_aragorn_1_2_36`
Tool Suites	`suite_$name`	`suite_samtools`
Tools	`$name`	`stringtie`, `bedtools`

Advanced Parameters

> Currently there exists a tension between what is best for developers (storing all tools in a single repository - e.g. `ncbi_blast_plus` or `bedtools`) and what is best for Galaxy users (storing a single repository per tool and collecting them together with a suite - e.g. `samtools` or `gatk`).

Thus a number of advanced parameters were added for helping developers manage suites of tools.

```
auto_tool_repositories:
  name_template: "{{ tool_id }}"
  description_template: "Wrapper for samtools application {{ tool_name }}."
suite:
  name: "suite_samtools_1_2"
  description: "A suite of Galaxy tools designed to work with version 1.2 of the␣
↪SAMtools package."
  long_description: |
    SAM (Sequence Alignment/Map) format is a generic format for storing large nucleotide␣
↪sequence
    alignments.   This repository suite associates selected repositories containing␣
↪Galaxy utilities that require
    version 1.2 of the SAMTools package.  These associated Galaxy utilities consist of a␣
↪Galaxy Data
    Manager contained in the repository named data_manager_sam_fasta_index_builder and␣
↪Galaxy tools
    contained in several separate repositories.
```

This example assumes the `.shed.yml` file is placed in a "flat" directory with each samtools tool wrapper. Planemo will create and update repositories for each individual tool given the specified templates in auto_tool_repositories. The suite key here will auto-generate a suite repository for all of these tools and will automatically created the corresponding repository_dependencies.xml to populate it with (this is generated during shed_upload and never needs to exist in your repository).

Again this example is admittedly idealized, but if `auto_tool_repositories` is not specified, a repositories list can be specified instead. There are some examples of this in the planemo's test data:

- This .shed.yml is a simple example of specifying custom repositories for individual tools.
- This demonstrates complex inclusions files from sub-directories and renaming.

The test data also includes some more advanced usages of the suite key as well - specifically using it without auto_tool_repositories as a generic replacement for repository_dependencies.xml and adding additional dependent repositories in addition to the ones defined by the .shed.yml file.

Shed Upload Includes/Excludes

Sometimes it is of interest to have shared data in a single directory, and then to `include` that when needed. A good example of this are the blast wrappers which take advantage of the feature in order to share test data amongst a number of directories which all need the data.

```
include:
- strip_components: 2
  source:
  - ../../test-data/blastdb.loc
  - ../../test-data/blastdb_d.loc
  - ../../test-data/blastdb_p.loc
  - ../../test-data/blastn_arabidopsis.extended.tabular
```

This snippet informs planemo that it should include specific datasets from `../../test-data` and that as part of the include process it should strip the first two path components.

The `exclude` functionality works similarly, just specify a list of paths you wish to exclude:

```
exclude:
  - test-data/my-gigantic-test-dataset.fastq
```

1.7 Repository Management

1.7.1 Repository Management on GitHub and the ToolShed

Manual Management

For smaller groups, manual management of your repositories may be sufficient. The IUC strongly recommends the use of Planemo for uploading tools. This is as simple as defining your `~/.planemo.yml` and running `planemo shed_upload -m "We added new feature X" path/to/my/repo`.

Automated Management

The IUC has developed github actions configurations in order to assist in continually synchronizing your GitHub repository with the toolshed components. You can view these in .github/workflows in the IUC's Github Repo

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search