DStore Documentation

Release 0.1.0a1

Mark Pittaway

February 13, 2017

Contents:

DStore (DataStore) is a Python Object Relational Mapper (ORM) that allows easy description of data models.

The source code can be found on GitHub at https://github.com/MarkLark/dstore

The Python Package can be found on PyPi at https://pypi.python.org/pypi/DStore

CHAPTER 1

Minimal Example

```
from dstore import MemoryStore, Model, var, mod

class Car( Model ):
    _namespace = "cars.make"
    _vars = [
        var.RowID,
        var.String( "manufacturer", 32, mods = [ mod.NotNull() ] ),
        var.String( "make", 32, mods = [ mod.NotNull() ] ),
        var.Number( "year", mods = [ mod.NotNull(), mod.Min( 1950 ), mod.Max( 2017 ) ] ),
    ]

# Create the MemoryStore instance, and add Models to it
store = MemoryStore( [ Car ] )
store.init_app()
store.connect()
store.create_all()

# Create a new Car, then retrieve it using filter and all
Car( manufacturer = "Holden", make = "Commodore", year = 2010 ).add()
holdens = Car.filter( manufacturer = "Holden" )
cars = Car.all()

# Destroy all instances and shut down the application
store.destroy_all()
store.disconnect()
store.destroy_app()
```

1.1 Introduction To DStore

DStore (DataStore) is a Python Object Relational Mapper (ORM) that allows easy description of data models.

1.1.1 Installing

From PyPi

DStore is available from the PyPi repository at DStore.

This means that all you have to do to install DStore is run the following in a console:

```
$ pip install dstore
```

From Source

DStore can also be installed from source by downloading from GitHub and running setup.py.

```
$ wget https://github.com/MarkLark/dstore/archive/master.tar.gz
$ tar xvf master.tar.gz
$ cd dstore-master
$ python setup.py install
```

1.1.2 Requirements

DStore does not rely on any other Python Packages.

It has also been thoroughly tested to work on the following Python Versions:

- 2.7
- 3.3
- 3.4
- 3.5
- 3.6

1.1.3 Minimal Example

```
from dstore import MemoryStore, Model, var, mod

class Car( Model ):
    _namespace = "cars.make"
    _vars = [
        var.RowID,
        var.String( "manufacturer", 32, mods = [ mod.NotNull() ] ),
        var.String( "make", 32, mods = [ mod.NotNull() ] ),
        var.Number( "year", mods = [ mod.NotNull(), mod.Min( 1950 ), mod.Max( 2017 ) ] ),
    ]

# Create the MemoryStore instance, and add Models to it
store = MemoryStore( [ Car ] )
store.init_app()
store.connect()
store.create_all()

# Create a new Car, then retrieve it using filter and all
Car( manufacturer = "Holden", make = "Commodore", year = 2010 ).add()
holdens = Car.filter( manufacturer = "Holden" )
cars = Car.all()

# Destroy all instances and shut down the application
store.destroy_all()
store.disconnect()
store.destroy_app()
```

1.1.4 How It Works

DStore works by defining the models that you wish to use. It then provides a layer for storing the model instances.

1.2 Models

Models are used to describe the data types you wish to use.

1.2.1 Base Model

The base model

1.2.2 Variables

Variables within a Model

class dstore.var.**Variable**(*name*, *default=None*, *mods=None*)

The base class used for creating Variable types This also provides a validate method which iterates over the modifiers and runs their validate method on this variable instance.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

validate(*instance*)

Validate the value of an instance of this Variable type (this is used by the dstore.Model.validate method)

Parameters **instance** – The instance of this variable type

Returns If not a valid value, an exception of dstore.Error.ValidationError will be raised

RowID

RowID denotes an instance ID, and is not a Class but an instance of dstore.var.Number:

```
from dstore import var, mod
RowID = Number( "id", mods = [ mod.AutoIncrement(), mod.PrimaryKey(), mod.Unique() ])
```

In terms of a MySQL it equates to:

```
id INT NOT NULL AUTO_INCREMENT,
UNIQUE (id),
PRIMARY KEY (id)
```

Usage

```
from dstore import Model, var

class Car( Model ):
    _namespace = "cars.make"
    _vars = [
```

```
        var.RowID
    ]
```

Number

class dstore.var.**Number**(*name*, *default=None*, *mods=None*)
This variable type is used to store an integer value.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

Usage:

```
from dstore import var, mod
var.Number( "year", 1950, mods = [ mod.NotNull() ] )
```

Boolean

class dstore.var.**Boolean**(*name*, *default=None*, *mods=None*)
This variable is used to store a boolean value.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

Usage:

```
from dstore import var
var.Boolean( "is_available", False )
```

Character

class dstore.var.**Character**(*name*, *length*, *default=None*, *mods=None*)
This variable is used to store a string of static length.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **length** – The length of the string
- **default** – The default value to give this variable
- **mods** – A list of modifiers. The mod dstore.mod.Length(length) is always added to this list

Usage:

```
from dstore import var
var.Character( "is_available", 4 )
```

Binary

class dstore.var.**Binary**(*name*, *length*, *default=None*, *mods=None*)
This variable is used to store binary data.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **length** – The length of the data
- **default** – The default value to give this variable
- **mods** – A list of modifiers. The mod dstore.mod.Length(length) is always added to this list

Usage:

```
from dstore import var
var.Binary( "data", 32 )
```

String

class dstore.var.**String**(*name*, *length*, *default=None*, *mods=None*)
This variable is used to store a string of variable length.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **length** – The length of the string
- **default** – The default value to give this variable
- **mods** – A list of modifiers. The mod dstore.mod.Length(length) is always added to this list

Usage:

```
from dstore import var
var.String( "username", 32 )
```

Text

class dstore.var.**Text**(*name*, *default=None*, *mods=None*)
This variable is used to store text.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

Usage:

```
from dstore import var
var.Text( "description" )
```

Float

class dstore.var.**Float**(*name*, *default=None*, *mods=None*)
This variable is used to store a floating point integer.

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

Usage:

```
from dstore import var
var.Float( "price" )
```

Enum

class dstore.var.**Enum**(*name*, *values*, *default=None*, *mods=None*)
This variable is used to store a an enum.

Parameters

- **name** (*str*) – The name of the Variable instance inside a Model Class
- **values** (*list[str]*) – The list of available choices
- **default** (*str or None*) – The default value to give this variable
- **mods** (*list[dstore.mod.Mod]*) – A list of modifiers. The mod dstore.mod.InEnum(values) is always added to this list

Usage:

```
from dstore import var, mod
var.Enum( "car_type", [ "truck", "bus", "ute", "car", "motorbike" ], mods = [ mod.NotNull() ] )

# or

var.Enum(
    name    = "car_type",
    values  = [ "truck", "bus", "ute", "car", "motorbike" ],
    default = "car",
    mods    = [ mod.NotNull() ]
)
```

ForeignKey

class dstore.var.**ForeignKey**(*namespace*)
This variable is used to link the Model Instance to another Model Instance

Parameters **namespace** – The namespace of the Model Class this variable references

This variable type is a subclass of dstore.var.Number, with the following mod:

•dstore.mod.ForeignKey

The name given to this variable is the name of the Model Class being referenced with a suffix of '_id"

For example:

```
from dstore import var
var.ForeignKey( "cars.make" )
```

Is the same as:

```
from dstore import var
var.Number( "cars_make_id", mods = [ dstore.mod.ForeignKey( "cars.make" ) ] )
```

Date

class dstore.var.**Date**(*name*, *default=None*, *mods=None*)

This variable is used to store a date value (without a timezone offset).

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

Usage:

```
from dstore import var
var.Date( "build_date" )
```

Time

class dstore.var.**Time**(*name*, *default=None*, *mods=None*)

This variable is used to store a time value (without a timezone offset).

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

Usage:

```
from dstore import var
var.Time( "build_time" )
```

DateTime

class dstore.var.**DateTime**(*name*, *default=None*, *mods=None*)

This variable is used to store a date and time value (without a timezone offset).

Parameters

- **name** – The name of the Variable instance inside a Model Class
- **default** – The default value to give this variable
- **mods** – A list of modifiers, i.e. dstore.mod.NotNull()

Usage:

```
from dstore import var
var.DateTime( "built_on" )
```

1.2.3 Modifiers

class dstore.mod.**Mod**
The base class used for creating a Modifier.

A Modifier is a way to validate Model instance Variables upon add and/or update.

validate(*instance*, *var*, *val*)

Parameters

- **instance** – The Model instance
- **var** – The Model Variable Class
- **val** – The Model instance variable

Returns If not a valid value, an exception of dstore.Error.ValidationError will be raised

PrimaryKey

class dstore.mod.**PrimaryKey**
This modifier specifies that the variable is used as a Model ID.

This is automatically used by dstore.val.RowID.

NotNull

class dstore.mod.**NotNull**
This modifier ensures that the value is not None.

validate(*instance*, *var*, *val*)

Raises Raises dstore.mod.NotNull.NotNull_Error if val is None

AutoIncrement

class dstore.mod.**AutoIncrement**
This modifier specifies that the variable is automatically incremented.

This is automatically used by dstore.val.RowID.

Unique

class dstore.mod.**Unique**
This modifier ensures that the value is unique to all other instances.

ForeignKey

class dstore.mod.**ForeignKey**(*namespace*)
This modifier specifies that the value references another Model instance.

This is automatically used by dstore.val.ForeignKey.

Min

class dstore.mod.**Min**(*value*)
This modifier ensures that the value is greater than a specific number.

Parameters **value** – The minimum value allowed

validate(*instance*, *var*, *val*)

Raises Raises dstore.mod.Min.Min_Error if val is less than the allowed number.

Max

class dstore.mod.**Max**(*value*)
This modifier ensures that the value is less than a specific number.

Parameters **value** – The maximum value allowed

validate(*instance*, *var*, *val*)

Raises Raises dstore.mod.Max.Max_Error if val is greater than the allowed number.

Length

class dstore.mod.**Length**(*value*)
This modifier ensures that the value does not exceed the given length.

Parameters **value** – The maximum length of the value allowed

validate(*instance*, *var*, *val*)

Raises Raises dstore.mod.Length.Length_Error if the length of the value exceed the allowed size.

InEnum

class dstore.mod.**InEnum**(*values*)
This modifier ensures that the value exists in the enumerated list.

This is automatically used by `dstore.val.Enum`.

Parameters **value** – A list of the allowed values

validate(*instance*, *var*, *val*)

Raises Raises dstore.mod.InEnum.InEnum_Error if the value does not exist in the enumerated list.

1.3 Storage

DStore provides an abstraction layer to store the model instances.

1.3.1 Memory

Introduction

The default storage in DStore is the MemoryStore.

This simply stores the model instances in an array in memory.

Keep in mind that this storage type is runtime only, meaning that the data is wiped when the application closes.

Usage

```
from dstore import MemoryStore
store = MemoryStore( models )
```

1.3.2 MySQL

Introduction

This storage type allows model instances to be stored in a MySQL DataBase.

In order to use this storage type, you need to install the Python Package for it.

Installing

You have two choices to install dstore-mysql, using pip or from source.

From PyPi

DStore is available from the PyPi repository at DStore.

This means that all you have to do to install PyMan is run the following in a console:

```
$ pip install dstore-mysql
```

From Source

DStore can also be installed from source by downloading from GitHub and running setup.py.

```
$ wget https://github.com/MarkLark/dstore-mysql/archive/master.tar.gz
$ tar xvf master.tar.gz
$ cd dstore-mysql-master
$ python setup.py install
```

Requirements

DStore-MySQL requires the Python package MySQL-python.

Usage

```
from dstore_mysql import MySQLStore
store = MySQLStore( models )
```

1.3.3 MongoDB

Introduction

This storage type allows model instances to be stored in a Mongo DataBase.

In order to use this storage type, you need to install the Python Package for it.

Installing

You have two choices to install dstore-mongo, using pip or from source.

From PyPi

DStore-Mongo package is available from the PyPi repository.

This means that all you have to do to install PyMan is run the following in a console:

```
$ pip install dstore-mongo
```

From Source

DStore-Mongo can also be installed from source by downloading from GitHub and running setup.py.

```
$ wget https://github.com/MarkLark/dstore-mongo/archive/master.tar.gz
$ tar xvf master.tar.gz
$ cd dstore-mongo-master
$ python setup.py install
```

Requirements

DStore-Mongo requires the Python package PyMongo.

Usage

```
from dstore_mongo import MongoStore
store = MongoStore( models )
```

1.4 CRUD

The Data Store provides functionality to Create, Read, Update, Delete and Filter the Model instances.

1.4.1 Create

Creating Model Instances

1.4.2 Read

Reading Model Instances

1.4.3 Update

Updating Model Instances

1.4.4 Delete

Deleting Model Instances

1.4.5 Filter

Filtering Model Instances

1.5 Security - DStore-ACL

1.5.1 Introduction

DStore-ACL is a Security Layer for DStore. Simply by provide the ACL when describing your Models, and DStore-ACL will do the rest.

1.5.2 Installing

You have two choices to install dstore-acl, using pip or from source.

From PyPi

DStore is available from the PyPi repository at DStore.

This means that all you have to do to install PyMan is run the following in a console:

```
$ pip install dstore-acl
```

From Source

DStore can also be installed from source by downloading from GitHub and running setup.py.

```
$ wget https://github.com/MarkLark/dstore-acl/archive/master.tar.gz
$ tar xvf master.tar.gz
$ cd dstore-acl-master
$ python setup.py install
```

1.6 Web API - Flask-DStore

1.6.1 Introduction

Flask-DStore is a Web API and Javascript Client.

The API routes, logic and client code is automatically generated for you.

1.6.2 Installing

You have two choices to install dstore-mysql, using pip or from source.

From PyPi

DStore is available from the PyPi repository at https://pypi.python.org/pypi/flask-dstore.

This means that all you have to do to install PyMan is run the following in a console:

```
$ pip install flask-dstore
```

From Source

DStore can also be installed from source by downloading from GitHub and running setup.py.

```
$ wget https://github.com/MarkLark/flask-dstore/archive/master.tar.gz
$ tar xvf master.tar.gz
$ cd flask-dstore-master
$ python setup.py install
```

1.6.3 Minimal Example

```
from flask import Flask
from dstore import MemoryStore, Model, var, mod
from flask_dstore import API

class Car( Model ):
    _namespace = "cars.make"
    _vars = [
        var.RowID,
        var.String( "manufacturer", 32, mods = [ mod.NotNull() ] ),
        var.String( "make", 32, mods = [ mod.NotNull() ] ),
        var.Number( "year", mods = [ mod.NotNull(), mod.Min( 1950 ), mod.Max( 2017 ) ] ),
```

```
    ]

# Create the app instances
app = Flask( __name__ )
store = MemoryStore( [ Car ] )
api = API( store, app )

# While inside the Flask app context, create all storage and add a car
with app.app_context():
    store.create_all()
    Car( manufacturer = "Holden", make = "Commodore", year = 2005 ).add()

# Run the Flask dev. server
app.run()

# Now destroy all data
with app.app_context():
    store.destroy_all()

store.destroy_app()
```

1.6.4 Javascript Client

Flask-DStore provides a javascript library that allows you to easily create, read, update and delete model instances within Javascript, for every Model type registered, a Factory class is created.

To include this library, add the following to your base html template

```
<script src="/dstore-client.js"></script>
<script src="/dstore-models.js"></script>
<script src="/dstore-view.js"></script>
```

DS.Factory

This Factory contains the following methods:

class DS.**Factory**()

load_all()
Load all Model instances from the server into browser memory

load(*id*)
Load a specific Model Instance from the server into the browser memory

Arguments

- **id** (*int*) – The instance ID to retrieve

get(*id*)
Get a Model Instance from browser memory, null if it doesn't exist

Arguments

- **id** (*int*) – The instance ID to retrieve

add(*args*)
Add a new Model Instance into browser memory.

You need to execute save on the returned object to save the instance to the server.

Arguments

- **args** – A dictionary of values to store in the Model Instance

DS.Model

The Model class is as follows:

class DS.**Model**()

save()
Save this Model Instance to the Server.

delete()
Delete the Model Instance from the server, and local browser memory.

Example Usage

You don't directly use the DS.Factory and DS.Model class' directly. Instead Flask-DStore generates instances of these class' for every Model type registered.

These are then stored under the 'ds' namespace, proceeded by the namespace of the Model types.

The following are examples of how to use this library with the following Model:

```
from dstore import MemoryStore, Model, var, mod

class Car( Model ):
    _namespace = "cars.make"
    _vars = [
        var.RowID,
        var.String( "manufacturer", 32, mods = [ mod.NotNull() ] ),
        var.String( "make", 32, mods = [ mod.NotNull() ] ),
        var.Number( "year", mods = [ mod.NotNull(), mod.Min( 1950 ), mod.Max( 2017 ) ] ),
    ]
```

Load All

```
ds.cars.make.load_all();
car = ds.cars.make.add({ manufacturer: "Holden", make: "Commodore", year: 2010 });
car.save();
```

Create

```
ds.cars.make.load_all();
car = ds.cars.make.add({ manufacturer: "Holden", make: "Commodore", year: 2010 });
car.save();
```

Update

```
ds.cars.make.load_all();
car = ds.cars.make.get(1);
car.year = 2011;
car.save();
```

Delete

```
ds.cars.make.load_all();
car = ds.cars.make.get(1);
car.delete();
```

1.7 Events

Table of Contents

1.7.1 Introduction

DStore makes the use of an Event Manager in the Store and Models themselves.

These events allow you to hook custom code before or after an action takes place.

In fact, this is exactly how DStore-ACL works to provide a security layer to DStore Models.

Listening To Events

Listening to an event is as easy as adding your method to the event in question.

For example, to listen to before_init_app on the store:

```
from dstore import MemoryStore

def before_init_app( event, store ):
    print( "Before init store %s" % store.name )

def after_init_app( event, store ):
    print( "After init store %s" % store.name )

store = MemoryStore()
store.events.before_init_app += before_init_app
store.events.after_init_app  += after_init_app

store.init_app()
```

1.7.2 Store Events

To listen to a store event, you add your method to the store's event object.

```
store.events.before_init_app += method_to_call
```

init_app

This event is fired before and after you execute store.init_app()

before_init_app(*event*, *store*)
after_init_app(*event*, *store*)

store
The Data Store that init_app is being run on

destroy_app

This event is fired before and after you execute destroy_app on a store

before_destroy_app(*event*, *store*)
after_destroy_app(*event*, *store*)

store
The Data Store that destroy_app is being run on

register_models

This event is fired before and after all models have been registered.

This happens automatically when init_app is run on the store

before_register_models(*event*, *store*)

after_register_models(*event*, *store*)

store
The Data Store that register_models is being run on

register_model

This event is fired when a single Model is being registered on the store.

before_register_models(*event*, *store*, *model*)
after_register_models(*event*, *store*, *model*)

store
The Data Store that register_models is being run on

model
The Model Class that is being registered

create_all

This event is fired before and after you execute create_all on a store

before_create_all(*event*, *store*)
after_create_all(*event*, *store*)

store
The Data Store that create_all is being run on

destroy_all

This event is fired before and after you execute destroy_all on a store

before_destroy_all(*event*, *store*)
after_destroy_all(*event*, *store*)

store
The Data Store that destroy_all is being run on

empty_all

This event is fired before and after you execute empty_all on a store

before_empty_all(*event*, *store*)
after_empty_all(*event*, *store*)

store
The Data Store that empty_all is being run on

connect

This event is fired before and after you execute connect on a store

before_connect(*event*, *store*)
after_connect(*event*, *store*)

store
The Data Store that connect is being run on

disconnect

This event is fired before and after you execute disconnect on a store

before_disconnect(*event*, *store*)
after_disconnect(*event*, *store*)

store
The Data Store that disconnect is being run on

add_bulk

This event is fired before and after you execute add_bulk on a store

before_add_bulk(*event*, *store*, *data*)
after_add_bulk(*event*, *store*, *data*, *instances*)

store
The Data Store that disconnect is being run on

data
The dictionary provided that is used to add Model instances to the Store

instances
The Model instances that were added to the Store

1.7.3 Model Events

To listen to a store event, you add your method to the store's event object.

```
from dstore import MemoryStore, Model, var, mod

class Car( Model ):
    _namespace = "cars.make"
    _vars = [
        var.RowID,
        var.String( "manufacturer", 32, mods = [ mod.NotNull() ] ),
        var.String( "make", 32, mods = [ mod.NotNull() ] ),
        var.Number( "year", mods = [ mod.NotNull(), mod.Min( 1950 ), mod.Max( 2017 ) ] ),
    ]

def car_before_add( event, model, instance ):
    print( "Attempting to add a new %s instance" % model._namespace )
```

```
Car.events.before_add += car_before_add
```

add

This event is fired before and after you attempt to add a new Model Instance

before_add(*event*, *model*, *instance*)
after_add(*event*, *model*, *instance*)

model
The Model Class that a new instance is being added to

instance
The instance that is attempting to be added to the Model Class storage

delete

This event is fired before and after you attempt to delete an existing Model Instance

before_delete(*event*, *model*, *instance*)
after_delete(*event*, *model*, *instance*)

model
The Model Class of the instance to be deleted

instance
The instance that is attempting to be deleted from the Model Class storage

update

This event is fired before and after you attempt to update an existing Model Instance

before_update(*event*, *model*, *instance*)
after_update(*event*, *model*, *instance*)

model
The Model Class of the instance to be updated

instance
The instance that is attempting to be updated

validate

This event is fired before and after you attempt to add or update a Model Instance (i.e. on validation)

before_validate(*event*, *model*, *instance*)
after_validate(*event*, *model*, *instance*)

model
The Model Class of the instance to be added or updated

instance
The instance that is attempting to be added or updated

all

This event is fired before and after you attempt to get all Model instances

before_all (*event*, *model*)

model
The Model Class of the instance to be added or updated

after_all (*event*, *model*, *instances*)

model
The Model Class of the instance to be added or updated

instances
The list of all instances

get

This event is fired before and after you attempt to get a Model instance

before_get (*event*, *model*, *row_id*)

model
The Model Class of the instance to be added or updated

row_id
The ID of the instance to retrieve

after_get (*event*, *model*, *instance*)

model
The Model Class of the instance to be added or updated

instance
The Model instance retrieved

empty

This event is fired before and after you attempt to delete all Model instances

before_empty (*event*, *model*)
after_empty (*event*, *model*)

model
The Model Class that is to be emptied

create

This event is fired before and after you attempt to create the storage for the Model instances

before_create(*event*, *model*)
after_create(*event*, *model*)

model
The Model Class that storage is to be created for

destroy

This event is fired before and after you attempt to destroy the storage for the Model instances

before_destroy(*event*, *model*)
after_destroy(*event*, *model*)

model
The Model Class that storage is to be destroyed for

filter

This event is fired before and after you attempt to get a filtered list of the Model instances

before_filter(*event*, *model*, *params*)

model
The Model Class to filter for instances

params
A dictionary of the parameters used to filter the list

after_filter(*event*, *model*, *instances*, *params*)

model
The Model Class to filter for instances

instances
The filtered list of Model instances

params
A dictionary of the parameters used to filter the list

Indices and tables

- genindex
- modindex
- search

d

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V