Gladier

Release 0.7.1

The Gladier Team

Dec 13, 2022

CONTENTS:

Gladier: The Globus Architecture for Data-Intensive Experimental Research.

Gladier is a development tool for rapidly building new scientific flows for experimental facilities. It seamlessly integrates FuncX's remote function execution capabilities with the Globus Flows Services, enabling complete data pipelines to be established from data capture to publication. Run your experiments, analyze inside a super computing environment, and publish your results to Globus Search.

For a full explanation of the Gladier architecture and API below, continue reading the docs here. If you would like to skip ahead and play with **Examples**, there are Binder notebooks listed below with all requirements pre-installed.

CHAPTER

ONE

GLADIER

Gladier is a development tool that sits on top of the Globus Automate Client and FuncX. This allows a scientist to write arbitrary python code for any purpose, deploy it to an HPC, and tie the workflow together into a complete Globus Flow for execution on demand.

The documentation below will cover the following:

- Installing Gladier
- Setting up your environment (FuncX and GCP)
- Running Gladier Clients

When you are familiar with running Gladier Clients, you should browse the reference documentation on Gladier Clients, Tools, and how to tweak the final Globus Flow with flow generation modifiers.

1.1 Installation

Gladier requires Python 3.6 and higher. For a modern version of python, see the official Python Installation Guide.

Gladier includes two main packages available through PyPi:

- Gladier – The core development tooling for building and running flows
- Gladier Tools – An optional set of plugable tools for common tasks

With pip installed, you can do the following:

```
pip install gladier gladier-tools
```

1.2 Setup

The typical Gladier Flow is composed of two major parts

- Transfer data into a location where execution can be performed (usually a super computer)
- Perform the execution

The first part is solved by Globus Transfer, and the second by FuncX Endpoints. In order for your computer to be accessible by Globus, it needs to be running a Globus Endpoint. See the Globus Connect Personal section below. For the second part, you will need to setup a FuncX Endpoint to tell FuncX where it should execute functions. See the FuncX Endpoint section below for details.

1.2.1 Globus Connect Personal

To run tools like `gladier_tools.globus.Transfer`, we need access to Globus endpoints. One can use Globus Tutorial Endpoint 1 and Globus Tutorial Endpoint 2. However, setting up your own endpoint gives you more control over the data you transfer, which brings us to Globus Connect Personal. Follow the Globus Connect Personal instructions to set up your own GCP endpoint, then navigate to Globus Web App endpoints to see details about the endpoint.

Your Globus endpoint should be accessible on the same machine you install your FuncX endpoint.

1.2.2 FuncX Endpoint

Another thing we need to run flows are an endpoint to run the funcx-functions on. There is a tutorial endpoint that can be used but it is generally preferred to set up a funcx-endpoint. Follow the FuncX Endpoint instructions to set up your own endpoint. Once an endpoint has been configured, you can list all the endpoints you have using `funcx-endpoint list`. To start an endpoint, use `funcx-endpoint start <endpoint-name>`.

1.3 Getting Started

The guide here shows a step by step guide to archiving a file and transferring it. Provided you have done the *Setup*, you can run this full example in the Tar and Transfer section in *Examples*.

A typical Gladier Client looks like the following:

```
@generate_flow_definition
class TarAndTransfer(GladierBaseClient):
    gladier_tools = [
        'gladier_tools.posix.Tar',
        'gladier_tools.globus.Transfer',
    ]
```

The new class has two main attributes:

- `gladier_tools` – Defines which tools you want to use for this client
- `@generate_flow_definition` – Splices together flows on each tool into one single runnable flow

Each Gladier tool sets its own FuncX Functions, required input, and separate Globus Flow. The Gladier Client gathers everything into one place. It handles registering functions, ensuring each tools input requirements are met, and deploying/updating the final combined Globus flow. After that, running the flow is as simple as these two lines:

```
tar_transfer_cli = TarAndTransfer()
flow = tar_transfer_cli.run_flow()
```

The first line instantiates the client with default values, which means it will automatically trigger login for Globus Auth and register Flows and FuncX functions as needed.

The second line, `flow = hello_cli.run_flow()`, is responsible for several different things:

- Gathering default input from each tool
- Validating input for each tool
- Triggering an initial login for access to FuncX and the Flows Service
- Registering funcx functions (And re-registering, if they have changed)
- Registering the `flow_definition` (And updating if it has changed)

- Triggering a second login to authorize the deployed flow
- Starting the flow

Once this checklist is complete, the flow is stared and will run through each flow-state in sequence. First, taring a directory then transferring the resulting archive. Nothing more is needed from Gladier, however there are extra tools for tracking a flows progress and output.

```
tar_transfer_cli.progress(flow['action_id'])
details = tar_transfer_cli.get_status(flow['action_id'])
pprint(details)
```

`hello_cli.progress()` will periodically query the status of the flow until it finishes. It's a nice way to watch progression as the flow executes. Once the flow has finished, `hello_cli.get_status()` will fetch output from the Globus Flow, so it can be displayed in a readable format.

1.3.1 Re-Running Flows

The most useful aspects of Gladier come into play when making changes to a flow or function. Both Flows and Functions are checksummed on each run to detect changes, and Gladier will automatically re-register any function that has changed, or update the flow if it has changed. This allows users to focus on writing functions without worrying if a change has been applied. Entire tools can also be added or removed from a flow simply by commenting them out.

Each flow or function update is logged in the `INFO` log in python. Additionally, if you wish to see the full flow definition at any time, it can be queried in an instantiated client like so:

```
tar_transfer_cli = TarAndTransfer()
pprint(tar_transfer_cli.flow_definition)
```

1.3.2 Next Steps

Running examples in the *Examples* section is highly recommended.

1.4 Clients

```
import os
from gladier import GladierBaseClient, generate_flow_definition
from my_tools import RunAnalysis


@generate_flow_definition
class ExampleAnalysisPipeline(GladierBaseClient):
    secret_config_filename = os.path.expanduser('~/.gladier-secrets.cfg')
    config_filename = 'gladier.cfg'
    app_name = 'gladier_client'
    client_id = 'e6c75d97-532a-4c88-b031-8584a319fa3e'

    gladier_tools = [
        'my_repo.custom_tools.PrepareData',
        RunAnalysis,
        'my_other_repo.stats.GenerateStatistics',
```

(continues on next page)

(continued from previous page)

```
        'my_other_repo.custom_tools.Publish',
    ]
```

1.4.1 Main Components

There are two main components of a Gladier Client:

- `gladier_tools` – A list of Gladier Tools derived from `gladier.GladierBaseTool` More on Gladier Tools below.
- `flow_definition` – A complete Globus Flow, supplied automatically with `@generate_flow_definition`

`gladier_tools` may be specified as a dotted string path (Ex: `my_repo.custom_tools.PrepareData`) or as a class that derives from `gladier.GladierBaseTool` (Ex: `RunAnalysis`). If `@generate_flow_definition` is used, the order the tools appear in the list determines the order they will appear in the flow. Each tool *must* have a complete and valid flow defined on it.

`flow_definition` may be a python dict which defines a full and complete flow, and can be an alternative to using `@generate_flow_definition`. Note that `@generate_flow_definition will overwrite the ``flow_definition` set on the client on instantiation.

1.4.2 Other Attritubes

- `globus_group` – adds the Globus Group to all Automate permission levels: visible_to, runnable_by, administered_by, manage_by, monitor_by. See the Flows Client Docs for more info.
- `secret_config_filename` – defines where login credentials are stored. Also stores function/flow ids and checksums
- `config_filename` – defines where GladierClient specific runtime variables are stored
- `app_name` – The name of the application, as it appears in Globus
- `client_id` – The UUID of the registered Globus Application, from https://developers.globus.org

1.4.3 See Also

- The complete *SDK Reference*
- *Flow Generation*

1.5 Tools

```python
from gladier import GladierBaseTool, generate_flow_definition


def makedirs(**data):
    """Make a directory on the filesystem"""
    import os
    os.makedirs(data['name'], mode=data['mode'], exist_ok=data['exist_ok'])
    return data['name']
```

(continues on next page)

(continued from previous page)

```
@generate_flow_definition
class MakeDirs(GladierBaseTool):
    """List files on the filesystem"""
    funcx_functions = [makedirs]
    required = ['name']
    flow_input = {
        'mode': 0o777,
        'exist_ok': False
    }
```

Gladier Tools are the glue that holds together Globus Flows and FuncX functions. Tools bundle everything the FuncX function needs to run, so the Glaider Client can register the function, check the requirements, and run it inside the Globus Flow.

Each tool is composed of zero or more `funcx_functions` and a `flow_definition`. Flows can be specified explicitly, or if they are a single FuncX function they can be generated automatically using the `@generate_flow_definition` decorator. When applied to tools, `@generate_flow_definition` will create a simple one-state flow for the function to run.

The main attributes of a Gladier Tool are here:

- funcx_functions (list of callables) – A list of functions this tool uses.
- flow_input (dict) – Default input that should be used in the flow. This is automatically overrided if the user supplies flow input to the Gladier Client
- required (list of strings) – A list of critical Funcx flow_input keys that **must** be present for the function to be run. Gladier will raise an exception if these are not present when the user attempts to run the flow.
- flow_definition (dict) – (Optional) A complete Globus Flow for running this tool. Provides a built-in flow new users can use to instantly run your tool.

1.6 Writing Custom Tools

Gladier was designed with the intention that users would write their own tools as needed. As your pipeline grows, you may include a mix of built-in Gladier Tools, your own custom tools, or even your own custom package of tools.

Starting with the simple tool below:

```
from gladier import GladierBaseTool, generate_flow_definition


def makedirs(**data):
    """Make a directory on the filesystem"""
    import os
    os.makedirs(data['name'], mode=data['mode'], exist_ok=data['exist_ok'])
    return data['name']


@generate_flow_definition
class MakeDirs(GladierBaseTool):
    """List files on the filesystem"""
```

(continues on next page)

(continued from previous page)

```
    funcx_functions = [makedirs]
    required = ['name']
    flow_input = {
        'mode': 0o777,
        'exist_ok': False
    }
```

In the same module, this can be added to a client by simply adding the `MakeDirs` class to `gladier_tools`:

```
# from gladier import GladierBaseClient
@generate_flow_definition
class ExampleClient(GladierBaseClient):
   gladier_tools = [MakeDirs]
```

Import strings are an optional feature of Gladier. They're used for built-in tools to keep a mental boundary from code executing remotely via FuncX Functions. You can create your own repository of tools and share it provided your tools are organized within a python package. See the Gladier Tools repository as an example.

1.7 Flow Generation

For simple flows, Gladier can handle the process of stitching together the flows of several tools into one big flow, or even generating a flow from scratch for a simple Gladier tool. Flow generation does not replace manually authored flows, but provides a path for automating simple flows that don't require a lot of branching.

Flow generation is available for both clients and tools with the `@generate_flow_definition` decorator. Although it's applied the same way, the decorator behaves a little differently for a client and a tool.

- Gladier Tool – Generates a flow for each function defined in `funcx_functions`
- Gladier Client – Combines flows for each Gladier Tool defined in `gladier_tools`

1.7.1 Flow Generation on Gladier Tools

For Gladier Tools that only need one flow step per FuncX Function, flow generation can be a good option. Gladier will automatically determine information about the functions on the tool, and incorporate them into the flow. The usage looks like this:

```
from gladier import GladierBaseTool, generate_flow_definition


def ls(data):
    """Do an 'ls' on the filesystem, given a ``dir``"""
    import os
    the_dir = data.get('dir', '~')
    return os.listdir(the_dir)


@generate_flow_definition
class FileSystemListCommand(GladierBaseTool):
    """List files on the filesystem"""
    funcx_functions = [ls]
```

The decorator `@generate_flow_definition` will automatically set the `flow_definition` attribute when a GladierClient includes it in `tools`. The flow it generates will be identical to the following:

```
{
  "Comment": "List files on the filesystem",
  "StartAt": "Ls",
  "States": {
    "Ls": {
      "ActionScope": "https://auth.globus.org/scopes/facd7ccc-c5f4-42aa-916b-
↪a0e270e2c2a9/automate2",
      "ActionUrl": "https://api.funcx.org/automate",
      "Comment": "Do an 'ls' on the filesystem, given a ``dir``",
      "Type": "Action",
      "ExceptionOnActionFailure": false,
      "Parameters": {
        "tasks": [
          {
            "endpoint.$": "$.input.funcx_endpoint_compute",
            "func.$": "$.input.ls_funcx_id",
            "payload.$": "$.input"
          }
        ]
      },
      "End": true,
      "ResultPath": "$.Ls",
      "WaitTime": 300
    }
  }
}
```

Flow Generation Modifiers

You'll notice that flow generation makes some assumptions you might want to change. For example, `ls` is not a compute heavy task, and might be better to run on the head node. Overriding some attributes of the flow can be done with `modifiers`.

```
@generate_flow_definition(modifiers={
    ls: {'endpoint': 'funcx_endpoint_non_compute'}
})
class FileSystemListCommand(GladierBaseTool):
    """List files on the filesystem"""
    funcx_functions = [ls]
```

1.7.2 Flow Generation on Gladier Clients

When Flow Generation is applied to Gladier Clients it does not generate the flow from scratch, but instead combines the flow on each Gladier Tool into one big flow.

Note that modifiers can also be used with Gladier Clients to customize some attributes.

```
@generate_flow_definition(modifiers={
   'generate_metadata': {'endpoint': 'funcx_endpoint_non_compute'},
   'publish_to_search': {'endpoint': 'funcx_endpoint_non_compute',
                         'payload': 'generate_metadata'}
})
class ProcessData(GladierBaseClient):
    gladier_tools = [
        'mytools.TransferData',
        'mytools.processing.ProcessData',
        'mytools.processing.GenerateGraphs',
        'mytools.processing.GenerateMetadata',
        'mytools.processing.PublishResults',
    ]
```

Modifiers

A complete reference is coming soon!

1.8 SDK Reference

class `gladier.client.`**`GladierBaseClient`**(*authorizers=None*, *auto_login=True*, *auto_registration=True*)

Bases: `object`

The Gladier Client ties together commonly used funcx functions and basic flows with auto-registration tools to make complex tasks easy to automate.

Default options are intended for CLI usage and maximum user convenience.

Parameters

- **`authorizers`** – Provide live globus_sdk authorizers with a dict keyed by scope.
- **`auto_login`** – Automatically trigger login() calls when needed. Should not be used with authorizers.
- **`auto_registration`** – Automatically register functions or flows if they are not previously registered or obsolete.

Raises

`gladier.exc.AuthException` – if authorizers given are insufficient

`login`(***login_kwargs*)

Login to the Gladier client. This will ensure the user has the correct tokens configured but it DOES NOT guarantee they are in the correct group to run a flow. Can be run both locally and on a server. See help(fair_research_login.NativeClient.login) for a full list of kwargs.

`logout`()

Log out and revoke this client's tokens. This object will no longer be usable until a new login is called.

`get_input()`

Get funcx function ids, funcx endpoints, and each tool's default input. Default input may not be enough to run the flow. For example if a tool does processing on a local filesystem, the file will always need to be provided by the user when calling run_flow().

Defaults rely on GladierBaseTool.flow_input defined separately for each tool.

Returns
input for a flow wrapped in an 'input' dict. For example: {'input': {'foo': 'bar'}}

`run_flow`(*flow_input=None*, *use_defaults=True*, ***flow_kwargs*)

Start a Globus Automate flow. Flows and Functions must be registered prior or self.auto_registration must be True.

If auto-registering a flow and self.auto_login is True, this may result in two logins. The first is for authorizing basic tooling, and the second is to autorize the newly registered automate flow.

Parameters

- **flow_input** – A dict of input to be passed to the automate flow. self.check_input() is called on each tool to ensure basic needs are met for each. Input MUST be wrapped inside an 'input' dict, for example {'input': {'foo': 'bar'}}.
- **use_defaults** – Use the result of self.get_input() to populate base input for the flow. All conflicting input provided by flow_input overrides values set in use_defaults.
- ****flow_kwargs** – Set several keyed arguments that include the label to be used in the automate app. If no label is passed the standard automate label is used. Also ensure label <= 64 chars long.

Raise
gladier.exc.ConfigException by self.check_input()

Raises
gladier.exc.FlowObsolete

Raises
gladier.exc.NoFlowRegistered

Raises
gladier.exc.RegistrationException

Raises
gladier.exc.FunctionObsolete

Raises
gladier.exc.AuthException

Raises
Any globus_sdk.exc.BaseException

`get_status`(*action_id*)

Get the current status of the automate flow. Attempts to do additional work on funcx functions to deserialize any exception output.

Parameters
action_id – The globus action UUID used for this flow. The Automate flow id is always the flow_id configured for this tool.

Raises
Globus Automate exceptions from self.flows_client.flow_action_status

Returns
a Globus Automate status object (with varying state structures)

`progress`(*action_id*, *callback=None*)

Continuously call self.get_status() until the flow completes. Each status response is used as a parameter to the provided callback, by default will use the builtin callback to print the current state to stdout.

Parameters

- `action_id` – The action id for a running flow. The flow is automatically pulled based on the current tool's flow_definition.
- `callback` – The function to call with the result from self.get_status. Must take a single parameter: mycallback(self.get_status())

CHAPTER TWO

GLADIER TOOLS

Gladier tools is an optional package containing commonly used operations. You can install Gladier Tools with the following:

```
pip install gladier-tools
```

Once the Gladier Tools packages is installed, individual tools can be added to your client like so:

```
@generate_flow_definition
class TarAndTransfer(GladierBaseClient):
    gladier_tools = [
        'gladier_tools.posix.Tar',
        'gladier_tools.globus.Transfer',
    ]
```

The following Gladier Tools are available:

```
# Posix
'gladier_tools.posix.Tar',
'gladier_tools.posix.UnTar',
'gladier_tools.posix.Encrypt',
'gladier_tools.posix.Decrypt',
'gladier_tools.posix.AsymmetricEncrypt',
'gladier_tools.posix.AsymmetricDecrypt',
'gladier_tools.posix.HttpsDownloadFile',

# Globus
'gladier_tools.globus.Transfer',

# Publish
'gladier_tools.publish.Publish',
```

See the sections below for detailed information about each tool.

2.1 Globus Tools

Globus Tools map to the list of Globus Maintained Action Providers

They may not even contain FuncX functions, but are rather a convenient way to add common flow operations to your Gladier Client without copying the JSON yourself.

2.1.1 Globus Transfer

class `gladier_tools.globus.transfer.`**`Transfer`**(*alias=None*, *alias_class=None*)

Bases: `GladierBaseTool`

Transfer is a simple single-state flow with no FuncX Functions, which talks directly to the Transfer Action Provider. It transfers only a single file or directory.

Parameters

- **`transfer_source_endpoint_id`** – Globus Source Endpoint UUID
- **`transfer_destination_endpoint_id`** – Globus Destination Endpoint UUID
- **`transfer_source_path`** – Globus Source Path
- **`transfer_destination_path`** – Globus Destination Path
- **`transfer_recursive`** – True if this is a directory, false otherwise.

2.2 Publish Tools

Publish Tools talk to both Globus Search and a Globus Transfer as one single operation. Typically, this is the final state in a processing flow as a way to submit data for viewing in a portal.

2.2.1 Publish

Publish takes files or directories and transfers them to another publication endpoint and ingests metadata about the files into Globus Search. Both the index in Globus Search and the Globus Endpoint in Globus Pilot must be setup first.

Setup only needs to be done once, then publish can be used freely afterwards. Both Globus Pilot and Globus Search CLI can be installed with the following:

```
pip install globus-search-cli globus-pilot
```

Globus Search CLI is responsible for setting up indices in Globus Search. Globus Pilot is a tool which generates metadata about files and directories and handles both the transfer to an endpoint and the ingest to search.

Create the index with the following:

```
globus-search index create my-index
```

Setup your publication endpoint with the index you created above with:

```
globus-pilot index setup <UUID from the step above>
```

After that, you should be ready to publish to your data. See documentation for both of the tools above here:

- Globus Search CLI

- Globus Pilot

`class gladier_tools.publish.Publish`(*alias=None*, *alias_class=None*)

Bases: `GladierBaseTool`

This function uses the globus-pilot tool to generate metadata compatible with portals on https://acdc.alcf.anl.gov/. Requires globus_pilot>=0.6.0.

FuncX Functions:

- publish_gather_metadata (funcx_endpoint_non_compute)

Publication happens in three steps:

- PublishGatherMetadata – A funcx function which uses globus-pilot to gather metadata on files or folders
- PublishTransfer – Transfers data to the Globus Endpoint selected in Globus Pilot
- PublishIngest – Ingest metadata gathered in fist step to Globus Search

Note: This tool needs internet access to fetch Pilot configuration records, which contain the destination endpoint and other project info. The default FuncX endpoint name is *funcx_endpoint_non_compute*. You can change this with the following modifier:

```
@generate_flow_definition(modifiers={
    'publish_gather_metadata': {'endpoint': 'funcx_endpoint_non_compute'},
})
```

More details on modifiers can be found at https://gladier.readthedocs.io/en/latest/gladier/flow_generation.html

NOTE: This tool nests input under the 'pilot' keyword. Submit your input as the following:

```
{
    'input': {
        'pilot': {
            'dataset': 'foo',
            'index': 'my-search-index-uuid',
            'project': 'my-pilot-project',
            'source_globus_endpoint': 'ddb59aef-6d04-11e5-ba46-22000b92c6ec',
        }
}
```

Parameters

- **`dataset`** – Path to file or directory. Used by Pilot to gather metadata, and set as the source for transfer to the publication endpoint configured in Pilot.
- **`destination`** – relative location under project directory to place dataset (Default `/`)
- **`source_globus_endpoint`** – The Globus Endpoint of the machine where you are executing
- **`index`** – The index to ingest this dataset in Globus Search
- **`project`** – The Pilot project to use for this dataset
- **`groups`** – A list of additional groups to make these records visible_to.
- **`funcx_endpoint_non_compute`** – A funcX endpoint uuid for gathering metadata. Requires internet access.

Requires: the 'globus-pilot' package to be installed.

2.3 Posix Tools

Below are Gladier Tools for common operations on posix filesystems.

2.3.1 Tar

`class gladier_tools.posix.tar.`**`Tar`**`(`*alias=None*, *alias_class=None*`)`

Bases:

The Tar tool makes it possible to create Tar archives from folders. FuncX Functions:

- tar (funcx_endpoint_compute)

Parameters

- **`tar_input`** – Input directory to archive.
- **`tar_output`** – (optional) output file to save the new archive. Defaults to the original input file with an extension (myfile.tgz) if not given.
- **`funcx_endpoint_compute`** – By default, uses the `compute` funcx endpoint.

Returns path

The name of the newly created archive.

2.3.2 Untar

`class gladier_tools.posix.untar.`**`UnTar`**`(`*alias=None*, *alias_class=None*`)`

Bases:

The UnTar tool makes it possible to extract data from Tar archives. FuncX Functions:

- untar (funcx_endpoint_compute)

Parameters

- **`untar_input`** – Input directory to archive.
- **`untar_output`** – (optional) output file to save the new archive. Defaults to the original # noqa input file with an extension '.tgz' removed.
- **`funcx_endpoint_compute`** – By default, uses the `compute` funcx endpoint. # noqa

Returns path

The name of the newly created archive.

2.3.3 HTTPS Download File

`class gladier_tools.posix.https_download_file.`**`HttpsDownloadFile`**`(`*alias=None*, *alias_class=None*`)`

Bases:

2.3.4 Encrypt

class gladier_tools.posix.encrypt.**Encrypt**(*alias=None*, *alias_class=None*)

Bases:

The Encrypt tool takes in a file and a password to perform 128-bit AES symmetric key encryption on the file. The original contents of the file are overwritten with the encrypted text. Adds an extension (.aes) to the name of the file. It has not been found to be compatible with 3rd party encryption/decryption tools.

FuncX Functions:

- encrypt (funcx_endpoint_compute)

Parameters

- **encrypt_input** – Path to the file which needs to be encrypted.
- **encrypt_output** – Custom path to outputfile. Default is the same file with the '.aes' suffix added
- **encrypt_key** – Symmetric key or "password" which can be used to decrypt the encrypted file.
- **funcx_endpoint_compute** – By default, uses the `compute` funcx endpoint.

Returns output_path
Location of the encrypted file.

2.3.5 Decrypt

class gladier_tools.posix.decrypt.**Decrypt**(*alias=None*, *alias_class=None*)

Bases:

Decrypt tool takes in an encrypted file and a password to perform decryption on the file. The decryption only works on files that have been encrypted by the Gladier Encrypt tool. It has not been found to be compatible with 3rd party encryption/decryption tools.

FuncX Functions:

- decrypt (funcx_endpoint_compute)

Parameters

- **decrypt_input** – Path to the file which needs to be decrypted.
- **decrypt_key** – Symmetric key or "password" which will be used to decrypt the encrypted file. Must be the same key that was used during encryption.
- **decrypt_output** – (optional) The full path to the decrypted file. If not provided, the decrypted file will have the same name as the input file, with the last 4 characters truncated(assuming it was a .aes file).
- **funcx_endpoint_compute** – By default, uses the `compute` funcx endpoint.

Returns output_path
Location of the decrypted file.

2.3.6 Asymmetric Encrypt

class `gladier_tools.posix.asymmetric_encrypt.`**`AsymmetricEncrypt`**(*alias=None*, *alias_class=None*)

Bases:

The Asymmetric Encrypt tool takes in a file and the path to the RSA public key to perform RSA encryption on the file. Adds an extension (.rsa) to the name of the file. It has not been found to be compatible with 3rd party encryption/decryption tools.

FuncX Functions:

- asymmetric_encrypt (funcx_endpoint_compute)

Parameters

- **`public_key_path`** – Path to the .pub file which contains the RSA public key. Defaults to ~/.ssh/id_rsa.pub
- **`asym_encrypt_file`** – File which needs to be encrypted.
- **`funcx_endpoint_compute`** – By default, uses the `compute` funcx endpoint.

Returns output_path
Location of the encrypted file.

2.3.7 Asymmetric Decrypt

class `gladier_tools.posix.asymmetric_decrypt.`**`AsymmetricDecrypt`**(*alias=None*, *alias_class=None*)

Bases:

The Asymmetric Decrypt tool takes in a file encrypted by the asymmetric encryption tool and the path to the RSA public key to perform decryption on the file. The output file can be passed in as a flow_input. If no output file passed, the last 4 characters (.rsa) of the input file are removed. It has not been found to be compatible with 3rd party encryption/decryption tools.

FuncX Functions:

- asymmetric_decrypt (funcx_endpoint_compute)

Parameters

- **`private_key_path`** – Path to the id_rsa file which contains the RSA private key. Defaults to ~/.ssh/id_rsa if not passed in.
- **`asym_decrypt_file`** – File which needs to be decrypted.
- **`funcx_endpoint_compute`** – By default, uses the `compute` funcx endpoint.
- **`asym_decrypt_password`** – (Optional) If the private file is password protected, pass it in through this argument.
- **`output_file`** – (Optional) Path to the output file which holds the decrypted contents.

Returns output_path
Location of the decrypted file.

CHAPTER

THREE

EXAMPLES

3.1 Pre Requisites

If you want to run the examples below on your own computer, you need to follow the *Setup* guide. The *Setup* guide covers setting up FuncX and GCP on the machine you are using.

Note: For a quick 5-minute start, you can run the Binder Examples, which are all run in a pre-built environment and require no setup.

3.2 Example Flows

3.2.1 Tar and Transfer

This example highlights the usage of the Tar and Transfer Gladier tools in a single flow. The tar step creates an archive of the input file, and the transfer step transfers the archived file between Globus Endpoints.

```
from gladier import GladierBaseClient, generate_flow_definition
from pprint import pprint


@generate_flow_definition
class TarAndTransfer(GladierBaseClient):
    gladier_tools = [
        'gladier_tools.posix.Tar',
        'gladier_tools.globus.Transfer',
    ]


if __name__ == '__main__':
    flow_input = {
        'input': {
            # The directory below should exist with files. It will be archived by the␣
↪Tar Tool.
            'tar_input': '~/myfiles',
            # Set this to your own funcx endpoint where you want to tar files
            # 'funcx_endpoint_compute': '',
            # Set this to the globus endpoint where your tarred archive has been created
```

(continues on next page)

(continued from previous page)

```
            # 'transfer_source_endpoint_id': '',
            # By default, this will transfer the tar file to Globus Tutorial Endpoint 1
            'transfer_destination_endpoint_id': 'ddb59aef-6d04-11e5-ba46-22000b92c6ec',
            # By default, the Tar Tool will append '.tgz' to the archive it creates
            'transfer_source_path': '~/myfiles.tgz',
            'transfer_destination_path': '~/my_archives/myfiles.tgz',
            'transfer_recursive': False,
        }
    }
    # Instantiate the client
    tar_and_transfer = TarAndTransfer()

    # Optionally, print the flow definition
    # pprint(tar_and_transfer.flow_definition)

    # Run the flow
    flow = tar_and_transfer.run_flow(flow_input=flow_input)

    # Track the progress
    action_id = flow['action_id']
    tar_and_transfer.progress(action_id)
    pprint(tar_and_transfer.get_status(action_id))
```

Steps

Start the workflow by overriding the Gladier Client with a list of the tools you want to use. In this case, we want to use the Tar, and Transfer tools.

```
@generate_flow_definition
class TarAndTransfer(GladierBaseClient):
   gladier_tools = [
      'gladier_tools.posix.Tar',
      'gladier_tools.globus.Transfer',
   ]
```

The `@generate_flow_definition` decorator takes the flow definitions of the individual tools and constructs a flow definition for the new flow, so we do not have to define a custom flow definition. To view the constructed flow definition, use `pprint(tat.flow_definition)`. Refer to the Flow Generation doc for more details on the same.

The next step is to define the input for the flow. It might be helpful to refer to the docs for each of the tools to find out what needs to be passed in as input. For example, here are the documentations for the Tar and Transfer tools. Feel free to use the below blueprint:

```
flow_input = {
   'input': {
      # Set this to the file/folder that has to be tarred
      'tar_input': '',
      # Set this to your own funcx endpoint where you want to tar files
      'funcx_endpoint_compute': '',
      # Set this to the globus endpoint where your tarred archive has been created
      'transfer_source_endpoint_id': '',
      # By default, this will transfer the tar file to Globus Tutorial Endpoint 1
```

(continues on next page)

(continued from previous page)

```
        'transfer_destination_endpoint_id': 'ddb59aef-6d04-11e5-ba46-22000b92c6ec',
        # Provide the paths to the tarred file as input for the transfer step
        'transfer_source_path': '',
        'transfer_destination_path': '',
        'transfer_recursive': False,
    }
}
```

All that is left is to create an instance of the `GladierBaseClient` class and run the flow. Use the below code to view the progress of the flow:

```
tat = TarAndTransfer()
pprint(tat.flow_definition)
flow = tat.run_flow(flow_input=flow_input)
action_id = flow['action_id']
tat.progress(action_id)
pprint(tat.get_status(action_id))
```

3.2.2 Encrypt and Transfer

This example highlights the usage of built in tools (Tar, Encrypt and Transfer) along with a custom defined tool (MakeFiles) in the same flow.

```
from gladier import GladierBaseClient, generate_flow_definition, GladierBaseTool
from pprint import pprint


def make_files(**data):
    from pathlib import Path

    input_path = Path(data['make_input']).expanduser()
    input_path.mkdir(exist_ok=True, parents=True)

    for number in range(3):
        with open(input_path / f'file{number}.txt', 'w') as f:
            f.write(f'This is file no. {number}')

    return str(input_path)


@generate_flow_definition
class MakeFiles(GladierBaseTool):
    funcx_functions = [make_files]
    required_input = [
        'make_input',
        'funcx_endpoint_compute'
    ]


@generate_flow_definition
class EncryptAndTransfer(GladierBaseClient):
```

(continues on next page)

(continued from previous page)

```
    gladier_tools = [
        MakeFiles,
        'gladier_tools.posix.Tar',
        'gladier_tools.posix.Encrypt',
        'gladier_tools.globus.Transfer',
    ]


if __name__ == '__main__':
    flow_input = {
        'input': {
            # Set this to the folder in which you want to run the makeFiles function in
            'make_input': '/tmp/myfiles',
            # Set this to the same folder as above
            'tar_input': '/tmp/myfiles',
            # Set this to the resultant archive of the above folder
            'encrypt_input': '/tmp/myfiles.tgz',
            # Set this to the symmetric key you want to use to encrypt/decrypt the file
            'encrypt_key': 'my_secret',
            # Set this to your own funcx endpoint where you want to encrypt files
            # 'funcx_endpoint_compute': '',
            # Set this to the globus endpoint where your encrypted archive has been␣
↪created
            # 'transfer_source_endpoint_id': '',
            # By default, this will transfer the encrypt file to Globus Tutorial␣
↪Endpoint 1
            'transfer_destination_endpoint_id': 'ddb59aef-6d04-11e5-ba46-22000b92c6ec',
            'transfer_source_path': '/tmp/myfiles.tgz.aes',
            'transfer_destination_path': 'my_encrypted_files/myfiles.tgz.aes',
            'transfer_recursive': False,
        }
    }

    # Create the Client
    encrypt_and_transfer = EncryptAndTransfer()

    # Optionally print the flow definition
    # pprint(encrypt_and_transfer.flow_definition)

    # Run the flow
    flow = encrypt_and_transfer.run_flow(flow_input=flow_input)

    # Track progress
    action_id = flow['action_id']
    encrypt_and_transfer.progress(action_id)
    pprint(encrypt_and_transfer.get_status(action_id))
```

Steps

We start by writing our own custom tool. For the example, MakeFiles is a trivial tool that creates three files in a given folder. The funcx function that achieves this functionality is `make_files`. We then define the tool using `GladierBaseTool`.

```
@generate_flow_definition
class MakeFiles(GladierBaseTool):
    funcx_functions = [make_files]
    required_input = [
        'make_input',
        'funcx_endpoint_compute'
    ]
```

Defining a workflow is similar to the previous case, where we override the `GladierBaseClient` class. In this case, we want to use the MakeFiles, Tar, Encrypt, and Transfer tools.

```
@generate_flow_definition
class CustomTransfer(GladierBaseClient):
    gladier_tools = [
        MakeFiles,
        'gladier_tools.posix.Tar',
        'gladier_tools.posix.Encrypt',
        'gladier_tools.globus.Transfer',
    ]
```

The `@generate_flow_definition` decorator takes the flow definitions of the individual tools and constructs a flow definition for the new flow, so we do not have to define a custom flow definition. To view the constructed flow definition, use `pprint(ct.flow_definition)`. Refer to the Flow Generation doc for more details on the same.

The next step is to define the input for the flow. It might be helpful to refer to the docs for each of the tools to find out what needs to be passed in as input. For example, here are the documentations for the Tar, Encrypt and Transfer tools. Feel free to use the below blueprint:

```
flow_input = {
    'input': {
        # Set this to the folder in which you want to run the makeFiles function in
        'make_input': '',
        # Set this to the same folder as above
        'tar_input': '',
        # Set this to the resultant archive of the above folder
        'encrypt_input': '',
        # Set this to the symmetric key you want to use to encrypt/decrypt the file
        'encrypt_key': '',
        # Set this to your own funcx endpoint where you want to encrypt files
        'funcx_endpoint_compute': '',
        # Set this to the globus endpoint where your encrypted archive has been created
        'transfer_source_endpoint_id': '',
        # By default, this will transfer the encrypt file to Globus Tutorial Endpoint 1
        'transfer_destination_endpoint_id': 'ddb59aef-6d04-11e5-ba46-22000b92c6ec',
        'transfer_source_path': '',
        'transfer_destination_path': '',
        'transfer_recursive': False,
    }
}
```

All that is left is to create an instance of the `GladierBaseClient` class and run the flow. Use the below code to view the progress of the flow:

```
ct = CustomTransfer()
pprint(ct.flow_definition)
flow = ct.run_flow(flow_input=flow_input)
action_id = flow['action_id']
ct.progress(action_id)
pprint(ct.get_status(action_id))
```

CHAPTER

FOUR

UPGRADE MIGRATIONS

4.1 Migrating to v0.5.0

The only major change in v0.5.0 was the removal of the HelloWorld Tools from the main Gladier package. The following are no longer present:

- `gladier.tools.hello_world.HelloWorld`
- `gladier.tools.manifest.ManifestTransfer`
- `gladier.tools.manifest.ManifestToFuncXTasks`

There currently aren't plans to rewrite them in the Gladier Tools package, but open an issue if you would like us to consider changing that!

4.2 Migrating to v0.4.0

Gladier v0.3.x depended on FuncX v0.0.5 and FuncX Endpoint v0.0.3. Gladier v0.4.x now uses Funcx v0.2.3-v0.3.0+ (funcx-endpoint v0.2.3-v0.3.0+). There are a number of breaking changes between these two versions of FuncX, including funcx endpoints, flow definitions, and backend services.

4.2.1 FuncX Endpoints

All FuncX endpoints will need to be recreated with the never version of FuncX. Gladier typically names these endpoints as the following:

- `funcx_endpoint_non_compute`
- `funcx_endpoint_compute`

Since these use different backend services, using endpoints that don't match the FuncX version will result in errors. Using 0.0.3 endponits on 0.2.3+ will result in permission denied, using 0.2.3+ on 0.0.3 will result in Server 500s.

4.2.2 Argument Passing and Function Definitions

Previously, all arguments in a Flow were passed to FuncX functions as a dict. It looked like the following:

```
'Parameters': {'tasks': [{'endpoint.$': '$.input.funcx_endpoint_non_compute',
                          'function': '8227609b-4869-4c6f-9a1b-87dc49fcc687',
                          'payload.$': '$.input'}]},

def my_function(data):
    ...
```

In the above, `data` would get the entire dict from $.input, which was typically whatever input was passed to start the flow. In the new version of FuncX, this has changed. All arguments are either positional or keyword arguments and should be named. This is difficult in automate, since naming arguments requires specifying them explicitly in the flow definition. An easy migration path is the following:

```
'Parameters': {'tasks': [{'endpoint.$': '$.input.funcx_endpoint_non_compute',
                          'function': '8227609b-4869-4c6f-9a1b-87dc49fcc687',
                          'payload.$': '$.input'}]},

def my_function(**data):
    ...
```

Changing data to a keyword argument will allow re-creating the same behavior as before.

4.2.3 FuncX Functions

Like FuncX Endpoints, FuncX Functions also need to be changed between versions. This is an automatic process in most cases if you are running the latest version of Gladier and saw a big giant warning when upgrading. Gladier will automatically delete funcx functions that don't match the newly supported version of FuncX Gladier uses.

However, it's necessary to do a manual upgrade to remove these functions in some cases. To upgrade manually, edit the file `~/.gladier-secrets.cfg`, and remove all config items that end in `funcx_id` and `funcx_id_checksum`:

```
hello_world_funcx_id = 3bccfcdb-bc0e-4549-9297-8e08c6f50bd5
hello_world_funcx_id_checksum =␣
↪c590423de52051e7b7bb044dc173673d2c9ad965f7f71bee665494815b3a2046
```

4.2.4 Flow Definitions

Some items in Automate flow definitions also changed. See below for a list of the attributes.

FuncX Version 0.0.5 flow definitions:

- `ActionUrl` – 'https://api.funcx.org/automate'
- `ActionScope` – 'https://auth.globus.org/scopes/facd7ccc-c5f4-42aa-916b-a0e270e2c2a9/automate2'

FuncX Version 0.2.3+ flow definitions:

- `ActionUrl` – 'https://automate.funcx.org'
- `ActionScope` – 'https://auth.globus.org/scopes/b3db7e59-a6f1-4947-95c2-59d6b7a70f8c/action_all'

Additionally for FuncX Payloads, Function UUIDs are passed with a different name.

'func.$': '$.input.'

Needs to be changed to:

'function.$': '$.input.'

4.2.5 FuncX Flow Result Format

The format of the return value from FuncX functions has changed format. This only affects Flow states that depend on the output of a FuncX function/flow state.

Previous flow states were not returned in a list, and were referenced with the following:

```
'InputPath': '$.MyFuncXFunctionOutput.details.result',
```

FuncX now returns these in a list, and they need to be index. The above needs to be changed to the following:

```
'InputPath': '$.MyFuncXFunctionOutput.details.result[0]',
```

CHAPTER

FIVE

GLOSSARY

Gladier builds on a science services framework, in the form of Globus Auth, Transfer, Search, Groups, and Flows, plus funcX function-as-a-service. These services provide a reliable, secure, and high-performance substrate to access and manage data and computing resources. Here we highlight several of these services and describe how they are used to create Gladier deployments.

5.1 Globus

Globus provides a collection of data services built for science including: Globus Auth, Transfer, Search, Groups, and Flows, and funcX to enable distributed function-as-a-service execution. Globus Services are highly reliable, professionally operated cloud-hosted services that support the work of over 150,000 researchers worldwide as foundational capabilities for scientific applications and workflows; using them greatly reduces the burden on local systems, administrators, and programmers.

5.2 Globus Flows

Flows addresses the problem of securely and reliably automating sequences of data management tasks that may span locations, storage systems, administrative domains, and timescales, and integrate both mechanical and human inputs. Client libraries deployed on Globus endpoints and other sources enable the detection of events and invocation of a flow. The Flows service manages execution of user-supplied automation flows either manually or as a result of data events, and the invocation of actions from those automation flows, including actions provided by Globus endpoints and services. The service is extensible via the definition of new events and actions to meet the needs of specific communities.

5.3 FuncX

funcX is a function-as-a-service platform that implements a federated compute substrate, enabling computation to be registered as Python functions and invocations to be dispatched to remote computers for execution. The service provides a single point-of-contact, supporting function registration, sharing, and discovery as well as reliable and secure execution on connected endpoints. The funcX endpoint software, built on Parsl, allows functions to be executed in containers and for resources to be dynamically provisioned on cloud and cluster systems. These funcX endpoints provide serverless capabilities whereby researchers fire-and-forget tasks that are dynamically allocated across the supercomputer using an opportunistic backfill queue to utilize spare capacity.

5.4 Globus Queues and Triggers

Queues provides a reliable, cloud-based mechanism to manage and store events. The Queues service allows users to provision a dedicated queue for their instrument. Clients can then raise events to the queue using HTTP POST requests where they will be maintained until a subscriber consumes them. This enables experimental facilities and instruments to raise events as data are created without requiring heavy-weight installations on the edge device.

The **Triggers** service provides a cloud-based consumer of Queues events. Users can configure a Trigger to monitor a queue and initiate Flows as events are received. To create a trigger one defines:

- An event queue for the trigger to monitor
- A condition for when the trigger will fire
- An action to perform when the condition is met (e.g., a flow uuid)
- A template to create an input JSON document for the action. This often includes default values.

The combination of the **Queues** and **Triggers** services simplifies creating new Gladier deployments.

5.5 Globus Transfer

Transfer implements a location-agnostic data substrate that enables data to be accessed, shared, and moved among disparate storage systems, including at instruments, supercomputers, and on data services. Globus Transfer allows users and applications to modify data access permissions on remote storage systems and to move data reliably and securely between systems via a single API.

5.6 Globus Auth

Auth Auth allows for users to delegate permissions for clients to access services in the Gladier architecture, and for services to access other services on their behalf as well. For example, it allows Globus Flows to manage Globus Transfers and to execute analyses via funcX on systems accessible only to the user.

CHAPTER

SIX

CHANGELOG

All notable changes to this project will be documented in this file. See standard-version for commit guidelines.

6.1 0.7.1 (2022-08-25)

6.2 Bug Fixes

- Error on first time flow deployment if using group perms (b444f2b)

6.2.1 0.7.0 (2022-08-22)

6.3 BREAKING CHANGES

- Older funcx versions before v1.0 are no longer supported.
  - No code changes are required to migrate to Gladier v0.7.0 or FuncX v1.0

6.4 Features

- Add support for globus-automate-client 0.16 (dc9e82c)
- Upgrade to funcx v1 (c947077)

6.5 Bug Fixes

- Aliases not working with tools that use `@generate_flow_definition` (da30756)

6.6 0.6.3 (2022-07-20)

6.7 Bug Fixes

- Fix error when docstring is too long (a19a4bf)

6.8 0.6.2 (2022-05-06)

6.9 Bug Fixes

- Possible flows client 401 due to client caching (3b5a307)

6.10 0.6.1 (2022-05-05)

6.11 Features

- Add support for globus-automate-client 0.15.x (9ecd2e1)

6.11.1 0.6.0 (2022-05-05)

6.12 BREAKING CHANGES

- The following versions of FuncX and the Globus Automate Client will no longer be supported:
- Globus Automate Client: Requires 0.13.0 and above
- FuncX: Requires 0.3.6 and above

Older versions of these packages are only compatible with Globus SDK v2, and require updating any code that relies on the older Globus SDK version. See the SDK upgrade guide here:

6.13 Features

- Added 'aliased' tool chaining feature (48cbaaf)

6.14 Bug Fixes

- Add packaging dependency (95f6ec1)
- Added check when adding tool without flow states (c612383)
- Redeploy flow on 404s (6c0d3a7)
- tools/flows client not properly being cached in gladier clients (a6e5cec)

- Drop support for old versions of globus-automate-client/funcx (5d17d96), closes /globus-sdk-python.readthedocs.io/en/stable/upgrading.html#from-1-x-or-2-x-to-3-0

6.14.1 0.6.0b2 (2022-02-17)

6.15 Bug Fixes

- Add packaging dependency (95f6ec1)

6.15.1 0.6.0b1 (2022-02-17)

6.16 BREAKING CHANGES

- The following versions of FuncX and the Globus Automate Client will no longer be supported:
- Globus Automate Client: Requires 0.13.0 and above
- FuncX: Requires 0.3.6 and above

Older versions of these packages are only compatible with Globus SDK v2, and require updating any code that relies on the older Globus SDK version. See the SDK upgrade guide here:

6.17 Features

- Added 'aliased' tool chaining feature (48cbaaf)

6.18 Bug Fixes

- Added check when adding tool without flow states (c612383)
- Redeploy flow on 404s (6c0d3a7)
- tools/flows client not properly being cached in gladier clients (a6e5cec)
- Drop support for old versions of globus-automate-client/funcx (5d17d96), closes /globus-sdk-python.readthedocs.io/en/stable/upgrading.html#from-1-x-or-2-x-to-3-0

6.19 0.5.4 (2021-11-15)

6.20 Bug Fixes

- Only apply migrations when needed (670daea)

6.21 0.5.3 (2021-09-14)

6.22 Bug Fixes

- Limits run label length to 64 chars (53cd20f), closes #146

6.23 0.5.2 (2021-08-23)

6.24 Features

- Expanded flow modifiers to accept all top level state fields (3b3135f)

6.25 Bug Fixes

- Flow Modifier errors not propagating Client or Tool names (1af9724)
- Remove funcx-endpoint version check (8fe88a3)

6.26 0.5.1 (2021-08-19)

6.27 Bug Fixes

- Deploying new flows with the latest version of the flows service (afa5adf)

6.27.1 0.5.0 (2021-08-05)

6.28 BREAKING CHANGES

- Removal of older introductory testing tools

6.29 Bug Fixes

- logout not properly clearing authorizers cache (05b0d6c)
- Pass an ‘empty’ schema by default to fulfill automate requirement (bf9eb80)
- pin automate version to avoid future incompatible releases (b736fa2)
- Raise better exception when no flow definition set on tool (eb8ac03)
- Remove old “Hello World” tools. We have better ones now. (3a889f9)

6.30 0.4.1 (2021-07-20)

6.31 Features

- Added get_run_url for fetching the link to a running flow in the Globus webapp
- Arguments to run_flow support pass through args to the flows service

6.31.1 0.4.0 (2021-07-19)

6.32 BREAKING CHANGES

- – This will break all current funcx functions without modification. Everyone will need to upgrade to the new funcX endpoint package wherever they are executing functions. See the full migration doc in Migrating to V0.4.0

6.33 Features

- Upgrade to FuncX 0.2.3 (from 0.0.5) (83507f7)

6.34 0.3.5 (2021-07-14)

6.35 Features

- Added config migration system to Gladier (cdc7875)

6.36 0.3.4 (2021-07-09)

6.37 Bug Fixes

- gladier improperly falling back onto FuncX authorizers (3976a3a)

6.38 0.3.3 (2021-06-18)

6.39 Bug Fixes

- Tools with more than two states would raise error with flow gen (dd6586e)
- when user adds new AP to flow, Gladier now handles re-auth (e24a372)

6.40 0.3.2 (2021-06-17)

6.41 Bug Fixes

- add funcx-endpoint==0.0.3 to Gladier requirements (8ac47b8)

6.42 0.3.1 (2021-06-04)

6.43 Bug Fixes

- Fixed bug when instantiating two Gladier Clients (3aca2fe)

6.44 0.3.0 (2021-05-28)

6.45 Features

- Support flow modifiers, payload dict modifiers (a05b77a)
- FuncX ids and Flow ids are now saved in ~/.gladier-secrets.cfg instead of ./gladier.cfg
- Added support for setting Groups
- Added support for setting subscription ids

6.46 Bug Fixes

- Added changes lost from previous merges to fix client (c707d32)

6.47 0.2.0 - May 17, 2021

- Changed name `gladier.defaults.GladierDefaults` to `gladier.base.GladierBaseTool`
- Changed name `gladier.client.GladierClient` to `gladier.client.GladierBaseClient`
- Added a lot more documentation to the read-the-docs page!

6.48 0.0.1 - Apr 5, 2021

- Initial Release!

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX