charlatan Documentation

Release 0.4.6

Charles-Axel Dein

September 22, 2015

Contents

Charlatan is a library that lets you efficiently manage and install fixtures.

Its features include:

- Straightforward YAML syntax to define fixtures.
- Rich fixture definition functionalities, including inheritance and relationships (fixtures factory).
- ORM-agnostic. Tested with sqlalchemy, schematics, etc.
- Flexible thanks to *Hooks* or *Builders*.

Charlatan is a library that you can use in your tests to create database fixtures. Its aim is to provide a pragmatic interface that focuses on making it simple to define and install fixtures for your tests.

Charlatan supports Python 2 (only tested with 2.7) and 3 (tested with 3.3).

Why Charlatan? Since "charlatan" used to define "an itinerant seller of supposed remedies", we thought it would be a good name for a library providing fixtures for tests. Credit for the name goes to Zack Heller.

CHAPTER 1

Contents

1.1 Installation

From PyPI:

```
$ pip install charlatan
```

From sources:

```
$ git clone https://github.com/uber/charlatan.git
$ python setup.py install
```

1.2 Quickstart

1.2.1 A simple example

Let's say we have the following model:

```
class Toaster(object):

    def __init__(self, color, slots=2, content=None):
        self.color = color
        self.slots = slots
        self.content = content

    def __repr__(self):
        return "<Toaster '%s'>" % self.color


class User(object):

    def __init__(self, toasters):
        self.toasters = toasters
```

Let's define a very simple fixtures YAML file:

```
toaster:   # The fixture's name
  fields:  # The fixture's content
    color: red
    slots: 5
    content: !rel toasts  # You can reference other fixtures
```

```
  model: charlatan.tests.fixtures.simple_models:Toaster

toaster_green:
  # Charlatan also supports inheritance
  inherit_from: toaster
  fields:
    color: green

toasts:
  # No model is defined, so it defaults to what `fields` actually is, i.e.
  # in our case, a list.
  fields:
    - "Toast 1"
    - "Toast 2"
```

In this example:

- `toaster` and `toasts` are the fixture keys.
- `fields` is provided as argument when instantiating the class: `Toaster(**fields)`.
- `model` is the path to the model that we defined.
- `!rel` lets you create relationships by pointing to another fixture key.

You first need to load a fixtures file (do it once for the whole test suite) with *charlatan.FixturesManager.load()*:

```
>>> import charlatan
>>> fixtures_manager = charlatan.FixturesManager()
>>> fixtures_manager.load("./docs/examples/simple_fixtures.yaml",
...     models_package="toaster.models")
>>> toaster = fixtures_manager.install_fixture("toaster")
>>> toaster.color
'red'
>>> toaster.slots
5
>>> toaster.content
['Toast 1', 'Toast 2']
```

Voila!

1.2.2 Factory features

Charlatan provides you with factory features. In particular, you can override a fixture's defined attributes:

```
>>> toaster = fixtures_manager.install_fixture("toaster",
...     overrides={"color": "blue"})
>>> toaster.color
'blue'
```

You can also use inheritance:

```
>>> toaster = fixtures_manager.install_fixture("toaster_green")
>>> toaster.color
'green'
```

1.2.3 Using charlatan in test cases

Charlatan works best when used with `unittest.TestCase`. Your test class needs to inherit from *charlatan.FixturesManagerMixin*.

Charlatan uses an internal cache to store fixtures instance (in particular to create relationships). If you are resetting your database after each tests (using transactions or by manually truncating all tables), you need to clean the cache in `TestCase.setUp()`, otherwise *Charlatan* will try accessing objects that are not anymore in the sqlalchemy session.

```
import unittest

import charlatan

fixtures_manager = charlatan.FixturesManager()
fixtures_manager.load("./docs/examples/simple_fixtures.yaml")


class TestToaster(unittest.TestCase, charlatan.FixturesManagerMixin):

    def setUp(self):
        # Attach the fixtures manager to the instance
        self.fixtures_manager = fixtures_manager
        # Cleanup the cache
        self.init_fixtures()

    def test_example(self):
        """Verify that we can get fixtures."""
        toaster = self.install_fixture("toaster")
        self.assertEqual(toaster.color, "red")
        self.assertEqual(toaster.slots, 5)
        self.assertEqual(toaster.content, ['Toast 1', 'Toast 2'])
```

1.2.4 Using fixtures

There are multiple ways to require and use fixtures. When you install a fixture using the *charlatan.FixturesManagerMixin*, it gets attached to the instance and can be accessed as an instance attribute (e.g. `self.toaster`).

For each tests, in setUp and tearDown

```
class MyTest(FixturesManagerMixin):

    def setUp(self):
        # This will create self.toaster and self.brioche
        self.install_fixtures(("toaster", "brioche"))

    def test_toaster(self):
        """Verify that a toaster toasts."""
        self.toaster.toast(self.brioche)
```

For a single test

```
class MyTest(FixturesMixin):

    def test_toaster(self):
        self.install_fixture("toaster")
```

With pytest

It's extremely easy to use charlatan with pytest. There are multiple ways to achieve nice readability, here's one possibility.

In `conftest.py`:

```
import pytest


@pytest.fixture
def get_fixture(request):
    request.addfinalizer(fixtures_manager.clean_cache)
    return fixtures_manager.get_fixture
```

In your test file:

```
def test_toaster(get_fixture):
    """Verify that a toaster toasts."""
    toaster = get_fixture('toaster')
    toast = get_fixture('toast')
    ...
```

Getting a fixture without saving it

If you want to have complete control over the fixture, you can also get it without saving it nor attaching it to the test class:

```
class MyTest(FixturesManagerMixin):

    def test_toaster(self):
        self.toaster = self.get_fixture("toaster")
        self.toaster.brand = "Flying"
        self.toaster.save()
```

What happens when you install a fixture

Here's the default process (you can modify part or all of it using *Hooks* or *Builders*):

1. The fixture is instantiated: `Model(**fields)`.
2. If there's any post creation hook, they are run (see *Post creation* for more information).
3. The fixture is then saved. If it's a sqlalchemy model, charlatan will detect it, add it to the session and commit it (`db_session.add(instance); db_session.commit()`). If it's not a sqlalchemy model, charlatan will try to call a *save* method on the instance. If there's no such method, charlatan will do nothing.

Hooks are also supported.

Uninstalling fixtures

Because charlatan is not coupled with the persistence layer, it does not have strong opinions about resetting the world after a test runs. There's multiple ways to handle test tear down:

- Wrap test inside a transaction (if you're using sqlalchemy, its documentation has a good explanation about how to achieve that).
- Drop and recreate the database (not really efficient).
- Install and uninstall fixtures explicitly (you have to keep track of them though, if you forget to uninstall one fixture it will leak in the other tests). See *charlatan.FixturesManager.uninstall_fixture()*.

1.3 File format

charlatan only supports YAML at time of writing.

Fixtures are defined in a YAML file. Here is its general structure:

```
toaster:
  fields:
    brand: Flying
    number_of_toasts: 2
    toasts: [!rel toast1, !rel toast2]
    bought: !now -1y
  model: Toaster

toaster_already_in_db:
  id: 10
  model: Toaster

toast1:
  fields:
    bread: brioche
  model: .toast:Toast

toast2:
  fields:
    bread: campagne
  model: .toast:Toast

user:
  fields:
    name: Michel Audiard
    toaster: !rel toaster
  model: toaster.models.user:User
  post_creation:
    has_used_toaster: true
```

In this example:

- `toaster`, `toast1` and `toast2` are the fixture keys.
- `model` is where to get the model. Both relative and absolute addressing are supported
- `fields` are provided as argument when instantiating the class: `Toaster(**fields)`.
- `!rel` lets you create relationships by pointing to another fixture key.

- `!now` lets you enter timestamps. It supports basic operations (adding/subtracting days, months, years). It is evaluated when the fixture is instantiated.
- `!epoch_now` generates epoch timestamps and supports the same operations as `!now`.

Note: Inside `fields`, `!now` is supported only as a first level list item, or as a dictionary value.

1.3.1 Defining a fixture

A fixture has an identifier (in the example above, `toaster` is one of the fixture identifiers), as well as the following configuration:

- `fields`: a dictionary for which keys are attribute, and values are their values
- `model` gives information about how to retrieve the model
- `post_creation` lets you have some attribute values be assigned after instantiation.

1.3.2 Inheritance

Fixtures can inherit from other fixtures.

```
first:
  fields:
    foo: bar

# Everything is inherited from first.
second:
  inherit_from: first

# You can add fields without removing existing ones
third:
  inherit_from: first
  fields:
    # foo: bar is implied by inheritance
    toaster: toasted

# You can also overwrite the model.
fourth:
  inherit_from: first
  model: collections:Counter

# You can also overwrite both.
fifth:
  inherit_from: second
  fields:
    toaster: toasted
  model: collections:Counter
```

```
>>> import pprint
>>> from charlatan import FixturesManager
>>> manager = FixturesManager()
>>> manager.load("docs/examples/fixtures_inheritance.yaml")
>>> manager.get_fixture("first")
{'foo': 'bar'}
>>> manager.get_fixture("second")
{'foo': 'bar'}
```

```
>>> pprint.pprint(manager.get_fixture("third"))
{'foo': 'bar', 'toaster': 'toasted'}
>>> fourth = manager.get_fixture("fourth")
>>> fourth
Counter({'foo': 'bar'})
>>> fourth.__class__.__name__
'Counter'
>>> fifth = manager.get_fixture("fifth")
>>> fifth
Counter({'toaster': 'toasted', 'foo': 'bar'})
>>> fifth.__class__.__name__
'Counter'
```

If your fields are dict, then the first-level key will override everything, unless you use `deep_inherit`:

```
toaster:
  fields:
    toasts:
      toast1:
        type: brioche
        price: 10
        weight: 20

toaster2:
  inherit_from: toaster
  deep_inherit: true
  fields:
    toasts:
      toast1:
        type: bread
        # Because of deep_inherit, the following fields are implied:
        # price: 10
        # weight: 20
```

Example test:

```
from charlatan import FixturesManager


def test_deep_inherit():
    manager = FixturesManager()
    manager.load('./charlatan/tests/example/data/deep_inherit.yaml')
    toaster2 = manager.get_fixture('toaster2')
    assert toaster2['toasts']['toast1']['price'] == 10
    assert toaster2['toasts']['toast1']['weight'] == 20
```

New in version 0.4.5: You can use `deep_inherit` to trigger nested inheritance for dicts.

New in version 0.2.4: Fixtures can now inherits from other fixtures.

1.3.3 Having dictionaries as fixtures

If you don't specify the model, the content of `fields` will be returned as is. This is useful if you want to enter a dictionary or a list directly.

```
fixture_name:
  fields:
    foo: bar
```

```
fixture_list:
  fields:
    - "foo"
    - "bar"
```

```
>>> manager = FixturesManager()
>>> manager.load("docs/examples/fixtures_dict.yaml")
>>> manager.get_fixture("fixture_name")
{'foo': 'bar'}
>>> manager.get_fixture("fixture_list")
['foo', 'bar']
```

New in version 0.2.4: Empty models are allowed so that dict ands lists can be used as fixtures.

1.3.4 Getting an already existing fixture from the database

You can also get a fixture directly from the database (it uses `sqlalchemy`): in this case, you just need to specify the `model` and an `id`.

```
toaster_already_in_db:
  id: 10
  model: Toaster
```

1.3.5 Dependencies

If a fixture depends on some side effect of another fixture, you can mark that dependency (and, necessarily, ordering) by using the `depend_on` section.

```
fixture1:
  fields:
    - name: "foo"

fixture2:
  depend_on:
    - fixture1
  fields:
    - name: "bar"
  post_creation:
    - some_descriptor_that_depend_on_fixture1: true
```

New in version 0.2.7.

1.3.6 Post creation

Example:

```
user:
  fields:
    name: Michel Audiard
  model: User
  post_creation:
    has_used_toaster: true
    # Note that rel are allowed in post_creation
    new_toaster: !rel blue_toaster
```

For a given fixture, `post_creation` lets you change some attributes after instantiation. Here's the pseudo-code:

```
instance = ObjectClass(**fields)
for k, v in post_creation:
    setattr(instance, k, v)
```

New in version 0.2.0: It is now possible to use `rel` in post_creation.

1.3.7 Linking to other objects

Example:

```
toaster:
  model: Toaster
  fields:
    color: red

user:
  model: User
  fields:
    # You can link to another fixture
    toasters:
      - !rel toaster

toaster_colors:
  # You can also link to a specific attribute
  fields:
    color: !rel toaster.color
```

To link to another object defined in the configuration file, use `!rel`. You can link to another objet (e.g. `!rel toaster`) or to another object's attribute (e.g. `!rel toaster.color`).

```
>>> manager = FixturesManager()
>>> manager.load("docs/examples/relationships.yaml",
...     models_package="charlatan.tests.fixtures.simple_models")
>>> manager.get_fixture("user").toasters
[<Toaster 'red'>]
>>> manager.get_fixture("toaster_colors")
{'color': 'red'}
```

You can also link to specific attributes of collection's item (see *Collections of Fixtures* for more information about collections).

```
toaster_colors_list:
  fields: ['red']

# Let's define a collection
toasters:
  model: Toaster
  objects:
    red:
      color: red

toaster_from_collection:
  inherit_from: toaster
  fields:
    # You can link a specific attribute of a collection's item.
    color: !rel toasters.red.color
```

```
>>> manager.get_fixture("toaster_from_collection")
<Toaster 'red'>
```

New in version 0.2.0: It is now possible to link to another object' attribute.

1.3.8 Collections of Fixtures

Charlatan also provides more efficient way to define variations of fixtures. The basic idea is to define the model and the default fields, then use the `objects` key to define related fixtures. There's two ways to define those fixtures in the `objects` key:

- Use a list. You will then be able to access those fixtures via their index, e.g. `toaster.0` for the first item.
- Use a dict. The key will be the name of the fixture, the value a dict of fields. You can access them via their namespace: e.g. `toaster.blue`.

You can also install all of them by installing the name of the collection.

```
toasters:
  model: charlatan.tests.fixtures.simple_models:Toaster

  # Those are the default for all fixtures
  fields:
    slots: 5

  # You can have named fixtures in the collection. Note the use of dict.
  objects:
    green:  # This fixture can be accessed via toaster.green
      color: green
    blue:
      color: blue

anonymous_toasters:
  inherit_from: toasters

  # Here we define unamed fixtures. Note that we use a list instead of a dict.
  objects:
    # You access the first fixture via anonymous_toaster.0
    -
      color: yellow
    -
      color: black

# Those collections can be used as is in relationships.

collection:
  fields:
    # Since we defined the toasters collection as a dict, things's value will
    # be a dict as well
    things: !rel toasters

users:
  model: charlatan.tests.fixtures.simple_models:User

  objects:

    1:
      toasters: !rel anonymous_toasters
```

```
    2:
      # You can also link to specific relationships using the namespace
      toasters: [!rel toasters.green]

    3:
      toasters: [!rel anonymous_toasters.0]
```

Here's how you would use this fixture file to access specific fixtures:

```
>>> manager = FixturesManager()
>>> manager.load("docs/examples/collection.yaml")
>>> manager.get_fixture("toasters.green")
<Toaster 'green'>
>>> manager.get_fixture("anonymous_toasters.0")
<Toaster 'yellow'>
```

You can also access the whole collection:

```
>>> pprint.pprint(manager.get_fixture("toasters"))
{'blue': <Toaster 'blue'>, 'green': <Toaster 'green'>}
>>> manager.get_fixture("anonymous_toasters")
[<Toaster 'yellow'>, <Toaster 'black'>]
```

Like any fixture, this collection can be linked to in a relationship using the `!rel` keyword in an intuitive way.

```
>>> pprint.pprint(manager.get_fixture("collection"))
{'things': {'blue': <Toaster 'blue'>, 'green': <Toaster 'green'>}}
>>> user1 = manager.get_fixture("users.1")
>>> user1.toasters
[<Toaster 'yellow'>, <Toaster 'black'>]
>>> manager.get_fixture("users.2").toasters
[<Toaster 'green'>]
>>> manager.get_fixture("users.3").toasters
[<Toaster 'yellow'>]
```

Changed in version 0.3.4: Access to unnamed fixture by using a `.{index}` notation instead of `_{index}`.

New in version 0.3.4: You can now have list of named fixtures.

New in version 0.2.8: It is now possible to retrieve lists of fixtures and link to them with `!rel`

1.3.9 Loading Fixtures from Multiple Files

Loading fixtures from multiple files works similarly to loading collections. In this case, every fixture in a single file is preceded by a namespace taken from the name of that file. Relationships between fixtures in different files specified using the `!rel` keyword may be specified by prefixing the desired target fixture with its file namespace.

```
toaster:
  model: Toaster
  fields:
    color: red

user:
  model: User
  fields:
    # You can link to another fixture
    toasters:
      - !rel toaster
```

```
toaster_colors:
  # You can also link to a specific attribute
  fields:
    color: !rel toaster.color

toaster_colors_list:
  fields: ['red']

# Let's define a collection
toasters:
  model: Toaster
  objects:
    red:
      color: red

toaster_from_collection:
  inherit_from: toaster
  fields:
    # You can link a specific attribute of a collection's item.
    color: !rel toasters.red.color
```

```
toaster:
  model: Toaster
  fields:
    color: !rel relationships.toaster.color
```

```
>>> manager = FixturesManager()
>>> manager.load(["docs/examples/relationships.yaml",
...      "docs/examples/files.yaml"],
...      models_package="charlatan.tests.fixtures.simple_models")
>>> manager.get_fixture("files.toaster")
<Toaster 'red'>
```

New in version 0.3.7: It is now possible to load multiple fixtures files with `FixturesManager`

1.3.10 Datetime and timestamps

Use `!now`, which returns timezone-aware datetime. You can use modifiers, for instance:

- `!now +1y` returns the current datetime plus one year
- `!now +5m` returns the current datetime plus five months
- `!now -10d` returns the current datetime minus ten days
- `!now +15M` (note the case) returns the current datetime plus 15 minutes
- `!now -30s` returns the current datetime minus 30 seconds

For naive datetime (see the definition in Python's datetime module documentation), use `!now_naive`. It also supports deltas.

For Unix timestamps (seconds since the epoch) you can use `!epoch_now`:

- `!epoch_now +1d` returns the current datetime plus one year in seconds since the epoch
- `!epoch_now_in_ms` returns the current timestamp in milliseconds

All the same time deltas work.

New in version 0.4.6: `!epoch_now_in_ms` was added.

New in version 0.4.4: `!now_naive` was added.

New in version 0.2.9: It is now possible to use times in seconds since the epoch

1.3.11 Unicode Strings

New in version 0.3.5.

In python 2 strings are not, by default, loaded as unicode. To load all the strings from the yaml files as unicode strings, pass the option *use_unicode* as *True* when you instantiate your fixture manager.

1.4 Database fixtures

1.4.1 SQLAlchemy

Charlatan has been heavily used and tested with sqlalchemy. Here's a simple example:

Tests:

```
from charlatan import testing
from charlatan import FixturesManager
from charlatan.tests.fixtures.models import Session, Base, engine
from charlatan.tests.fixtures.models import Toaster

session = Session()
manager = FixturesManager(db_session=session)
manager.load("./charlatan/tests/example/data/sqlalchemy.yaml")


class TestSqlalchemyFixtures(testing.TestCase):

    def setUp(self):
        self.manager = manager

        # There's a lot of different patterns to setup and teardown the
        # database. This is the simplest possibility.
        Base.metadata.create_all(engine)

    def tearDown(self):
        Base.metadata.drop_all(engine)
        session.close()

    def test_double_install(self):
        """Verify that there's no double install."""
        self.manager.install_fixture('toaster')

        toaster = session.query(Toaster).one()
        assert toaster.color.name == 'red'
```

YAML file:

```
toaster:
  fields:
    color: !rel color
    name: "toaster1"
  model: charlatan.tests.fixtures.models:Toaster
```

```
color:
  fields:
    name: "red"
  model: charlatan.tests.fixtures.models:Color
```

Model definition:

```
from sqlalchemy import create_engine
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import sessionmaker, relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
engine = create_engine('sqlite:///:memory:')
Session = sessionmaker(bind=engine)


class Toaster(Base):

    __tablename__ = "toasters"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    color_id = Column(String, ForeignKey('colors.name'))

    color = relationship("Color", backref='toasters')


class Color(Base):

    __tablename__ = "colors"

    id = Column(Integer, primary_key=True)
    name = Column(String)
```

1.5 Hooks

The following hooks are available:

- `before_install`: called before doing anything. The callback takes no argument.
- `before_save`: called before saving an instance using the SQLAlchemy session. The callback takes a single argument which is the instance being saved.
- `after_save`: called after saving an instance using the SQLAlchemy session. The callback takes a single argument which is the instance that was saved.
- `after_install`: called after doing anything. The callback must accept a single argument that will be the exception that may have been raised during the whole process. This function is guaranteed to be called.
- `before_uninstall`: called before uninstalling fixtures. The callback takes no argument.
- `before_delete`: called before deleting an instance using either the SQLAlchemy session or in the following order *delete_instance* and *delete*. The callback takes a single argument which is the instance being deleted.
- `after_delete`: called after deleting an instance using either the SQLAlchemy session or in the following order *delete_instance* and *delete*. The callback takes a single argument which is the instance that was deleted.

- `after_uninstall`: called after uninstalling fixtures. The callback must accept a single argument that will be the exception that may have been raised during the whole process. This function is guaranteed to be called.

`FixturesManager.`**`set_hook`**(*hookname*, *func*)
Add a hook.

Parameters

- **hookname** (*str*) –
- **func** (*function*) –

1.6 Builders

Builders provide a powerful way to customize getting fixture. You can define your own builders and provide them as arguments when you instantiate *charlatan.FixturesManager*.

1.6.1 Example

Here's an example inspired by the schematics library, which expects a dict of attributes as a single instantiation argument:

```
from charlatan import FixturesManager
from charlatan.builder import Builder


class Toaster(object):

    def __init__(self, attrs):
        self.slots = attrs['slots']


class DictBuilder(Builder):

    def __call__(self, fixtures, klass, params, **kwargs):
        # A "normal" object would be instantiated this way:
        # return klass(**params)

        # Yet schematics object expect a dict of attributes as only argument.
        # So we'll do:
        return klass(params)


def test_custom_builder():
    manager = FixturesManager(get_builder=DictBuilder())
    manager.load('./charlatan/tests/example/data/custom_builder.yaml')
    assert manager.get_fixture('toaster').slots == 3
```

YAML file:

```
toaster:
  model: charlatan.tests.example.test_custom_builder:Toaster
  fields:
    slots: 3
    color: blue
```

1.6.2 API

class charlatan.builder.**Builder**

__call__(*fixtures*, *klass*, *params*, ***kwargs*)
Build a fixture.

Parameters

- **fixtures** (FixturesManager) –
- **klass** – the fixture's class (`model` in the definition file)
- **params** – the fixture's params (`fields` in the definition file)
- **kwargs** (*dict*) –

`kwargs` allows passing arguments to the builder to change its behavior.

class charlatan.builder.**DeleteAndCommit**

__call__(*fixtures*, *instance*, ***kwargs*)

delete(*instance*, *session*)
Delete instance.

class charlatan.builder.**InstantiateAndSave**

__call__(*fixtures*, *klass*, *params*, ***kwargs*)
Save a fixture instance.

If it's a SQLAlchemy model, it will be added to the session and the session will be committed.

Otherwise, a *`save()`* method will be run if the instance has one. If it does not have one, nothing will happen.

Before and after the process, the `before_save()` and `after_save()` hook are run.

instantiate(*klass*, *params*)
Return instantiated instance.

save(*instance*, *fixtures*, *session*)
Save instance.

1.7 API Reference

1.7.1 FixturesManager

class charlatan.**FixturesManager**(*db_session=None*, *use_unicode=False*, *get_builder=None*, *delete_builder=None*)
Manage Fixture objects.

Parameters

- **db_session** (*Session*) – sqlalchemy Session object
- **use_unicode** (*bool*) –
- **get_builder** (*func*) –

- **delete_builder** (*func*) –

New in version 0.4.0: `get_builder` and `delete_builder` arguments were added.

Deprecated since version 0.4.0: `delete_instance`, `save_instance` methods were deleted in favor of using builders.

New in version 0.3.0: `db_session` argument was added.

clean_cache()
Clean the cache.

delete_fixture(*fixture_key*, *builder=None*)
Delete a fixture instance.

Parameters

- **fixture_key** (*str*) –
- **builder** (*func*) –

Before and after the process, the `before_delete()` and `after_delete()` hook are run.

New in version 0.4.0: `builder` argument was added.

Deprecated since version 0.4.0: `delete_instance` method renamed to `delete_fixture` for consistency reason.

get_all_fixtures(*builder=None*)
Get all fixtures.

Parameters **fixture_keys** (*iterable*) –

Return type list of instantiated but unsaved fixtures

New in version 0.4.0: `builder` argument was added.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

get_fixture(*fixture_key*, *overrides=None*, *builder=None*)
Return a fixture instance (but do not save it).

Parameters

- **fixture_key** (*str*) –
- **overrides** (*dict*) – override fields
- **builder** (*func*) – build builder.

Return type instantiated but unsaved fixture

New in version 0.4.0: `builder` argument was added. `attrs` argument renamed `overrides`.

Deprecated since version 0.4.0: `do_not_save` argument was removed.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

get_fixtures(*fixture_keys*, *builder=None*)
Get fixtures from iterable.

Parameters **fixture_keys** (*iterable*) –

Return type list of instantiated but unsaved fixtures

New in version 0.4.0: `builder` argument was added.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

get_hook(*hook_name*)
Return a hook.

Parameters **hook_name** (*str*) – e.g. `before_delete`.

install_all_fixtures()
Install all fixtures.

Return type list of `fixture_instance`

Deprecated since version 0.4.0: `do_not_save` argument was removed.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

install_fixture(*fixture_key*, *overrides=None*)
Install a fixture.

Parameters

- **fixture_key** (*str*) –
- **overrides** (*dict*) – override fields

Return type `fixture_instance`

Deprecated since version 0.4.0: `do_not_save` argument was removed. `attrs` argument renamed `overrides`.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

install_fixtures(*fixture_keys*)
Install a list of fixtures.

Parameters **fixture_keys** (*str or list of strs*) – fixtures to be installed

Return type list of `fixture_instance`

Deprecated since version 0.4.0: `do_not_save` argument was removed.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

keys()
Return all fixture keys.

load(*filenames*, *models_package=''*)
Pre-load the fixtures. Does not install anything.

Parameters

- **filename** (*list_or_str*) – file or list of files that holds the fixture data
- **models_package** (*str*) – package holding the models definition

Deprecated since version 0.3.0: `db_session` argument was removed and put in the object's constructor arguments.

Changed in version 0.3.7: `filename` argument was changed to `filenames`, which can be list or string.

set_hook(*hookname*, *func*)
Add a hook.

Parameters

- **hookname** (*str*) –
- **func** (*function*) –

uninstall_all_fixtures()
Uninstall all installed fixtures.

Return type None

Deprecated since version 0.4.0: do_not_delete argument was removed. This function does not return anything.

uninstall_fixture(*fixture_key*)
Uninstall a fixture.

Parameters **fixture_key** (*str*) –

Return type None

Deprecated since version 0.4.0: do_not_delete argument was removed. This function does not return anything.

uninstall_fixtures(*fixture_keys*)
Uninstall a list of installed fixtures.

Parameters **fixture_keys** (*str or list of strs*) – fixtures to be uninstalled

Return type None

Deprecated since version 0.4.0: do_not_delete argument was removed. This function does not return anything.

1.7.2 FixturesManagerMixin

class charlatan.**FixturesManagerMixin**
Class from which test cases should inherit to use fixtures.

Changed in version 0.3.12: FixturesManagerMixin does not install class attributes fixtures anymore.

Changed in version 0.3.0: use_fixtures_manager method renamed init_fixtures.

Changed in version 0.3.0: Extensive change to the function signatures.

get_fixture(**args*, ***kwargs*)
Return a fixture instance (but do not save it).

Parameters

- **fixture_key** (*str*) –
- **overrides** (*dict*) – override fields
- **builder** (*func*) – build builder.

Return type instantiated but unsaved fixture

New in version 0.4.0: builder argument was added. attrs argument renamed overrides.

Deprecated since version 0.4.0: do_not_save argument was removed.

Deprecated since version 0.3.7: include_relationships argument was removed.

get_fixtures(**args*, ***kwargs*)
Get fixtures from iterable.

Parameters **fixture_keys** (*iterable*) –

Return type list of instantiated but unsaved fixtures

New in version 0.4.0: `builder` argument was added.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

init_fixtures()
Initialize the fixtures.

This function *must* be called before doing anything else.

install_all_fixtures(**args*, ***kwargs*)
Install all fixtures.

Return type list of `fixture_instance`

Deprecated since version 0.4.0: `do_not_save` argument was removed.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

install_fixture(**args*, ***kwargs*)
Install a fixture.

Parameters

- **fixture_key** (*str*) –
- **overrides** (*dict*) – override fields

Return type `fixture_instance`

Deprecated since version 0.4.0: `do_not_save` argument was removed. `attrs` argument renamed `overrides`.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

install_fixtures(**args*, ***kwargs*)
Install a list of fixtures.

Parameters **fixture_keys** (*str or list of strs*) – fixtures to be installed

Return type list of `fixture_instance`

Deprecated since version 0.4.0: `do_not_save` argument was removed.

Deprecated since version 0.3.7: `include_relationships` argument was removed.

uninstall_all_fixtures(**args*, ***kwargs*)
Uninstall all installed fixtures.

Return type `None`

Deprecated since version 0.4.0: `do_not_delete` argument was removed. This function does not return anything.

uninstall_fixture(**args*, ***kwargs*)
Uninstall a fixture.

Parameters **fixture_key** (*str*) –

Return type `None`

Deprecated since version 0.4.0: `do_not_delete` argument was removed. This function does not return anything.

uninstall_fixtures(**args*, ***kwargs*)
Uninstall a list of installed fixtures.

Parameters **fixture_keys** (*str or list of strs*) – fixtures to be uninstalled

Return type None

Deprecated since version 0.4.0: `do_not_delete` argument was removed. This function does not return anything.

1.7.3 Fixture

class charlatan.**Fixture**(*key*, *fixture_manager*, *model=None*, *fields=None*, *inherit_from=None*, *deep_inherit=False*, *post_creation=None*, *id_=None*, *models_package=''*, *depend_on=frozenset([])*)

Represent a fixture that can be installed.

static extract_rel_name(*name*)

Return the relationship and attr from an argument to !rel.

extract_relationships()

Return all dependencies.

Rtype generator

Yields `(depends_on, attr_name)`.

get_class()

Return class object for this instance.

get_instance(*path=None*, *overrides=None*, *builder=None*)

Instantiate the fixture using the model and return the instance.

Parameters

- **path** (*str*) – remaining path to return
- **overrides** (*dict*) – overriding fields
- **builder** (*func*) – function that is used to get the fixture

Deprecated since version 0.4.0: `fields` argument renamed `overrides`.

New in version 0.4.0: `builder` argument added.

Deprecated since version 0.3.7: `include_relationships` argument removed.

get_relationship(*name*)

Get a relationship and its attribute if necessary.

1.7.4 Utils

charlatan.utils.**copy_docstring_from**(*klass*)

Copy docstring from another class, using the same function name.

charlatan.utils.**datetime_to_epoch_in_ms**(*a_datetime*)

Return the epoch timestamp for the given datetime.

Parameters **a_datetime** (*datetime*) – The datetime to translate

Return type int

```
>>> a_datetime = datetime.datetime(2013, 11, 21, 1, 33, 11, 160611)
>>> datetime_to_epoch_timestamp(a_datetime)
1384997591.160611
>>> datetime_to_epoch_in_ms(a_datetime)
1384997591161
```

charlatan.utils.**datetime_to_epoch_timestamp**(*a_datetime*)

Return the epoch timestamp for the given datetime.

Parameters **a_datetime** (*datetime*) – The datetime to translate

Return type float

```
>>> a_datetime = datetime.datetime(2013, 11, 21, 1, 33, 11, 160611)
>>> datetime_to_epoch_timestamp(a_datetime)
1384997591.160611
```

charlatan.utils.**deep_update**(*source*, *overrides*)

Update a nested dictionary or similar mapping.

Modify `source` in place.

charlatan.utils.**extended_timedelta**(***kwargs*)

Return a `timedelta` object based on the arguments.

Parameters

- **years** (*integer*) –
- **months** (*integer*) –
- **days** (*integer*) –

Return type `timedelta` instance

Since `timedelta`'s largest unit are days, `timedelta` objects cannot be created with a number of months or years as an argument. This function lets you create `timedelta` objects based on a number of days, months and years.

```
>>> extended_timedelta(months=1)
datetime.timedelta(30)
>>> extended_timedelta(years=1)
datetime.timedelta(365)
>>> extended_timedelta(days=1, months=1, years=1)
datetime.timedelta(396)
>>> extended_timedelta(hours=1)
datetime.timedelta(0, 3600)
```

charlatan.utils.**get_timedelta**(*delta*)

Return timedelta from string.

Parameters **delta** (*str*) –

Return type `datetime.timedelta` instance

```
>>> get_timedelta("")
datetime.timedelta(0)
>>> get_timedelta("+1h")
datetime.timedelta(0, 3600)
>>> get_timedelta("+10h")
datetime.timedelta(0, 36000)
>>> get_timedelta("-10d")
datetime.timedelta(-10)
>>> get_timedelta("+1m")
datetime.timedelta(30)
>>> get_timedelta("-1y")
datetime.timedelta(-365)
>>> get_timedelta("+10d2h")
datetime.timedelta(10, 7200)
```

```
    >>> get_timedelta("-10d2h")
    datetime.timedelta(-11, 79200)
    >>> get_timedelta("-21y2m1d24h")
    datetime.timedelta(-7727)
    >>> get_timedelta("+5M")
    datetime.timedelta(0, 300)
```

charlatan.utils.**is_sqlalchemy_model**(*instance*)
Return True if instance is an SQLAlchemy model instance.

charlatan.utils.**richgetter**(*obj*, *path*)
Return a attrgetter + item getter.

charlatan.utils.**safe_iteritems**(*items*)
Safely iterate over a dict or a list.

1.8 Contributing

Install the requirements:

```
$ make bootstrap
```

Run the tests:

```
$ make test
```

1.9 Changelog for Charlatan

1.9.1 0.4.6 (2015-09-22)

- Add support for `epoch_now_in_ms` (thanks to @chunyan)

1.9.2 0.4.5 (2015-05-29)

- Add `deep_inherit` to allow nested inheritance of fields.

1.9.3 0.4.4 (2015-05-28)

- Added `!now_naive` YAML file command to return naive datetime.

1.9.4 0.4.3 (2015-05-26)

- Fixed anonymous list objects name resolution (thanks to @jvrsantacruz)

1.9.5 0.4.2 (2015-05-19)

- **Breaking change**: the `!now` YAML command now returns timezone-aware datetime by default. You can change that behavior by changing *charlatan.file_format.TIMEZONE_AWARE*.

- Fixed bug where uninstalling a sqlalchemy fixture would not commit the delete to the session.
- Fixed bug where dict fixtures could not reference fields from other collections of dicts.

1.9.6 0.4.1 (2015-02-26)

- Fixed bug where `!rel a_database_model.id`, where `id` is a primary key generated by the database, would be `None` because of how fixtures are cached.
- Removed `as_list` and `as_dict` feature. It was unnecessarily complex and would not play well with caching fixtures.

1.9.7 0.4.0 (2015-02-18)

- **Breaking change**: `get_builder` and `delete_builder` arguments were added to *charlatan.FixturesManager*.
- **Breaking change**: `delete_instance`, `save_instance` methods were deleted in favor of using builders (see below).
- **Breaking change**: `fields` argument on `charlatan.fixture.Fixture` and fixtures collection class has been renamed `overrides` for consistency reasons.
- **Breaking change**: `attrs` argument on *charlatan.FixturesManager* been renamed `overrides` for consistency reasons.
- **Breaking change**: deleting fixtures will not return anything. It used to return the fixture or list of fixtures that were successfully deleted. It has been removed to apply the command query separation pattern. There are other ways to check which fixtures are installed, and hooks or builders can be used to customize deletion.
- **Breaking change**: `do_not_save` and `do_not_delete` arguments have been removed from all functions, in favor of using builders.
- The notion of *charlatan.builder.Builder* was added. This allows customizing how fixtures are instantiated and installed. A `builder` argument has been added to most method dealing with getting, installing or deleting fixtures. Sane defaults have been added in most places.
- Improve documentation about using pytest with charlatan.
- Fix bug preventing being able to load multiple fixtures file.

1.9.8 0.3.12 (2015-01-14)

- Do not install the class' `fixtures` variable on *charlatan.FixturesManagerMixin* initialization. This can lead to bad pattern where a huge list of fixtures is installed for each test, even though each test uses only a few. Also, it's safer to be explicit about this behavior and let the user have this automatic installation. Note that you can easily reimplement this behavior by subclassing or installing those in the class `setUp` method.

1.9.9 0.3.11 (2015-01-06)

- Fix getting relationships with fields that are nested more than one level

1.9.10 0.3.10 (2014-12-31)

- Get `utcnow` at fixture instantiation time, to allow using `freezegun` intuitively

1.9.11 0.3.9 (2014-11-13)

- Fix saving collection of fixtures to database (thanks to @joegilley)

1.9.12 0.3.8 (2014-08-19)

- Support loading of globbed filenames

1.9.13 0.3.7 (2014-07-07)

- Support loading of multiple fixtures files
- Remove include_relationships option in instance creation

1.9.14 0.3.6 (2014-06-02)

- Update PYYaml

1.9.15 0.3.5 (2014-06-02)

- Support loading all strings as unicode

1.9.16 0.3.4 (2014-01-21)

- Fix getting attribute from relationships

1.9.17 0.3.3 (2014-01-18)

- Add support for Python 3

1.9.18 0.3.2 (2014-01-16)

- Add ability to uninstall fixtures (thanks to @JordanB)

1.9.19 0.3.1 (2014-01-10)

- Numerous tests added, a lot of cleanup.
- Clarification in documentation.
- Remove `load`, `set_hook` and `install_all_fixtures` shortcuts from charlatan package.
- Remove `FIXTURES_MANAGER` singleton. Remove `charlatan.fixtures_manager` shortcut.
- Remove `db_session` argument to `FixturesManager.load`.
- Add `db_session` argument to `FixturesManager` constructor.
- Remove `charlatan.fixtures_manager.FixturesMixin`. Replaced by `charlatan.testcase.FixturesManagerMixin`.

- `FixturesManagerMixin` now exposes pretty much the same method as `FixturesManager`.
- `FixturesManagerMixin`'s `use_fixtures_manager` was renamed `init_fixtures`.

1.9.20 0.2.9 (2013-11-20)

- Add `!epoch_now` for Unix timestamps (thanks to @erikformella)

1.9.21 0.2.8 (2013-11-12)

- Add ability to point to a list fixture (thanks to @erikformella)

1.9.22 0.2.7 (2013-10-24)

- Add ability to define dependencies outside of fields through the *depend_on* key in the yaml file (thanks to @Roguelazer)

1.9.23 0.2.6 (2013-09-06)

- Fix regression that broke API. install_fixture started returning the fixture as well as its name. (thanks to @erikformella)

1.9.24 0.2.5 (2013-09-06)

- Allow relationships to be used in dicts and lists. (thanks to @erikformella)
- Allow for seconds and minutes in relative timestamps (thanks to @kmnovak)

1.9.25 0.2.4 (2013-08-08)

- Empty models are allowed so that dict ands lists can be used as fixtures.
- Fixtures can now inherits from other fixtures.

1.9.26 0.2.3 (2013-06-28)

- Added ability to link to a relationship's attribute in YAML file.
- Added ability to use `!rel` in `post_creation`.

1.9.27 0.1.2 (2013-04-01)

- Started tracking changes

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

C

Symbols

B

C

D

E

F

G

I

K

L

R

S

U