MiniDataAPI Spec

The `MiniDataAPI` is a persistence API specification that designed to be small and relatively easy to implement across a wide range of datastores. While early implementations have been SQL-based, the specification can be quickly implemented in key/value stores, document databases, and more.

Work in Progress

The MiniData API spec is a work in progress, subject to change. While the majority of design is complete, expect there could be breaking changes.

Why?

The MiniDataAPI specification allows us to use the same API for many different database engines. Any application using the MiniDataAPI spec for interacting with its database requires no modification beyond import and configuration changes to switch database engines. For example, to convert an application from Fastlite running SQLite to FastSQL running PostgreSQL, should require only changing these two lines:

FastLite version

```
from fastlite import *
db = Database('test.db')
```

FastSQL version

```
from fastsql import *
db = Database('postgres:...')
```

As both libraries adhere to the MiniDataAPI specification, the rest of the code in the application should remain the same. The advantage of the MiniDataAPI spec is that it allows people to use whatever datastores they have access to or prefer.

Note

Switching databases won't migrate any existing data between databases.

Easy to learn, quick to implement

The MiniDataAPI specification is designed to be easy-to-learn and quick to implement. It focuses on straightforward Create, Read, Update, and Delete (CRUD) operations.

MiniDataAPI databases aren't limited to just row-based systems. In fact, the specification is closer in design to a key/value store than a set of records. What's exciting about this is we can write implementations for tools like Python dict stored as JSON, Redis, and even the venerable ZODB.

Limitations of the MiniDataAPI Specification

> "Mini refers to the lightweightness of specification, not the data."

> – Jeremy Howard

The advantages of the MiniDataAPI come at a cost. The MiniDataAPI specification focuses a very small set of features compared to what can be found in full-fledged ORMs and query languages. It intentionally avoids nuances or sophisticated features.

This means the specification does not include joins or formal foreign keys. Complex data stored over multiple tables that require joins isn't handled well. For this kind of scenario it's probably for the best to use more sophisticated ORMs or even direct database queries.

Summary of the MiniDataAPI Design

- Easy-to-learn
- Relative quick to implement for new database engines
- An API for CRUD operations
- For many different types of databases including row- and key/value-based designs
- Intentionally small in terms of features: no joins, no foreign keys, no database specific features
- Best for simpler designs, complex architectures will need more sophisticated tools.

Connect/construct the database

We connect or construct the database by passing in a string connecting to the database endpoint or a filepath representing the database's location. While this example is for SQLite running in memory, other databases such as PostgreSQL, Redis, MongoDB, might instead use a URI pointing at the database's filepath or endpoint. The method of connecting to a DB is *not* part of this API, but part of the underlying library. For instance, for fastlite:

```
db = database(':memory:')
```

Here's a complete list of the available methods in the API, all documented below (assuming `db` is a database and `t` is a table):

- `db.create`
- `t.insert`
- `t.delete`
- `t.update`
- `t[key]`
- `t(...)`
- `t.xtra`

Tables

For the sake of expediency, this document uses a SQL example. However, tables can represent anything, not just the fundamental construct of a SQL databases. They might represent keys within a key/value structure or files on a hard-drive.

Creating tables

We use a `create()` method attached to `Database` object (`db` in our example) to create the tables.

```python
class User: name:str; email: str; year_started:int
users = db.create(User, pk='name')
users
```

```
<Table user (name, email, year_started)>
```

```python
class User: name:str; email: str; year_started:int
users = db.create(User, pk='name')
users
```

```
<Table user (name, email, year_started)>
```

If no `pk` is provided, `id` is assumed to be the primary key. Regardless of whether you mark a class as a dataclass or not, it will be turned into one – specifically into a `flexiclass`.

```python
@dataclass
class Todo: id: int; title: str; detail: str; status: str; name: str
todos = db.create(Todo)
todos
```

```
<Table todo (id, title, detail, status, name)>
```

Compound primary keys

The MiniData API spec supports compound primary keys, where more than one column is used to identify records. We'll also use this example to demonstrate creating a table using a dict of keyword arguments.

```python
class Publication: authors: str; year: int; title: str
publications = db.create(Publication, pk=('authors', 'year'))
```

Transforming tables

Depending on the database type, this method can include transforms - the ability to modify the tables. Let's go ahead and add a password field for our table called `pwd`.

```python
class User: name:str; email: str; year_started:int; pwd:str
users = db.create(User, pk='name', transform=True)
users
```

```
<Table user (name, email, year_started, pwd)>
```

Manipulating data

The specification is designed to provide as straightforward CRUD API (Create, Read, Update, and Delete) as possible. Additional features like joins are out of scope.

.insert()

Add a new record to the database. We want to support as many types as possible, for now we have tests for Python classes, dataclasses, and dicts. Returns an instance of the new record.

Here's how to add a record using a Python class:

```
users.insert(User(name='Braden', email='b@example.com', year_started=2018))
```

```
User(name='Braden', email='b@example.com', year_started=2018, pwd=None)
```

We can also use keyword arguments directly:

```
users.insert(name='Alma', email='a@example.com', year_started=2019)
```

```
User(name='Alma', email='a@example.com', year_started=2019, pwd=None)
```

And now Charlie gets added via a Python dict.

```
users.insert({'name': 'Charlie', 'email': 'c@example.com', 'year_started': 2018})
```

```
User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)
```

And now TODOs. Note that the inserted row is returned:

```
todos.insert(Todo(title='Write MiniDataAPI spec', status='open', name='Braden'))
todos.insert(title='Implement SSE in FastHTML', status='open', name='Alma')
todo = todos.insert(dict(title='Finish development of FastHTML', status='closed', name
todo
```

```
Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')
```

Let's do the same with the `Publications` table.

```
publications.insert(Publication(authors='Alma', year=2019, title='FastHTML'))
publications.insert(authors='Alma', year=2030, title='FastHTML and beyond')
publication= publications.insert((dict(authors='Alma', year=2035, title='FastHTML, the
publication
```

```
Publication(authors='Alma', year=2035, title='FastHTML, the early years')
```

Square bracket search []

Get a single record by entering a primary key into a table object within square brackets. Let's see if we can find Alma.

```
user = users['Alma']
user
```

```
User(name='Alma', email='a@example.com', year_started=2019, pwd=None)
```

If no record is found, a `NotFoundError` error is raised. Here we look for David, who hasn't yet been added to our users table.

```
try: users['David']
except NotFoundError: print(f'User not found')
```

```
User not found
```

Here's a demonstration of a ticket search, demonstrating how this works with non-string primary keys.

```
todos[1]
```

```
Todo(id=1, title='Write MiniDataAPI spec', detail=None, status='open',
name='Braden')
```

Compound primary keys can be supplied in lists or tuples, in the order they were defined. In this case it is the `authors` and `year` columns.

Here's a query by compound primary key done with a `list`:

```
publications[['Alma', 2019]]
```

```
Publication(authors='Alma', year=2019, title='FastHTML')
```

Here's the same query done directly with index args.

```
publications['Alma', 2030]
```

```
Publication(authors='Alma', year=2030, title='FastHTML and beyond')
```

Parentheses search ()

Get zero to many records by entering values with parentheses searches. If nothing is in the parentheses, then everything is returned.

```
users()
```

```
[User(name='Braden', email='b@example.com', year_started=2018, pwd=None),
 User(name='Alma', email='a@example.com', year_started=2019, pwd=None),
 User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)]
```

We can order the results.

```
users(order_by='name')
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None),
 User(name='Braden', email='b@example.com', year_started=2018, pwd=None),
 User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)]
```

We can filter on the results:

```
users(where="name='Alma'")
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None)]
```

Generally you probably want to use placeholders, to avoid SQL injection attacks:

```
users("name=?", ('Alma',))
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None)]
```

We can limit results with the `limit` keyword:

```
users(limit=1)
```

```
[User(name='Braden', email='b@example.com', year_started=2018, pwd=None)]
```

If we're using the `limit` keyword, we can also use the `offset` keyword to start the query later.

```
users(limit=5, offset=1)
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None),
 User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)]
```

.update()

Update an existing record of the database. Must accept Python dict, dataclasses, and standard classes. Uses the primary key for identifying the record to be changed. Returns an instance of the updated record.

Here's with a normal Python class:

```
user
```

```
User(name='Alma', email='a@example.com', year_started=2019, pwd=None)
```

```
user.year_started = 2099
users.update(user)
```

```
User(name='Alma', email='a@example.com', year_started=2099, pwd=None)
```

Or use a dict:

```
users.update(dict(name='Alma', year_started=2199, email='a@example.com'))
```

```
User(name='Alma', email='a@example.com', year_started=2199, pwd=None)
```

Or use kwargs:

```
users.update(name='Alma', year_started=2149)
```

```
User(name='Alma', email='a@example.com', year_started=2149, pwd=None)
```

If the primary key doesn't match a record, raise a `NotFoundError`.

John hasn't started with us yet so doesn't get the chance yet to travel in time.

```
try: users.update(User(name='John', year_started=2024, email='j@example.com'))
except NotFoundError: print('User not found')
```

```
User not found
```

.delete()

Delete a record of the database. Uses the primary key for identifying the record to be removed. Returns a table object.

Charlie decides to not travel in time. He exits our little group.

```
users.delete('Charlie')
```

```
<Table user (name, email, year_started, pwd)>
```

If the primary key value can't be found, raises a `NotFoundError`.

```
try: users.delete('Charlies')
except NotFoundError: print('User not found')
```

```
User not found
```

In John's case, he isn't time travelling with us yet so can't be removed.

```
try: users.delete('John')
except NotFoundError: print('User not found')
```

```
User not found
```

Deleting records with compound primary keys requires providing the entire key.

```
publications.delete(['Alma' , 2035])
```

```
<Table publication (authors, year, title)>
```

`in` keyword

Are `Alma` and `John` contained `in` the Users table? Or, to be technically precise, is the item with the specified primary key value `in` this table?

```
'Alma' in users, 'John' in users
```

```
(True, False)
```

Also works with compound primary keys, as shown below. You'll note that the operation can be done with either a `list` or `tuple`.

```
['Alma', 2019] in  publications
```

```
True
```

And now for a `False` result, where John has no publications.

```
('John', 1967) in publications
```

```
False
```

.xtra()

If we set fields within the `.xtra` function to a particular value, then indexing is also filtered by those. This applies to every database method except for record creation. This makes it easier to limit users (or other objects) access to only things for which they have permission.

For example, if we query all our records below without setting values v ia the `.xtra` function, we can see todos for everyone. Pay special attention to the `id` values of all three records, as we are about to filter most of them away.

```
todos()
```

```
[Todo(id=1, title='Write MiniDataAPI spec', detail=None, status='open',
name='Braden'),
 Todo(id=2, title='Implement SSE in FastHTML', detail=None, status='open',
name='Alma'),
 Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')]
```

Let's use `.xtra` to constrain results just to Charlie. We set the `name` field in Todos, but it could be any field defined for this table.

```
todos.xtra(name='Charlie')
```

We've now set a field to a value with `.xtra`, if we loop over all the records again, only those assigned to records with a `name` of `Charlie` will be displayed.

```
todos()
```

```
[Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')]
```

The `in` keyword is also affected. Only records with a `name` of Charlie will evaluate to be `True`. Let's demonstrate by testing it with a Charlie record:

```
ct = todos[3]
ct
```

```
Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')
```

Charlie's record has an ID of 3. Here we demonstrate that Charlie's TODO can be found in the list of todos:

```
ct.id in todos
```

```
True
```

If we try `in` with the other IDs the query fails because the filtering is now set to just records with a name of Charlie.

```
1 in todos, 2 in todos
```

```
(False, False)
```

```
try: todos[2]
except NotFoundError: print('Record not found')
```

```
Record not found
```

We are also constrained by what records we can update. In the following example we try to update a TODO not named 'Charlie'. Because the name is wrong, the `.update` function will raise a `NotFoundError`.

```
try: todos.update(Todo(id=1, title='Finish MiniDataAPI Spec', status='closed', name='B
except NotFoundError as e: print('Record not updated')
```

```
Record not updated
```

Unlike poor Braden, Charlie isn't filtered out. Let's update his TODO.

```
todos.update(Todo(id=3, title='Finish development of FastHTML', detail=None, status='c
```

```
Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')
```

Finally, once constrained by `.xtra`, only records with Charlie as the name can be deleted.

```
try: todos.delete(1)
except NotFoundError as e: print('Record not updated')
```

```
Record not updated
```

Charlie's TODO was to finish development of FastHTML. While the framework will stabilize, like any good project it will see new features added and the odd bug corrected for many years to come. Therefore, Charlie's TODO is nonsensical. Let's delete it.

```
todos.delete(ct.id)
```

```
<Table todo (id, title, detail, status, name)>
```

When a TODO is inserted, the `xtra` fields are automatically set. This ensures that we don't accidentally, for instance, insert items for others users. Note that here we don't set the `name` field, but it's still included in the resultant row:

```
ct = todos.insert(Todo(title='Rewrite personal site in FastHTML', status='open'))
ct
```

```
Todo(id=3, title='Rewrite personal site in FastHTML', detail=None, status='open',
name='Charlie')
```

If we try to change the username to someone else, the change is ignored, due to `xtra`:

```
ct.name = 'Braden'
todos.update(ct)
```

```
Todo(id=3, title='Rewrite personal site in FastHTML', detail=None, status='open',
name='Charlie')
```

SQL-first design

```
users = None
User = None
```

```
users = db.t.user
users
```

```
<Table user (name, email, year_started, pwd)>
```

(This section needs to be documented properly.)

From the table objects we can extract a Dataclass version of our tables. Usually this is given an singular uppercase version of our table name, which in this case is `User`.

```
User = users.dataclass()
```

```
User(name='Braden', email='b@example.com', year_started=2018)
```

```
User(name='Braden', email='b@example.com', year_started=2018, pwd=UNSET)
```

Implementations

Implementing MiniDataAPI for a new datastore

For creating new implementations, the code examples in this specification are the test case for the API. New implementations should pass the tests in order to be compliant with the specification.

Implementations

- fastlite - The original implementation, only for Sqlite
- fastsql - An SQL database agnostic implementation based on the excellent SQLAlchemy library.

Report an issue