TotalPerspectiveVortex Documentation

Release 1.2.0

Galaxy and GVL projects

Jun 15, 2022

CONTENTS:

Total Perspective Vortex

TotalPerspectiveVortex (TPV) provides an installable set of dynamic rules for the Galaxy application that can route entities (Tools, Users, Roles) to appropriate destinations based on a configurable yaml file. The aim of TPV is to build on and unify previous efforts, such as Dynamic Tool Destinations, the Job Router and Sorting Hat, into a configurable set of rules that that can be extended arbitrarily with custom Python logic.

TPV provides a dynamic rule that can be plugged into Galaxy via `job_conf.xml`. The dynamic rule will also have an associated configuration file, that maps entities (tools, users, roles) to specific destination through a flexible tagging system. Destinations can have arbitrary tags defined, and each entity can express a preference or aversion to specific tags. Based on this tagging, jobs are routed to the most appropriate destination. In addition, admins can also plugin arbitrary python based rules for making more complex decisions, as well as custom ranking functions for choosing between matching destinations. For example, a ranking function could query influx metrics to determine the least loaded destination, and route jobs there, providing a basic form of "metascheduling" functionality.

CHAPTER ONE

GETTING STARTED

1. *pip install total-perspective-vortex* into Galaxy's python virtual environment
2. Configure Galaxy to use TPV's dynamic destination rule
3. Create the TPV job mapping yaml file, indicating job routing preferences
4. Submit jobs as usual

1.1 TPV by example

1.1.1 Simple configuration

The simplest possible example of a useful TPV config might look like the following:

```yaml
tools:
  https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
    cores: 12
    mem: cores * 4
    gpus: 1

destinations:
 slurm:
   cores: 16
   mem: 64
   gpus: 2
 general_pulsar_1:
   cores: 8
   mem: 32
   gpus: 1
```

Here, we define one tool and its resource requirements, the destinations available, and the total resources available at each destination (optional). The tools are matched by tool id, and can be a regular expression. Note how resource requirements can also be computed as python expressions. If resource requirements are defined at the destination, TPV will check whether the job will fit. For example, hisat2 will not schedule on *general_pulsar_1* as it has insufficient cores. If resource requirements are omitted in the tool or destination, it is considered a match.

1.1.2 Default inheritance

Inheritance provides a mechanism for an entity to inherit properties from another entity, reducing repetition.

```yaml
global:
  default_inherits: default

tools:
  default:
    cores: 2
    mem: 4
    params:
      nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
↪{mem*1024}"
  https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/2.1.0+galaxy7:
    cores: 12
    mem: cores * 4
    gpus: 1
```

The *global* section is used to define global TPV properties. The *default_inherits* property defines a "base class" for all tools to inherit from.

In this example, if the *bwa* tool is executed, it will match the *default* tool, as there are no other matches, thus inheriting its resource requirements. The hisat2 tool will also inherit these defaults, but is explicitly overriding cores, mem and gpus. It will inherit the *nativeSpecification* param.

1.1.3 Explicit inheritance

Explicit inheritance provides a mechanism for exerting greater control over the inheritance chain.

```yaml
global:
  default_inherits: default

tools:
  default:
    cores: 2
    mem: 4
    params:
      nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
↪{mem*1024}"
  https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
    cores: 12
    mem: cores * 4
    gpus: 1
  .*minimap2.*:
    inherits: https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
    cores: 8
    gpus: 0
```

In this example, the minimap2 tool explicitly inherits requirements from the hisat2 tool, which in turn inherits the default tool. There is no limit to how deep the inheritance hierarchy can be.

1.1.4 Scheduling tags

Scheduling tags provide a means by which to control how entities match up, and can be used to route jobs to preferred destinations, or to explicitly control which users can execute which tools, and where.

```yaml
tools:
  default:
    cores: 2
    mem: 4
    params:
      nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
↪{mem*1024}"
    scheduling:
      reject:
        - offline
  https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
    cores: 4
    mem: cores * 4
    gpus: 1
    scheduling:
      require:
      prefer:
        - highmem
      accept:
      reject:
  https://toolshed.g2.bx.psu.edu/repos/iuc/minimap2/.*:
    cores: 4
    mem: cores * 4
    gpus: 1
    scheduling:
      require:
        - highmem

destinations:
 slurm:
   cores: 16
   mem: 64
   gpus: 2
   scheduling:
      prefer:
        - general

 general_pulsar_1:
   cores: 8
   mem: 32
   gpus: 1
   scheduling:
      prefer:
        - highmem
      reject:
        - offline
```

In this example, all tools reject destinations marked as offline. The hisat2 tool expresses a preference for highmem, and inherits the rejection of offline tags. Inheritance can be used to override scheduling tags. For example, the minimap2 tool inherits hisat2, but now requires a highmem tag, instead of merely preferring it.

The destinations themselves can be tagged in similar ways. In this case, the *general_pulsar_1* destination also prefers the highmem tag, and thus, the hisat2 tool would schedule there. However, *general_pulsar_1* also rejects the offline tag, and therefore, the hisat2 tool cannot schedule there. Therefore, it schedules on the only available destination, which is slurm.

The minimap2 tool meanwhile requires highmem, but rejects offline tags, which leaves it nowhere to schedule. This results in a JobMappingException being thrown.

A full table of how scheduling tags match up can be found in the Scheduling section.

1.1.5 Rules

Rules provide a means by which to conditionally change entity requirements.

```yaml
tools:
  default:
    cores: 2
    mem: cores * 3
    rules:
      - id: my_overridable_rule
        if: input_size < 5
        fail: We don't run piddling datasets of {input_size}GB
  bwa:
    scheduling:
      require:
        - pulsar
    rules:
      - id: my_overridable_rule
        if: input_size < 1
        fail: We don't run piddling datasets
      - if: input_size <= 10
        cores: 4
        mem: cores * 4
        execute: |
           from galaxy.jobs.mapper import JobNotReadyException
           raise JobNotReadyException()
      - if: input_size > 10 and input_size < 20
        scheduling:
          require:
            - highmem
      - if: input_size >= 20
        fail: Input size: {input_size} is too large shouldn't run
```

The if clause can contain arbitrary python code, including multi-line python code. The only requirement is that the last statement in the code block must evaluate to a boolean value. In this example, the *input_size* variable is an automatically available contextual variable which is computed by totalling the sizes of all inputs to the job. Additional available variables include app, job, tool, and user.

If the rule matches, the properties of the rule override the properties of the tool. For example, if the input_size is 15, the bwa tool will require both pulsar and highmem tags.

Rules can be overridden by giving them an id. For example, the default for all tools is to reject input sizes < 5 by using the *my_overridable_rule* rule. We override that for the bwa tool by specifically referring to the inherited rule by id. If no id is specified, an id is auto-generated and no longer overridable.

Note the use of the {input_size} variable in the fail message. The general rule is that all non-string expressions are evaluated as python code blocks, while string variables are evaluated as python f-strings.

The execute block can be used to create arbitrary side-effects if a rule matches. The return value of an execute block is ignored.

1.1.6 User and Role Handling

Scheduling rules can also be expressed for users and roles.

```yaml
tools:
  default:
    scheduling:
      require: []
      prefer:
        - general
      accept:
      reject:
        - pulsar
    rules: []
  dangerous_interactive_tool:
    cores: 8
    mem: 8
    scheduling:
      require:
        - authorize_dangerous_tool
users:
  default:
    scheduling:
      reject:
        - authorize_dangerous_tool
  fairycake@vortex.org:
    cores: 4
    mem: 16
    scheduling:
      accept:
        - authorize_dangerous_tool
      prefer:
        - highmem

roles:
  training.*:
    cores: 5
    mem: 7
    scheduling:
      reject:
        - pulsar
```

In this example, if user *fairycake@vortex.org* attempts to dispatch a *dangerous_interactive_tool* job, the requirements for both entities would be combined. Most requirements would simply be merged, such as env vars and job params. However, when combining gpus, cores and mem, the lower of the two values are used. In this case, the combined entity would have a core value of 4 and a mem value of 8. This allows training users for example, to be forced to use a lower number of cores than usual.

In addition, for these entities to be combined, the scheduling tags must also be compatible. In this instance the *dangerous_interactive_tool* requires the *authorize_dangerous_tool* tag, which all users by default reject. Therefore, most users cannot run this tool by default. However, *fairycake@vortex.org* overrides that and accepts the *authorize_dangerous_tool* allowing only that user to run the dangerous tool.

Roles can be matched in this exact way. Rules can also be defined at the user and role level.

1.1.7 Metascheduling

Custom rank functions can be used to implement metascheduling capabilities. A rank function is used to select the best matching destination from a list of matching destination. If no rank function is provided, the default rank function simply chooses the most preferred destination out of the available destinations.

When more sophisticated control over scheduling is required, a rank function can be implemented through custom python code.

```yaml
tools:
  default:
    cores: 2
    mem: 8
    rank: |
      import requests

      params = {
        'pretty': 'true',
        'db': 'pulsar-test',
        'q': 'SELECT last("percent_allocated") from "sinfo" group by "host"'
      }

      try:
        response = requests.get('http://stats.genome.edu.au:8086/query', params=params)
        data = response.json()
        cpu_by_destination = {s['tags']['host']:s['values'][0][1] for s in data.get('results')[0].get('series', [])}
        # sort by destination preference, and then by cpu usage
        candidate_destinations.sort(key=lambda d: (-1 * d.score(entity), cpu_by_destination.get(d.id)))
        final_destinations = candidate_destinations
      except Exception:
        log.exception("An error occurred while querying influxdb. Using a weighted random candidate destination")
        final_destinations = helpers.weighted_random_sampling(candidate_destinations)
      final_destinations
```

In this example, the rank function queries a remote influx database to find the least loaded destination, The matching destinations are available to the rank function through the *candidate_destinations* contextual variable. Therefore, in this example, the candidate destinations are first sorted by the best matching destination (score is the default ranking function), and then sorted by CPU usage per destination, obtained from the influxdb query.

Note that the final statement in the rank function must be the list of sorted destinations.

1.1.8 Custom contexts

In addition to the automatically provided context variables (see *Concepts and Organisation*), TPV allows you to define arbitrary custom variables, which are then available whenever an expression is evaluated. Contexts can be defined both globally or at the level of each entity, with entity level context variables overriding global ones.

```yaml
global:
  default_inherits: default
  context:
    ABSOLUTE_FILE_SIZE_LIMIT: 100
    large_file_size: 10
    _a_protected_var: "some value"

tools:
  default:
    context:
      additional_spec: --my-custom-param
    cores: 2
    mem: 4
    params:
      nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
↪{mem*1024} {additional_spec}"
     rules:
      - if: input_size >= ABSOLUTE_FILE_SIZE_LIMIT
        fail: Job input: {input_size} exceeds absolute limit of: {ABSOLUTE_FILE_SIZE_
↪LIMIT}
      - if: input_size > large_file_size
        cores: 10

  https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/2.1.0+galaxy7:
    context:
      large_file_size: 20
      additional_spec: --overridden-param
    mem: cores * 4
    gpus: 1
```

In this example, three global context variables are defined, which are made available to all entities. Variable names follow Python conventions, where all uppercase variables indicate constants that cannot be overridden. Lower case indicates a public variable that can be overridden and changed, even across multiple TPV config files. An underscore indicates a protected variable that can be overridden within the same file, but not across files.

Additional, the tool defaults section defines an additional context variable named ‘additional_spec`, which is only available to inheriting tools.

If we were to dispatch a job, say bwa, with an input_size of 15, the large file rule in the defaults section would kick in, and the number of cores would be set to 10. If we were to dispatch a hisat2 job with the same input size however, the large_file_size rule would not kick in, as it has been overridden to 20. The main takeaway from this example is that variables are bound late, and therefore, rules and params can be crafted to allow inheriting tools to conveniently override values, even across files. While this capability can be powerful, it needs to be treated with the same care as any global variable in a programming language.

1.1.9 Multiple matches

If multiple regular expressions match, the matches are applied in order of appearance. Therefore, the convention is to specify more general rule matches first, and more specific matches later. This matching also applies across multiple TPV config files, again based on order of appearance.

```yaml
tools:
  default:
    cores: 2
    mem: 4
    params:
      nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
↪{mem*1024}"

  https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/*:
    mem: cores * 4
    gpus: 1

  https://toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/2.1.0+galaxy7:
    env:
       MY_ADDITIONAL_FLAG: "test"
```

In this example, dispatching a hisat2 job would result in a mem value of 8, with 1 gpu. However, dispatching the specific version of *2.1.0+galaxy7* would result in the additional env variable, with mem remaining at 8.

1.1.10 Job Resubmission

TPV has explict support for job resubmissions, so that advanced control over job resubmission is possible.

```yaml
tools:
  default:
    cores: 2
    mem: 4 * int(job.destination_params.get('SCALING_FACTOR', 1)) if job.destination_
↪params else 1
    params:
      SCALING_FACTOR: "{2 * int(job.destination_params.get('SCALING_FACTOR', 2)) if job.
↪destination_params else 2}"
    resubmit:
      with_more_mem_on_failure:
        condition: memory_limit_reached and attempt <= 3
        destination: tpv_dispatcher
```

In this example, we have defined a resubmission handler that resubmits the job if the memory limited is reached. Note that the resubmit section looks exactly the same as Galaxy's, except that it follows a dictionary structure instead of being a list. Refer to the Galaxy job configuration docs for more information on resubmit handlers. One twist in this example is that we automatically increase the amount of memory provided to the job on each resubmission. This is done by setting the SCALING_FACTOR param, which is a custom parameter which we have chosen for this example, that we increase on each resubmission. Since each resubmission's destination is TPV, the param is re-evaluated on each resubmission, and scaled accordingly. The memory is allocated based on the scaling factor, which therefore, also scales accordingly.

1.2 Concepts and Organisation

1.2.1 Object types

Conceptually, TPV consists of the following types of objects.

1. Entities - An entity is anything that will be considered for scheduling by TPV. Entities include Tools, Users, Groups, Rules and Destinations. All entities have some common properties (id, cores, mem, env, params, scheduling tags).

2. Scheduling Tags - Entities can have scheduling tags defined on them that determine which entities match up, and which destination they can schedule on. Tags fall into one of four categories, (required, preferred, accepted, rejected), ranging from indicating a requirement for a particular entity, to indicating complete aversion.

3. Loader - The loader is responsible for loading entity definitions from a config file. The loader will parse and validate entity definitions, including compiling python expressions, and processing inheritance, to produce a list of entities suitable for mapping. The loader is also capable of loading config files from multiple sources, including https urls.

4. Mapper - The mapper is responsible for routing a Galaxy job to its destination, based on the current user, tool and job that must be scheduled. The mapper will respect the scheduling constraints expressed by the loaded entities.

1.2.2 Operations

When a mapper routes jobs to a destination, it does so by applying 5 basic operations on entities.

1. Inherit

The inherit operation enables an entity to inherit the properties of another entity of the same type, and to override any required properties. While a Tool can inherit another tool, which can in-turn inherit yet another tool, it cannot inherit a User, as it's a different entity type. It is also possibly to globally define a *default_inherits* field, which is the entity that all entity name that all entities will inherit from should they not have an *inherits* tag explicitly defined. Inheritance is generally processed at load time by the *Loader*, so that there's no cost at runtime. However, the *Mapper* will process default inheritance, should the user, role or tool that is being dispatched does not have an entry in the entities list.

When inheriting scheduling tags, if the same tag is defined by both the parent and the child, only the child's tag will take effect. For example, if a parent defines *high-mem* as a required tag, but a child defines *high-mem* as a preferred tag, then the tag will be treated as a preferred tag.

2. Combine

The combine operation matches up the current user, role and tool entities, and creates a combined entity that shares all their respective preferences. The combine operation follows specific rules:

Combining gpus, cores and mem In this case, the lower of the two values are used. For example, if a user entity specific 8 cores, and a tool requires 2 cores, then the lower value of 2 is used. An example of how this property can be used is to restrict training users from running jobs with lower memory than the defaults when running assembly jobs.

Combining tags

When combining tags, if a role expresses a preferences for tag *training* for example, and a tool expresses a requirement for tag *high-mem*, the combined entity would share both preferences. This can be used to route certain roles or users to specific destinations for example.

However, if the tags are mutually exclusive, then an IncompatibleTagsException is raised. For example, if a role expressed a preference for training, but the tool rejected tag *training*, then the job can no longer be scheduled. If the tags are compatible, then the tag with the stronger claim takes effect. For example, if a tool requires ‘high-mem` and a user prefers *high-mem*, then the combined entity will require *high-mem*. An example of using this property would be to restrict the availability of dangerous tools only to trusted users.

Combining envs and params

In this case, these requirements are simply merged, with duplicate envs and params merged in the following order: User > Role > Tool.

3. Evaluate

This operation evaluates any python expressions in the TPV config. It is divided into two steps, evaluate_early() and evaluate_late(). The former runs before the combine step and evaluates expressions for cores, mem and gpus. This ensures that at the time of combining entities, these values are concrete and can be compared. After the combine() step, the evaluate_late() function evaluates all remaining variables, ensuring that they have the latest possible values after combining requirements.

4. Match

The match operation is used to find matching destinations for the combined, evaluated entity. This step ensures that the destination has sufficient gpus, cores and mem to satisfy the entity’s request, assuming these are defined. If these are not defined, a match is assumed. In addition, all destinations that do not have tags required by the entity are rejected, and all destinations that have tags rejected by the entity are also rejected. Preference and acceptance is not considered at this stage, simply compatibility with available destinations based on the tag compatibility table documented later.

5. Rank

After the matching destinations are short listed, they are ranked using a pluggable rank function. The default rank function simply sorts the destinations by tags that have the most number of preferred tags, with a penalty if preferred tags are absent. However, this default rank function can be overridden per entity, allowing a custom rank function to be defined in python code, with arbitrary logic for picking the best match from the available candidate destinations.

1.2.3 Job Dispatch Process

When a typical job is dispatched, TPV follows the process below.

1. lookup - Looks up Tool, User and Role entity definitions that match the job
2. evaluate_early() - Evaluates gpu, cores, and mem expressions
3. combine() - Combines entity requirements to create a merged entity. Uses lower of gpu, cores and mem requirements
4. evaluate_late() - Evaluates remaining expressions as late as possible

5. match() - Matches the combined entity requirements with a suitable destination
6. rank() - The matching destinations are ranked
7. choose - The ranked destinations are evaluated, with the first non-failing match chosen (no rule failures)

1.2.4 Expressions

Most TPV properties can be expressed as python expressions. The rule of thumb is that all string expressions are evaluated as python f-strings, and all integers or boolean expressions are evaluated as python code blocks. For example, cpu, cores and mem are evaluated as python code blocks, as they evaluate to integer/float values. However, env and params are evaluated as f-strings, as they result in string values. This is to improve the readability and syntactic simplicity of TPV config files.

At the point of evaluating these functions, there is an evaluation context, which is a default set of variables that are available to that expression. The following default variables are available to all expressions:

Default evaluation context

Custom evaluation contexts

These are user defined context values that can be defined globally, or locally at the level of each entity. Any defined context value is available as a regular variable at the time the entity is evaluated.

Special evaluation contexts

In addition to the defaults above, additional context variables are available at different steps.

gpu, core and mem expressions - these are evaluated in order, and thus can be referred to in that same order. For example, gpu expressions cannot refer to core and mem, as they have not been evaluated yet. cpu expressions can be based on gpu values. mem expressions can refer to both cores and gpus.

env and param expressions - env expressions can be based on gpu, cores or mem. param expressions can additional refer to evaluated env expressions.

rank functions - these can refer to all prior expressions, and are additional passed in a *candidate_destinations* array, which is a list of matching TPV destinations.

1.2.5 Scheduling

TPV offers several mechanisms for controlling scheduling, all of which are optional. In its simplest form, no scheduling constraints would be defined at all, in which case the entity would schedule on the first available entity. Admins can use additional

Tag Type	Description
require	required tags must match up for scheduling to occur. For example, if a tool is marked as requiring the *high-mem* tag, only destinations that are tagged as requiring, preferring or accepting the *high-mem* tag would be considering for scheduling.
prefer	prefer tags are ranked higher that accept tags when scheduling decisions are made.
accept	accept tags can be used to indicate that a entity can match up or support another entity, even if not preferentially.
reject	reject tags cannot be present for scheduling to occur. For example, if a tool is marked as rejecting the *pulsar* tag, only destinations that do not have that tag are considered for scheduling. If two entities have the same reject tag, they still repel each other.

Tag compatibility table

Tag Type	Require	Prefer	Accept	Reject	Not Tagged
Require	✓	✓	✓	×	×
Prefer	✓	✓	✓	×	✓
Accept	✓	✓	✓	×	✓
Reject	×	×	×	×	✓
Not Tagged	×	✓	✓	✓	✓

Scheduling by tag match

Tags can be used to model anything from compatibility with a destination, to permissions to execute a tool. (e.g. a tool can be tagged as requiring the “restricted” tag, and users can be tagged as rejecting the “restricted” tag by default. Then, only users who are specifically marked as requiring, tolerating, or preferring the “restricted” tag can execute that tool. Of course, the destination must also be marked as not rejecting the “restricted” tag.

Scheduling by rules

Rules can be used to conditionally modify any entity requirement. Rules can be given an ID, which can subsequently be used by an inheriting entity to override the rule. If no ID is specified, a unique ID is generated, and the rule can no longer be overridden. Rules are typically evaluated through an *if* clause, which specifies the logical condition under which the rule matches. If the rule matches, cores, memory, scheduling tags etc. can be specified to override inherited values. The special clause *fail* can be used to immediately fail the job with an error message. The *execute* clause can be used to execute an arbitrary code block on rule match.

Scheduling by custom ranking functions

The default rank function sorts destinations by scoring how well the tags match the job’s requirements. As this may often be too simplistic, the rank function can be overridden by specifying a custom rank clause. The rank clause can contain an arbitrary code block, which can do the desired sorting, for example by determining destination load by querying the job manager, influx statistics etc. The final statement in the rank clause must be the list of sorted destinations.

1.3 Configuring Galaxy

1.3.1 Simple configuration

1. First install the TotalPerspectiveVortex into your Galaxy virtual environment.

```
cd <galaxy_home>
source .venv/bin/activate
pip install --upgrade total-perspective-vortex
```

2. Edit your *job_conf.yml* in the *<galaxy_home>/config* folder and add the highlighted sections to it.

   You can refer to a local file for the `tpv_config_files` setting, or alternatively, provider a link to a remote url.

```
runners:
  local:
    load: galaxy.jobs.runners.local:LocalJobRunner
    workers: 4
  drmaa:
    load: galaxy.jobs.runners.drmaa:DRMAAJobRunner
  k8s:
    load: galaxy.jobs.runners.kubernetes:KubernetesJobRunner

handling:
  assign:
    - db-skip-locked

execution:
  default: tpv_dispatcher
  environments:
    tpv_dispatcher:
      runner: dynamic
      type: python
      function: map_tool_to_destination
      rules_module: tpv.rules
      tpv_config_files:
        - https://github.com/galaxyproject/total-perspective-vortex/raw/main/tpv/tests/
↪fixtures/mapping-rules.yml
        - config/tpv_rules_local.yml
    local:
      runner: local
    k8s_environment:
      runner: k8s
      docker_enabled: true
```

3. Add your own custom rules to your local `tpv_config_file`, following instructions in the next section.

1.3.2 Combining multiple remote and local configs

TPV allows rules to be loaded from remote or local sources.

```yaml
tpv_dispatcher:
  runner: dynamic
  type: python
  function: map_tool_to_destination
  rules_module: tpv.rules
  tpv_config_files:
    - https://usegalaxy.org/shared_rules.yml
    - config/tpv_rules_australia.yml
```

The config files listed first are overridden by config files listed later. The normal rules of inheritance apply. This allows a central database of common rules to be maintained, with individual, site-specific overrides.

1.4 Shell Commands

1.4.1 lint

TPV config files can be checked for linting errors using the tpv lint command.

```
cd <galaxy_home>
source .venv/bin/activate
pip install --upgrade total-perspective-vortex
tpv lint <url_or_path_to_config_file>
```

If linting is successful, a lint successful message will be displayed with an exit code of zero. If the linting fails, a lint failed message with the relevant error will be displayed with an exit code of 1.

1.5 Indices and tables

- genindex
- modindex
- search