intercom_test Documentation

Richard T. Weeks

Mar 02, 2021

Contents:

CHAPTER 1

intercom_test

1.1 intercom_test package

1.1.1 Subpackages

intercom_test.augmentation package

Submodules

intercom_test.augmentation.compact_file module

class intercom_test.augmentation.compact_file.**CaseIndexer**

Bases: object

Collector of case keys and their "jump indexes" in a compact file

Objects of this class consume YAML events (as from `yaml.parse()`) and collect the test case keys and their corresponding starting offsets within the file, assuming the file represents the top level mapping in block format.

class State

Bases: enum.Enum

An enumeration.

case_data = 3

case_key = 2

header = 1

tail = 4

read(*event*)

class intercom_test.augmentation.compact_file.**DataValueReader**(*stream*, *start_byte*, *case_key*, *, *safe_loading=None*)

Bases: object

Reads the augmentation data for a single case in a compact file

The code constructing this object should know the starting byte offset into the stream and the test case key located at that offset. This allows the reader to skip directly to that case and process only that case.

augment(*d*)

augmentation_data_events()

safe_loading = True

class intercom_test.augmentation.compact_file.**TestCaseAugmenter**(*file_path*, *offset*, *case_key*, *, *safe_loading=None*)

Bases: object

Callable to augment a test case from a compact entry

case_data_events()

safe_loading = True

class intercom_test.augmentation.compact_file.**Updater**(*updates*, *excluded_keys=()*)

Bases: object

YAML event-stream editor for compact augumentation data files

Objects of this class support applying a set of updates/additions to the stream of YAML events from a compact augmentation data file. Each event is fed to *filter()*, which returns an iterable of events to include in the output.

Updates are a dict (or similar by duck-type) keyed by *case keys*; the corresponding or return values are either a dict of augmentation values to associate with the test case or an iterable of yaml.Event objects representing a YAML node to be used as the augmenting value. The event list approach allows more fidelity in preserving the representation from the update file.

class State

Bases: enum.Enum

An enumeration.

case_data = 3

header = 1

tail = 4

top_mapping = 2

filter(*event*)

Converts an event into an iterable of events (possibly empty)

intercom_test.augmentation.compact_file.**augment_dict_from**(*d*, *file_ref*, *case_key*, *, *safe_loading=True*)

intercom_test.augmentation.compact_file.**case_keys**(*data_file*)

intercom_test.augmentation.update_file module

class intercom_test.augmentation.update_file.**CaseReader**(*stream*, *start_byte*, *key_fields*, *, *safe_loading=None*)

Bases: object

Given a file and a starting point, reads the case data

Can be used to *augment()* a dict of test case values or to read *augmentation_data_events()* for updating a compact file.

class State

Bases: enum.Enum

An enumeration.

key = 1

value = 2

TRAILING_WS = re.compile('\\s+\n')

augment(*d*)

augmentation_data_events()

safe_loading = True

class intercom_test.augmentation.update_file.**Indexer**(*key_fields*, *, *safe_loading=None*)

Bases: object

Builds an index of the augmentation data in a working/update file

While an update file may be in any valid YAML format, certain formats are more efficient for the system to manage. Specifically, it is best if:

- The top-level sequence is represented in block (not flow) format
- Each case is "atomic" – that is, contains no aliases to nodes outside it's own entry in the sequence

In cases where these conditions are not met, the indexer notes the cases in the output, but does not provide a starting offset into the file. The result: augmenting the case (or updating the compact augmentation file) requires reloading the entire YAML file, not just the single case.

class State

Bases: enum.Enum

An enumeration.

case_data_key_collection = 4

case_data_value = 5

case_data_value_collection = 6

case_mapping = 3

header = 1

tail = 7

top_sequence = 2

read(*event*)

safe_loading = True

class intercom_test.augmentation.update_file.**TestCaseAugmenter**(*file_path*, *offset*, *key_fields*, ***, *case_index=None*, *safe_loading=None*)

Bases: object

Callable to augment a test case from an update file entry

case_data_events()

case_reference

deposit_file_path

safe_loading = True

intercom_test.augmentation.update_file.**index**(*paths*, *key_fields*, ***, *safe_loading=True*)

Module contents

intercom_test.json_asn1 package

Submodules

intercom_test.json_asn1.convert module

intercom_test.json_asn1.convert.**asn1**(*value*)

intercom_test.json_asn1.convert.**asn1_der**(*value*)

intercom_test.json_asn1.convert.**kvp**(*k*, *v*)

intercom_test.json_asn1.types module

class intercom_test.json_asn1.types.**JSONObject**(**args*, ***kwargs*)

Bases: pyasn1.type.univ.SetOf

componentType = <KeyValuePair schema object, tagSet=<TagSet object, tags 64:32:1>, sub

class intercom_test.json_asn1.types.**JSONValue**(***kwargs*)

Bases: pyasn1.type.univ.Choice

componentType = <NamedTypes object, types <NamedType object, type nullval=<Null schema

class intercom_test.json_asn1.types.**KeyValuePair**(***kwargs*)

Bases: pyasn1.type.univ.Sequence

componentType = <NamedTypes object, types <NamedType object, type key=<UTF8String sche

tagSet = <TagSet object, tags 64:32:1>

Module contents

1.1.2 Submodules

intercom_test.aws_http module

Module for adapting HTTP requests to AWS API Gateway events

class intercom_test.aws_http.**CasePreparer**(*case: collections.abc.Mapping*)
Bases: object

Common base class for building an AWS API Gateway event for a Lambda Function

Parameters **case** – test case data

The following keys in *case* are consulted when generating the Lambda Function input value (*lambda_input()*):

'method' (str, **required**) The HTTP method

'url' (str, **required**) The path part of the URL

'stageVariables' (dict) Mapping of stage variables to their values

'request headers' (dict or list of 2-item lists) HTTP headers for request

'request body' A str, bytes, or JSONic data type giving the body of the request to test; JSONic data is rendered to JSON for submission and implies a Content-Type header of 'application/json'

Subclasses may also consult additional keys in *case*; see the documentation of the subclass.

lambda_input() → dict
Get the Lambda Function input for the test case

method
HTTP method of the test case

url
URL of the test case

class intercom_test.aws_http.**FunctionalHandlerMapper**(*mapper: Callable[[str, str], Callable[[dict, dict], dict]]*)
Bases: *intercom_test.aws_http.HandlerMapper*

Adapter class to convert a mapper function to a *HandlerMapper*

map(*method: str*, *path: str*) → Callable[[dict, dict], dict]
Given an HTTP method and request path, return a handler function

class intercom_test.aws_http.**HandlerMapper**
Bases: abc.ABC

Abstract base class for classes that can map HTTP requests to handlers

map(*method: str*, *path: str*) → Callable[[dict, dict], dict]
Given an HTTP method and request path, return a handler function

class intercom_test.aws_http.**HttpCasePreparer**(*case: collections.abc.Mapping*)
Bases: *intercom_test.aws_http.CasePreparer*

Prepare Lambda Function input event for HTTP API

Parameters **case** – test case data

In addition to the keys listed in base class *CasePreparer*, this class also consults the following optional keys of *case* when building the Lambda Function input:

'client certificate' (dict) The field of the client certificate provided for the test, to populate $.requestContext.authentication.clientCert

'request authorization' (dict) Used to populate $.requestContext.authorizer

exception intercom_test.aws_http.**InvalidPathTemplate**
Bases: Exception

Raised when an invalid routing path template

ATTRIBUTES = 'path_template error_index'

error_index
Index 1 argument to the constructor

path_template
Index 0 argument to the constructor

class intercom_test.aws_http.**LambdaHandlerProxy**(*handler: Callable[[dict, dict], dict], *, resource: Optional[str] = None*)
Bases: object

Wrapper for a Lambda handler allowing decoration with additional attributes

A single handler function may be bound to multiple integrations, and the information relevant to that binding may be useful or needed for constructing the event to send to the handler.

exception intercom_test.aws_http.**NoRoute**
Bases: Exception

Raised when no route matched the given method and path

ATTRIBUTES = 'method path'

method
Index 0 argument to the constructor

path
Index 1 argument to the constructor

class intercom_test.aws_http.**OpenAPIPathMatcher**(*route_method: str*, *route_path: str*)
Bases: object

A callable class to match and extract parameters from a URL path

Parameters

- **route_method** – HTTP method or `'*'` for a wildcard
- **route_path** – path part of a URL to match, which may include OpenAPI template parameters

Instances accept a call with an HTTP method and a URL path-part and return either `None` for no match, or a `dict` mapping path parameter names to their extracted values. A returned mapping may be empty, so be sure to use the `is not None` test instead of implicit conversion to `bool`.

NOTE: With the current implementation, template parameters are only allowed to match a full segment of the path (between slashes or from a slash to the end of the path).

class Param
Bases: str

String giving the name of a path parameter

isvartail

class intercom_test.aws_http.**RESPONSE_BODY_EXPECTATIONS**
Bases: object

Key class for customized response body expectations

For cases where a strict equality test of the response body generated by the handler to the response body specified in the interface data is not desirable, set the entry in the case `dict` keyed by this class itself (not an instance of it) to a callable that takes the parsed JSONic data as its only argument and returns a `bool` indicating whether the response expectations are met.

This can, for example, be used to integrate a JSONic data comparison library with more flexible matching.

A common place to set the entry in the case keyed with this class is in the callable passed in the *case_env=* keyword to *ServerlessHandlerMapper.case_tester()*.

If the default behavior were coded in client code, it would look like:

```
def around_case(case: dict):
    def response_expectations(response):
        return response == case['response body']
    case[RESPONSE_BODY_EXPECTATIONS] = response_expectations

    yield
```

class intercom_test.aws_http.**RestCasePreparer**(*case: collections.abc.Mapping*)

Bases: *intercom_test.aws_http.CasePreparer*

Prepare Lambda Function input event for REST API

Parameters **case** – test case data

In addition to the keys listed in base class *CasePreparer*, this class also consults the following optional keys of *case* when building the Lambda Function input:

'identity' (`dict`) Client identity information used to populate `$.requestContext.identity`

class intercom_test.aws_http.**ServerlessHandlerMapper**(*project_dir: Union[str, pathlib.Path]*)

Bases: *intercom_test.aws_http.HandlerMapper*

A *HandlerMapper* drawing information from a Serverless project config

Parameters **project_dir** – root directory of the Serverless project

The typical usage of this class is with *InterfaceCaseProvider*, as:

```
import intercom_test.aws_http

def around_interface_case(case: dict):
    # Some kind of setup, possibly using *case*
    try:
        yield
    finally:
        # The corresponding teardown

def test_interface_entrypoints():
    case_provider = intercom_test.InterfaceCaseProvider(<args>)
    service = intercom_test.aws_http.ServerlessHandlerMapper(<path-to-serverless-
↪project>)
    for test_runner in case_provider.case_runners(
        service.case_tester(case_env=around_interface_case)
    ):
        yield (test_runner,)
```

Note that the `case_env=` handler can set the *RESPONSE_BODY_EXPECTATIONS* key in the case to customize validation of the generated response body if there is reason to do so.

case_tester(*api_style: Optional[Callable] = None*, ***kwargs*) → Callable[[dict], None]
Convenience method for applying the HTTP API event adapter/testing logic

The result of this method is intended to be passed to *intercom_test.framework.InterfaceCaseProvider.case_runners()*.

See *HttpCasePreparer* and *CasePreparer* for information on keys of the test case that are consulted in constructing the Lambda Function input event. See *confirm_expected_response()* for information on keys of the test case consulted when evaluating the Lambda Function response.

config_file = 'serverless.yml'

map(*method: str*, *path: str*) → Callable[[dict, dict], dict]
Use routing defined in the Serverless config to map a handler

project_dir
Directory of the project

exception intercom_test.aws_http.**UnexpectedResponseBody**
Bases: AssertionError

Raised when the expected HTTP response was not generated

ATTRIBUTES = 'actual expected'

actual
Index 0 argument to the constructor

expected
Index 1 argument to the constructor

intercom_test.aws_http.**ala_http_api**(*handler_mapper: intercom_test.aws_http.HandlerMapper*, *context: Optional[dict] = None*, *case_env=None*) → Callable[[dict], None]
Build a case tester from a *HandlerMapper* for an HTTP API

Parameters

- **handler_mapper** – maps from method and path to an AWS Lambda handler function
- **context** – optional context information to pass to the identified handler
- **case_env** – context function for setup/teardown with access to the test case

The *case_env* (if given) must be either a generator function that yields a single time or a callable returning a *context manager*. If a generator function is given, it is converted to a context manager constructor with `contextlib.contextmanager()`. In either case, the context manager constructor is invoked with the test case data `dict` around invocation of the handler callable.

See *HttpCasePreparer* and *CasePreparer* for information on keys of the test case that are consulted in constructing the Lambda Function input event. See *confirm_expected_response()* for information on keys of the test case consulted when evaluating the Lambda Function response.

intercom_test.aws_http.**ala_rest_api**(*handler_mapper: intercom_test.aws_http.HandlerMapper*, *context: Optional[dict] = None*, *case_env=None*) → Callable[[dict], None]
Build a case tester from a *HandlerMapper* for a REST API

Parameters

- **handler_mapper** – maps from method and path to an AWS Lambda handler function
- **context** – optional context information to pass to the identified handler

- **case_env** – context function for setup/teardown with access to the test case

The *case_env* (if given) must be either a generator function that yields a single time or a callable returning a *context manager*. If a generator function is given, it is converted to a context manager constructor with `contextlib.contextmanager()`. In either case, the context manager constructor is invoked with the test case data `dict` around invocation of the handler callable.

See *RestCasePreparer* and *CasePreparer* for information on keys of the test case that are consulted in constructing the Lambda Function input event.

intercom_test.aws_http.**confirm_expected_response**(*handler_result: dict*, *case: dict*) → None

Confirm that the (normalized) output of the handler meets case expectations

Parameters

- **handler_result** – result from the Lambda Function handler
- **case** – the test case data

Normalization of the handler function output *does not occur* in this function; normalize *handler_result* before passing it in.

The following keys in *case* are consulted when evaluating the Lambda Function response:

'response status' (`int`) The HTTP response status code number expected, defaulting to 200

'response headers' (`dict` or list of 2-item lists) HTTP headers required in the response; if a header is listed here and is returned as a multi-value header (in `'multiValueHeaders'`), the *set* of values in the response is expected to match the *set* of values listed here in the test case

'response body' (**required**) A `str`, `bytes`, or JSONic data type giving the expected body of the response; JSONic data is compared against the response by parsing the body of the response as JSON, then comparing to the data given here in the test case

intercom_test.cases module

class intercom_test.cases.**IdentificationListReader**(*key_fields*, *, *safe_loading=None*)

Bases: `object`

Utility class to read case ID and associated events from a YAML event stream

This class is used internally to identify test cases when correlating the test case with existing augmentation data for editing. In that case, both the Python-native representation of the test case (for *hash_from_fields()*) and the YAML event stream for the key/value pairs (to preserve as much format from the source file) are needed.

class State

Bases: `enum.Enum`

An enumeration.

content = 2

header = 1

tail = 3

read(*event*)

safe_loading = True

intercom_test.cases.**hash_from_fields**(*test_case*)

Compute a string hash from any acyclic, JSON-ic `dict`

Parameters **test_case** (*dict*) – test case data to be hashed

Returns a repeatably generatable hash of *test_case*

Return type str

The hash is computed by encoding *test_case* in ASN1 DER (see `json_asn1.types.ASN1_SOURCE` for the ASN1 syntax of the data format), then hashing with SHA-256, and finally Base64 encoding to get the result.

Note that this function hashes **all** key/value pairs of *test_case*.

intercom_test.exceptions module

exception intercom_test.exceptions.**DataParseError**
Bases: Exception

Raised when a case augmentation file is incorrectly structured

exception intercom_test.exceptions.**MultipleAugmentationEntriesError**
Bases: Exception

Raised when the same case is augmented in multiple places

exception intercom_test.exceptions.**NoAugmentationError**
Bases: ValueError

Raised when lack of augmentation data prevents a requested operation

intercom_test.foreign module

class intercom_test.foreign.**Config**(*filepath*)
Bases: object

Configuration for command line interface

CASE_AUGMENTATION_KEYS = frozenset({'request keys', 'augmentation data'})

classmethod build_with_cui(*filepath*)

case_augmenter = None

request_keys = ()

class intercom_test.foreign.**DirPicker**(*what_for*, *start_dir='.'*, ***, *valid=None*)
Bases: *intercom_test.foreign.Menu*

SELECTION_OPTION = '<this directory>'

run()

selected_path = None

class intercom_test.foreign.**Menu**(*what_for*)
Bases: object

exception MenuCanceled
Bases: Exception

run(*options*)

intercom_test.foreign.**commit_updates**(*options*)
usage: {program} commitupdates [options]

Commit the augmentation updates to the compact files

Options:

-c CONFFILE, --config CONFFILE path to configuration file

intercom_test.foreign.**csmain**()

intercom_test.foreign.**enumerate**(*options*)
usage: {program} enumerate [options]

Enumerate all test cases, including any configured augmentation data

Options:

-c CONFFILE, --config CONFFILE path to configuration file

-o FORMAT, --output FORMAT format of output, e.g. yaml, jsonl [default: yaml]

intercom_test.foreign.**http_stub_exchange**(*options*)
usage: {program} hjx-stubber [options]

Each line of JSON Lines input on STDIN is treated as a request and the corresponding response is written to STDOUT, also as JSON Lines. If the request is successfully matched against the test cases, the matching test case will be returned; in this case a 'response status' key in the response (which defaults to 200 if not specified by the test case) is guaranteed. If no matching test case is found, there will not be a 'response status' key and the returned JSON will describe how the request can be modified to come closer to one or more test cases.

This subcommand is only intended to be used with HTTP retrieval of JSON or HTTP exchanges of JSON, and no provision is made here for binary data in the response body.

To use additional keys in matching requests (other than *method*, *url*, and *request body*), give the keys as a sequence under *request keys* in the config file. This interacts with the consultation of augmentation data: if using augmentation data, make sure to also list *method*, *url*, and *request body* under *request keys*.

Options:

-c CONFFILE, --config CONFFILE path to configuration file

intercom_test.foreign.**init**(*options*)
usage: {program} init [options]

Interactively create a configuration file

Options:

-c CONFFILE, --config CONFFILE path to configuration file

intercom_test.foreign.**main**(**args*)

intercom_test.foreign.**merge_cases**(*options*)
usage: {program} mergecases [options]

Merge all extension test case files into the main test case for for the service.

Options:

-c CONFFILE, --config CONFFILE path to configuration file

intercom_test.foreign.**subcommand**()

intercom_test.framework module

class intercom_test.framework.**CaseAugmenter**(*augmentation_data_dir*)
Bases: object

Base class of case augmentation data managers

This class uses and manages files in a case augmentation directory. The data files are intended to either end in '.yml' or '.update.yml'. The version control system should, typically, be set up to ignore files with the '.update.yml' extension. These two kinds of files have a different "data shape".

Update files (ending in '.update.yml') are convenient for manual editing because they look like the test case file from which the case came, but with additional entries in the case data `dict`. The problems with long term use of this file format are A) it is inefficient for correlation to test cases, and B) it duplicates data from the test case, possibly leading to confusion when modifying the .update.yml file does not change the test case.

Compact data files (other files ending in '.yml') typically are generated through this package. The format is difficult to manually correlate with the test file, but does not duplicate all of the test case data as does the update file data format. Instead, the relevant keys of the test case are hashed and the hash value is used to index the additional augmentation value entries.

It is an error for a test case to have multiple augmentations defined within .yml files (excluding .update.yml files), whether in the same or different files. It is also an error for multiple files with the .update.yml extension to specify augmentation for the same case, though within the same file the last specification is taken. When augmentations for a case exist within both one .update.yml and one .yml file, the .update.yml is used (with the goal of updating the .yml file with the new augmentation values).

Methods of this class depend on the class-level presence of `CASE_PRIMARY_KEYS`, which is not provided in this class. To use this class's functionality, derive from it and define this constant in the subclass. Two basic subclasses are defined in this module: *HTTPCaseAugmenter* and *RPCCaseAugmenter*.

__init__(*augmentation_data_dir*)
Constructing an instance

Parameters **augmentation_data_dir** – path to directory holding the augmentation data

UPDATE_FILE_EXT = '.update.yml'

augmentation_data_dir

augmented_test_case(*test_case*)
Add key/value pairs to *test_case* per the stored augmentation data

Parameters **test_case** (*dict*) – The test case to augment

Returns Test case with additional key/value pairs

Return type dict

augmented_test_case_events(*case_key*, *case_id_events*)
Generate YAML events for a test case

Parameters

- **case_key** (*str*) – The case key for augmentation
- **case_id_events** – An iterable of YAML events representing the key/value pairs of the test case identity

This is used internally when extending an updates file with the existing data from a case, given the ID of the case as YAML.

extend_updates(*file_name_base*)
Create an object for extending a particular update file

The idea is:

```
case_augmenter.extend_updates('foo').with_current_augmentation(sys.stdin)
```

classmethod key_of_case(*test_case*)
Compute the key (hash) value of the given test case

safe_loading = True

update_compact_files()
Update compact data files from update data files

class intercom_test.framework.**HTTPCaseAugmenter**(*augmentation_data_dir*)
Bases: *intercom_test.framework.CaseAugmenter*

A *CaseAugmenter* subclass for augmenting HTTP test cases

CASE_PRIMARY_KEYS = frozenset({'request body', 'url', 'method'})

class intercom_test.framework.**InterfaceCaseProvider**(*spec_dir*, *group_name*, *, *case_augmenter=None*)
Bases: object

Test case data manager

Use an instance of this class to:

- Generate test case data dicts
- Decorate the case runner function (if auto-updating of compact augmentation data files is desired)
- Merge extension test case files to the main test case file
- Other case augmentation management tasks

Setting *use_body_type_magic* to True automatically parses the "request body" value as JSON if "request type" in the same test case is "json", and similarly for "response body" and "response type".

__init__(*spec_dir*, *group_name*, *, *case_augmenter=None*)
Constructing an instance

Parameters

- **spec_dir** – File system directory for test case specifications
- **group_name** – Name of the group of tests to load
- **case_augmenter** – *optional* An object providing the interface of a *CaseAugmenter*

The main test case file of the group is located in *spec_dir* and is named for *group_name* with the '.yml' extension added. Extension test case files are found in the *group_name* subdirectory of *spec_dir* and all have '.yml' extensions.

case_augmenter
The *CaseAugmenter* instance used by this object, if any

case_runners(*fn*, *, *do_compact_updates=True*)
Generates runner callables from a callable

The callables in the returned iterable each call *fn* with all the positional arguments they are given, the test case dict as an additional positional argument, and all keyword arguments passed to the case runner.

Using this method rather than *cases()* directly for running tests has two advantages:

- The default of *do_compact_updates* automatically applies *update_compact_augmentation_on_success()* to *fn*
- Each returned runner callable will log the test case as YAML prior to invoking *fn*, which is helpful when updating the augmenting data for the case becomes necessary

cases()
Generates dicts of test case data

This method reads test cases from the group's main test case file and auxiliary files, possibly extending them with augmented data (if *case_augmentations* was given in the constructor).

extension_files()
Get an iterable of the extension files of this instance

group_name
Name of group of test cases to load for this instance

main_group_test_file
Path to the main test file of the group for this instance

merge_test_extensions()
Merge the extension files of the target group into the group's main file

safe_yaml_loading = True

spec_dir
The directory containing the test specification files for this instance

update_compact_augmentation_on_success(*fn*)
Decorator for activating compact data file updates

Using this decorator around the test functions tidies up the logic around whether to propagate test case augmentation data from update files to compact files. The compact files will be updated if all interface tests succeed and not if any of them fail.

The test runner function can be automatically wrapped with this functionality through *case_runners()*.

update_compact_files()
Calls the *CaseAugmenter* to apply compact data file updates

Raises ***NoAugmentationError*** – when no case augmentation data was specified during construction of this object

use_body_type_magic = False

class intercom_test.framework.**RPCCaseAugmenter**(*augmentation_data_dir*)
Bases: *intercom_test.framework.CaseAugmenter*

A *CaseAugmenter* subclass for augmenting RPC test cases

CASE_PRIMARY_KEYS = frozenset({'endpoint', 'request parameters'})

class intercom_test.framework.**UpdateExtender**(*file_name_base*, *case_augmenter*, *, *safe_loading=None*)
Bases: object

file_name

safe_loading = True

with_current_augmentation(*stream*)
Append the full test case with its current augmentation data to the target file

Parameters **stream** – A file-like object (which could be passed to yaml.parse())

The *stream* contains YAML identifying the test case in question. The identifying YAML from the test case _plus_ the augmentative key/value pairs as currently defined in the augmenting data files will be written to the file *file_name*.

intercom_test.framework.**data_files**(*dir_path*)
Generate data file paths from the given directory

intercom_test.framework.**extension_files**(*spec_dir*, *group_name*)
Iterator of file paths for extensions of a test case group

Parameters

- **spec_dir** – Directory in which specifications live
- **group_name** – Name of the group to iterate

intercom_test.http_best_matches module

Module for finding nearest imperfect match for HTTP request

class intercom_test.http_best_matches.**AvailableAdditionalFieldsReport**(*available_value_sets*)
Bases: *intercom_test.http_best_matches.Report*

as_jsonic_data()
Convert this report to JSON data

class intercom_test.http_best_matches.**AvailableHttpMethodsReport**(*methods*)
Bases: *intercom_test.http_best_matches.Report*

as_jsonic_data()
Convert this report to JSON data

class intercom_test.http_best_matches.**AvailableJsonRequestBodiesReport**(*diff_case_pairs*)
Bases: *intercom_test.http_best_matches.Report*

as_jsonic_data()
Convert this report to JSON data

class intercom_test.http_best_matches.**AvailablePathsReport**(*test_case_groups*)
Bases: *intercom_test.http_best_matches.Report*

as_jsonic_data()
Convert this report to JSON data

class intercom_test.http_best_matches.**AvailableQueryStringParamsetsReport**(*deltas*)
Bases: *intercom_test.http_best_matches.Report*

as_jsonic_data()
Convert this report to JSON data

class intercom_test.http_best_matches.**AvailableScalarRequestBodiesReport**(*test_cases*)
Bases: *intercom_test.http_best_matches.Report*

as_jsonic_data()
Convert this report to JSON data

class intercom_test.http_best_matches.**Database**(*cases: Iterable[dict]*, ***, *add_request_keys=()*)
Bases: object

best_matches(*request: dict*, ***, *timeout: float = 0.3*) → dict
Given HTTP request parameters, find the best known match

get_case(*request: dict*)

json_exchange(*request_json*, *reply_stream*)

class intercom_test.http_best_matches.**JsonComparer**(*ref*)
Bases: object

Utility to compare one JSON document with several others

CONGRUENT_DATA = 'congruent options'

class Delta

Bases: tuple

Differences between two JSON documents

This difference always consists of three parts, in decreasing order of precedence:

- Changes to which substructures are present,
- Changes to where substructures are located, and
- Changes to scalar values.

Only one of these three will be non-empty.

Use distance() to get a sortable distance measure for this delta.

edit_distance()

scalar_diffs

structure_diffs

structure_location_diffs

diff(*case*) → intercom_test.http_best_matches.JsonComparer.Delta

Diffs two JSON documents

The difference evaluation proceeds in three steps, with each of the later steps proceeding only if the earlier step produced no differences.

The steps are:

- All substructures (lists and dicts, with correct subitem signatures) are present.
- All substructures are in the correct locations.
- Each scalar value location holds the expected scalar value.

To reflect this, the differences are returned as a tuple of three Sequence`s wrapped in a :class:.JsonComparer.Delta‘, only one of which will contain any items:

- Changes to which substructures are present.
- Changes to where substructures are located.
- Changes to scalar values.

class intercom_test.http_best_matches.**JsonMap**(*json_data*)

Bases: object

items_from_signature(*sig*)

scalars

Indexes correspond with scalar_key_paths()

substruct_key_paths

Indexes correspond with substruct_signatures()

substruct_locations

substruct_signatures

Indexes correspond with substruct_key_paths()

class intercom_test.http_best_matches.**JsonType**
Bases: enum.IntEnum

An enumeration.

NoneType = 1

construct()

dict = 6

float = 4

int = 3

is_collection

list = 5

str = 2

class intercom_test.http_best_matches.**JsonWalker**
Bases: object

walk(*json_data*)

class intercom_test.http_best_matches.**QStringComparer**(*qsparams: Sequence[Tuple[str, str]]*)
Bases: object

Utility to compare one URL query string with several others

class Delta
Bases: tuple

edits

mods

params

diff(*case_qsparams: Sequence[Tuple[str, str]]*)

class intercom_test.http_best_matches.**Report**
Bases: abc.ABC

as_jsonic_data()
Convert this report to JSON data

intercom_test.http_best_matches.**lookup_json_type**()
Return the value for key if key is in the dictionary, else default.

intercom_test.utils module

class intercom_test.utils.**FilteredDictView**(*d*, ***, *key_filter=None*, *value_transform=None*)
Bases: object

dict-like access to a key-filtered and value-transformed dict

Only _viewing_ methods are supported, not modifications.

class Items(*dview*)
Bases: object

class Keys(*dview*)
Bases: object

class Values(*dview*)
Bases: object

get(*k*, *defval=None*)

items()

keys()

values()

intercom_test.utils.**attributed_error**(*cls*)
Expose exception instance constructor arguments (or specified names) as properties

If the only purpose of the exception class constructor would be to generate properties from the argument names, the ATTRIBUTES attribute of the class can be assigned with either an iterable of str or a single str (which will be str.split()) to more concisely specify the names to map to the arguments passed to the constructor.

intercom_test.utils.**complex_test_context**(*fn*)
Decorator to facilitate building a test environment through context managers

The callable decorated should accept a *test case* and a *context entry callable*. The test case is simply passed through from the wrapper. The context entry callable should be called on a context manager to enter its context, and all contexts will be exited in reverse order when the decorated function exits. This compares to the defer statement in the Go programming language or scope(exit) at the function level for the D programming language.

Example:

```
@complex_test_context
def around_interface_case(case, setup):
    setup(database_fixtures(case))
    setup(stubs(case))

    yield
```

intercom_test.utils.**def_enum**(*fn*)
Decorator allowing a function to DRYly define an enumeration

The decorated function should not require any arguments and should return an enumeration source, which will be passed to enum.Enum along with the name of the decorated function. The resulting enum.Enum-derived class will be returned.

The value returned by *fn* can be any kind of *source* accepted by the functional API of enum.Enum.

intercom_test.utils.**open_temp_copy**(*path*, *binary=False*, *, *blocksize=None*)
Make a temporary copy of *path* and return the opened file

The returned file object will be opened with mode 'w+' or 'w+b' (depending on *binary*) and will be positioned at the beginning of the file contents. If specified, *blocksize* indicates the size of the buffer to use (in bytes) when making the copy.

intercom_test.utils.**optional_key**(*mapping*, *key*)
Syntactic sugar for working with dict keys that *might* be present

Typical usage:

```
for value in optional_key(d, 'answer'):
    # Body executed once, with *value* assigned ``d['answer']``, if
```

(continues on next page)

(continued from previous page)

```
    # *d* contains ``'answer'``.  The body of the ``for`` is not
    # executed at all, otherwise.
    print(f"The answer: {value}")
```

intercom_test.version module

class intercom_test.version.**VersionInfo**(*version*)

Bases: object

base_version

get_modified_status()

get_version_unknown()

git_dir = '/home/docs/checkouts/readthedocs.org/user_builds/intercom-test/checkouts/la

modified = False

version_tag

version_unknown = False

intercom_test.yaml_tools module

class intercom_test.yaml_tools.**EventsToNodes**(*events*)

Bases: yaml.composer.Composer, yaml.resolver.Resolver

check_event(**choices*)

dispose()

get_event()

peek_event()

intercom_test.yaml_tools.**content_events**(*value*)

Return an iterable of events presenting *value* within a YAML document

intercom_test.yaml_tools.**get_load_all_fn**(*, *safe=True*)

intercom_test.yaml_tools.**get_load_fn**(*, *safe=True*)

intercom_test.yaml_tools.**value_from_event_stream**(*content_events*, *, *safe_loading=True*)

Convert an iterable of YAML events to a Pythonic value

The *content_events* MUST NOT include stream or document events.

1.1.3 Module contents

Intercomponent Testing (Interface by Example)

The main functionality of this package is accessible through *InterfaceCaseProvider*. *CaseAugmenter* and it's predefined subclasses, typically necessary for testing *service provider* code, are also available from this module. These classes come from *framework* but are imported into the base namespace of this package for ease of use.

For cross-language compatibility, the ASN1 source for encoding JSON values is available from this module as JSON_ASN1_SOURCE.

CHAPTER 2

`icy-test` Command Line Tool

Not every test harness is written in Python. To accommodate this, the *intercom_test package* can be installed to provide a command line tool call `icy-test` that provides access to the core functionality.

2.1 Installation

To install the `icy-test` command line tool, simply install *intercom_test package* with the *[cli] extra*, e.g.:

```
pip install intercom_test[cli]
```

This creates a command line tool named `icy-test`, which can be run with the `--help` flag to get usage information. This information will be the most recent and detailed available.

2.2 Configuration File

`icy-test` needs a configuration file to provide information that would, in a typical Python testing setting, be provided as parameters to the *`InterfaceCaseProvider`* constructor. The path to this file is specified with the `-c` or `--config` flag when running `icy-test`.

A text-mode helper for building a configuration file (which is a YAML file, usually with a `.yml` extension) is provided as `icy-test init`, and requires specifying a config file using one of the options mentioned above.

2.3 Consuming Test Cases

The main use of `icy-test` is to access the test cases. These are available in the output of `icy-test enumerate` in either a stream of YAML documents (one per test case) or as JSON Lines (each line contains a JSON document).

2.4 Committing Augmentation Data Updates

Where *InterfaceCaseProvider* used within a Python testing framework can provide *case runners* that can automatically update the compact augmentation data files when all test cases have passed, no such facility is easily implemented when consuming the test cases from another process and/or language. The augmentation data changes embodied in the *update files* need to be explicitly committed to the *compact files* by running `icy-test commitupdates`.

2.5 Merging Interface Extension Test Cases To Main File

Use the `icy-test mergecases` subcommand to invoke *intercom_test.framework.InterfaceCaseProvider.merge_test_extensions()* with appropriate setup taken from the `icy-test` configuration file.

2.6 Access HTTP JSON Exchange Stubs Outside Python

Because solutions involving exchanges of JSON documents over HTTP are becoming very popular, `icy-test` provides a subcommand to offload the logic of matching the elements of the HTTP request (method, URL (path and query string), and sometimes request body) with a test case. Moreover, `icy-test hjx-stubber` will, when given a request that *doesn't* exist in the test case set, respond with information on how the request can be changed to one that is in the test case set.

If changing the method, URL, and request body do not provide enough dimensions of control to adequately represent the gamut of request/response pairs for the represented service, `icy-test hjx-stubber` does reference the `request keys` configuration file entry, which can be used to add fields to the "test case key." An example would be listing `story` as a request key, then populating test cases that share the same method, URL, and request body with individual values for the the `story` field. To fully implement this, the interface-consuming project has to be willing to inject a `story` field into the request line passed to `icy-test hjx-stubber` during testing.

`icy-test hjx-stubber` accepts a request formatted as a JSON object on a single line (i.e. JSON Lines), where at least `method` and `url` properties are present. It will respond with a similar JSON Lines object which is either the full, matching test case (plus a `response status` field if one was not specified in the data files) or a set of diffs for the closest test cases `icy-test hjx-stubber` could find in the whole case set. See `icy-test hjx-stubber --help` for more information.

Starting up `icy-test hjx-stubber` is somewhat expensive for large sets of test cases, so it is best to start it when spinning up the test environment for a run of tests, then shut it down when testing finishes. Closing standard input is enough to get the program to exit.

CHAPTER 3

Support for Testing Serverless API Services on AWS

Many API services are currently hosted on AWS, and Serverless is one common Infrastructure-as-Code (IaC) system for organizing the bevy of resources necessary for a "serverless" service. Custom code for handling API requests in a Serverless application is integrated through Lambda Functions, which have a simple call interface in several languages. Where the language chosen is Python, using the *intercom_test package* allows development of the interface test cases in familiar HTTP terms but, through `ServerlessHandlerMapper`, allows the Lambda handler functions to be tested.

3.1 Extended Example

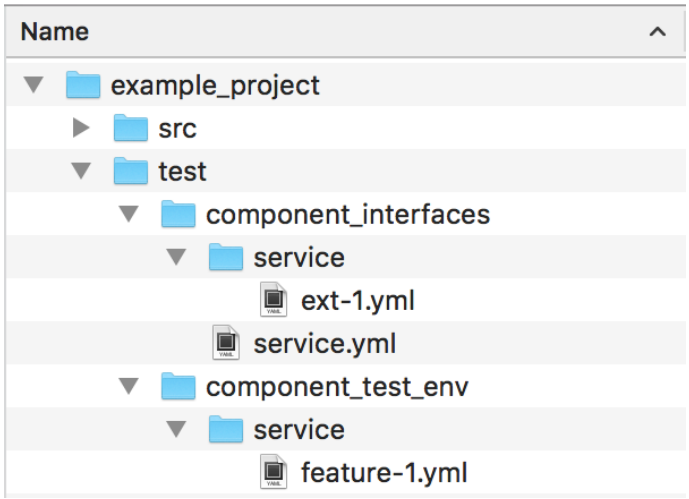

Fig. 1: Example directory tree for using `intercom_test`

Building on the *base example*, if we had a `serverless.yml` file in the `src` directory, we could create test code like:

```
from contextlib import ExitStack

from unittest import TestCase
from intercom_test import InterfaceCaseProvider, HTTPCaseAugmenter, aws_http, utils␣
↪as icy_utils

@icy_utils.complex_test_context
def around_interface_case(case, setup):
    setup(database(case))
    setup(stubs(case))

    yield

# ... define `database` and `stubs` to return context managers for the
# test case data they are given ...

class InterfaceTests(TestCase):
    def test_interface_case(self):
        # Construct an AWS Lambda handler function mapper from a Serverless
        # configuration (targeting the "aws" provider)
        service = aws_http.ServerlessHandlerMapper("src")

        # Get the case-testing callable, which accepts an entry (a dict)
        # from the test data file(s)
        case_tester = service.case_tester(case_env=around_interface_case)

        # Construct the case provider
        case_provider = InterfaceCaseProvider(
            "test/component_interfaces", "service",
            case_augmenter=HTTPCaseAugmenter("test/component_test_env/service")
        )

        # Use case_provider to construct a generator of case runner callables
        case_runners = case_provider.case_runners(case_tester)

        for i, run_test in enumerate(case_runners):
            with self.subTest(i=i):
                run_test()

if __name__ == '__main__':
    unittest.main()
```

The callable returned from *intercom_test.aws_http.ServerlessHandlerMapper.case_tester()* accepts a `dict` of test case data. It does not care where this `dict` comes from, but an *intercom_test.framework.InterfaceCaseProvider* is specifically designed to provide such a value.

Certain keys of the test case `dict` are consulted (documented in *intercom_test.aws_http.HttpCasePreparer*) when building the event passed to the handler function, and certain other keys (documented in *intercom_test.aws_http.confirm_expected_response()*) are used for evaluating correctness of the handler function's result.

CHAPTER 4

Using This Package

intercom_test provides *InterfaceCaseProvider* to iterate over test cases defined in YAML files. With the additional use of a *case_augmenter* – either an *HTTPCaseAugmenter*, a *RPCCaseAugmenter*, or your own class derived from *CaseAugmenter* – the *InterfaceCaseProvider* can add more data from a different directory to any test case; this supports decoupling a *service provider's* implementation details necessary to passing the given test case from the request and response information needed by both the *consumer* and the *provider*.

intercom_test, when installed with the `[cli]` *extra*, also provides a command line tool called `icy-test`. This tool makes the core functionality of *intercom_test* available to programs written in languages other than Python.

CHAPTER 5

What It Looks Like In Practice

The *intercom_test* package does not make many requirements of the directory structure for the project using it, but here is one example of how a project could be structured to use *intercom_test*.

This package has no direct concern with the `src` folder. Within the `test` folder, this example project has split the data managed by *intercom_test* into `component_interfaces` and `component_test_env`. The structure, relationship, and usage of these two folders are described below.

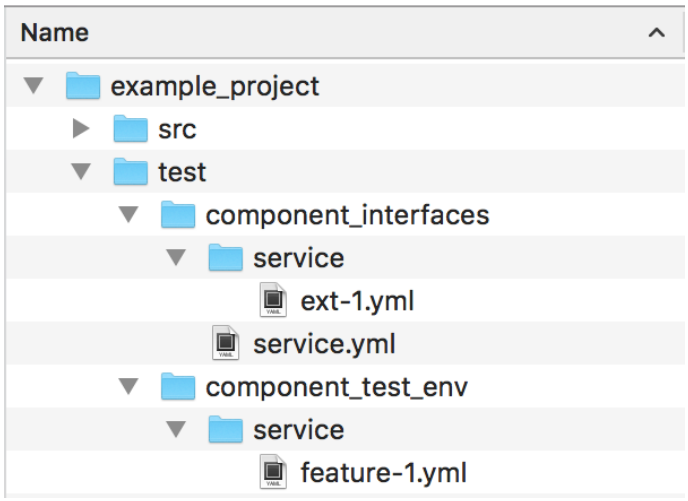

Fig. 1: Example directory tree for using *intercom_test*

In this example, the `component_interfaces` contain both a *main* test case file, (`$PROJECT/test/component_interfaces/service.yml`) and also an extension file (`$PROJECT/test/component_interfaces/service/ext-1.yml`), which will be combined into a single set of test cases by *intercom_test*. Any additional `.yml` files added to the `$PROJECT/test/component_interfaces/service` folder will also be read as additional test cases for *service*. Typically, the `component_interfaces` folder would be shared with the projects intending to consume this service via some version control mechanism like a Git submodule or a Subversion *externals definition*. When constructing an *InterfaceCaseProvider* with the `example_project` directory as the current directory, the first argument should be `"test/component_interfaces"` and the second argument `"service"` (indicating both the `service.yml` file *and* all files matching `service/*.yml`).

Additionally, the `component_test_env` subfolder contains information necessary for testing the service-provider, but which doesn't affect the component interface – database fixtures, data for mocking external services, etc. Within this folder, the *augmentation* data for the "service" is the set of YAML (`.yml`) files located in the `service` folder; it is important to note that the augmentation data will come from *all* files within the given folder, so having a separate

folder for each interface is probably wise. The path to pass when constructing one of the standard *CaseAugmenter* subclasses would therefore be `"test/component_test_env/service"`. Since this information is only important for testing the provider and is tightly coupled to the provider implementation, the entire `component_test_env` folder should probably be a part of the service provider's source code repository and not shared with service consumers. This example only contains one such file, named `feature-1.yml`, but as many augmentation files as desired can be created, though there is a restriction on augmenting a single test case in multiple separate files.

So, assuming this project represents some kind of HTTP API, the most logical way to create the *InterfaceCaseProvider* for this directory structure is:

```
from intercom_test import InterfaceCaseProvider, HTTPCaseAugmenter

...

def get_interface_case_provider():
    return InterfaceCaseProvider(
        "test/component_interfaces", "service",
        case_augmenter=HTTPCaseAugmenter("test/component_test_env/service")
    )

...
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

T

U

V

W