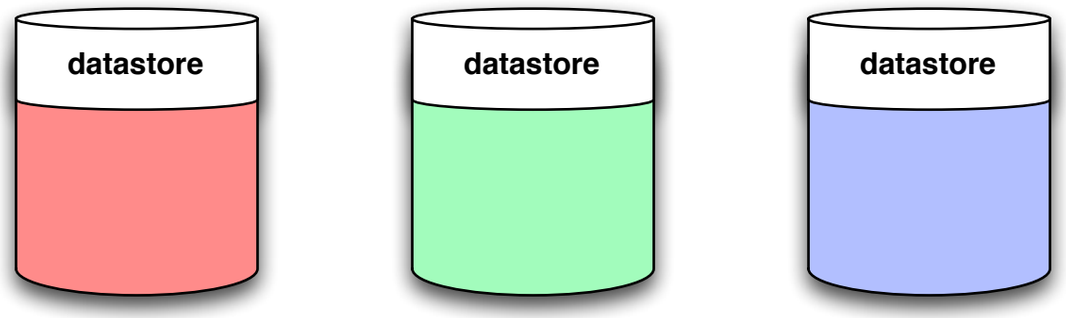
datastore

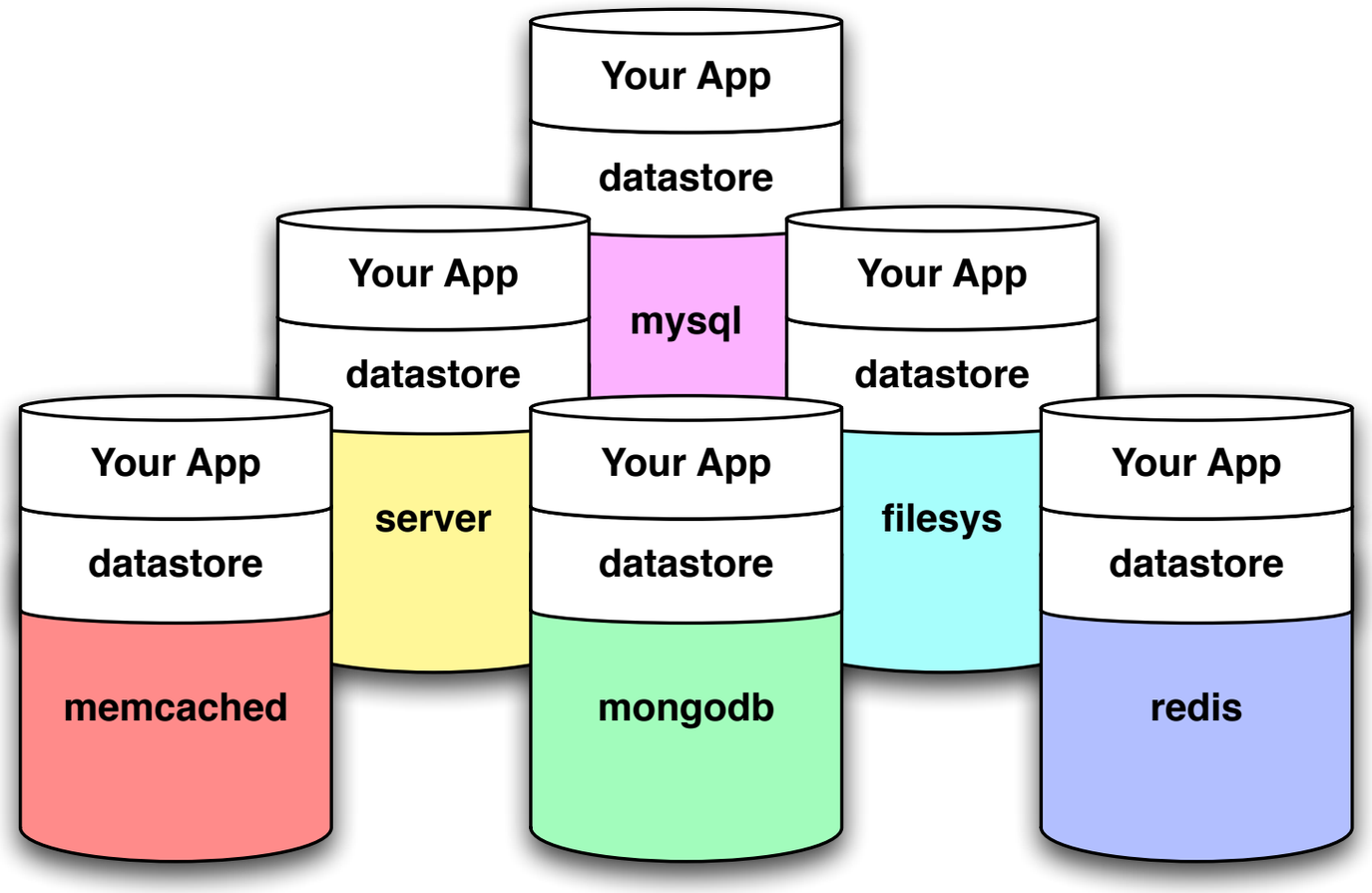
unified API for multiple data stores

https://github.com/jbenet/datastore



Juan Batiz-Benet
jbenet@cs.stanford.edu

datastore


Your App
datastore
Your App
datastore
mysql
Your App
datastore
Your App
datastore
server
Your App
datastore
filesys
Your App
datastore
memcached
mongodb
redis

datastore

- generic layer of abstraction for data store and database access
- **simple** api that enables application development in a datastore-agnostic way
- datastores can be swapped seamlessly without changing application code
- leverage different datastores with different strengths without committing the app to one datastore for its lifetime

hello world

```
>>> import datastore
>>> ds = datastore.basic.DictDatastore()
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

api

```
class Datastore(object):
  '''A Datastore represents storage for any key-value pair.

  Datastores are general enough to be backed by all kinds of different storage:
  in-memory caches, databases, a remote datastore, flat files on disk, etc.

  The general idea is to wrap a more complicated storage facility in a simple,
  uniform interface, keeping the freedom of using the right tools for the job.
  In particular, a Datastore can aggregate other datastores in interesting ways,
  like sharded (to distribute load) or tiered access (caches before databases).

  While Datastores should be written general enough to accept all sorts of
  values, some implementations will undoubtedly have to be specific (e.g. SQL
  databases where fields should be decomposed into columns), particularly to
  support queries efficiently.

  '''

  # Main API. Datastore implementations MUST implement these methods.

  def get(self, key):
    '''Return the object named by key or None if it does not exist.'''
    raise NotImplementedError

  def put(self, key, value):
    '''Stores the object `value` named by `key`.'''
    raise NotImplementedError

  def delete(self, key):
    '''Removes the object named by `key`.'''
    raise NotImplementedError

  def query(self, query):
    '''Returns an iterable of objects matching criteria expressed in `query`'''
    raise NotImplementedError
```

shims

Sometimes common functionality can be compartmentalized into logic that can be plugged in or not. For example, serializing and deserializing data as it is stored or extracted is a very common operation. Likewise, applications may need to perform routine operations as data makes its way from the top-level logic to the underlying storage.

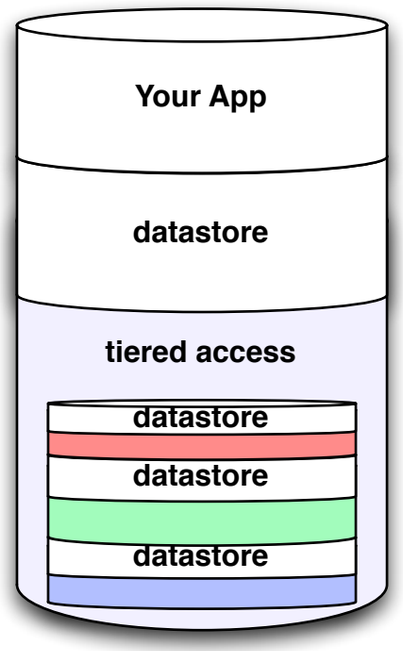
To address this use case in an elegant way, datastore uses the notion of a `shim` datastore, which implements all four main *datastore operations* in terms of an underlying child datastore. For example, a json serializer datastore could implement `get` and `put` as:

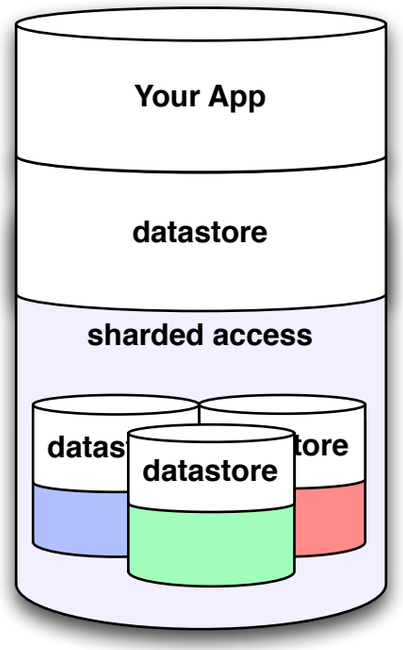
```
def get(self, key):
  value = self.child_datastore.get(key)
  return json.loads(value)

def put(self, key, value):
  value = json.dumps(value)
  self.child_datastore.put(key, value)
```

collections

Grouping datastores into datastore collections can significantly simplify complex access patterns. For example, caching datastores can be checked before accessing more costly datastores, or a group of equivalent datastores can act as shards containing large data sets.

As shims, datastore collections also derive from datastore, and must implement the four datastore operations *(get, put, delete, query).*

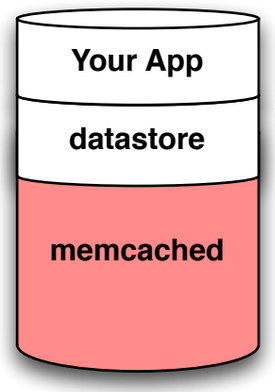




examples:

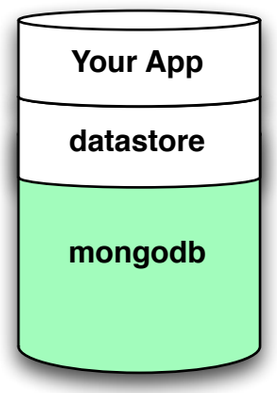
examples: memcached

```
>>> import pylibmc
>>> import datastore
>>> from datastore.impl.memcached import MemcachedDatastore
>>> mc = pylibmc.Client(['127.0.0.1'])
>>> ds = MemcachedDatastore(mc)
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```



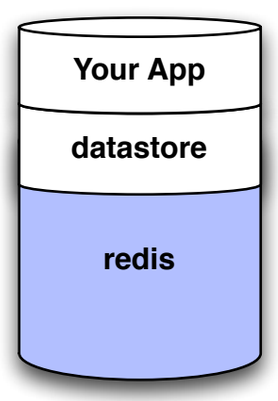
examples: mongodb

```
>>> import pymongo
>>> import datastore
>>> from datastore.impl.mongo import MongoDatastore
>>>
>>> conn = pymongo.Connection()
>>> ds = MongoDatastore(conn.test_db)
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```



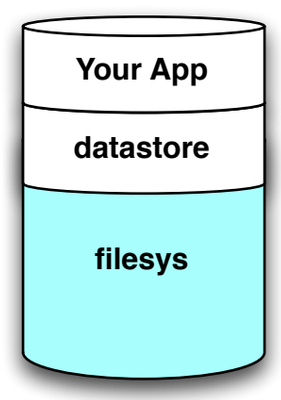
examples: redis

```
>>> import redis
>>> import datastore
>>> from datastore.impl.redis import RedisDatastore
>>> r = redis.Redis()
>>> ds = RedisDatastore(r)
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```



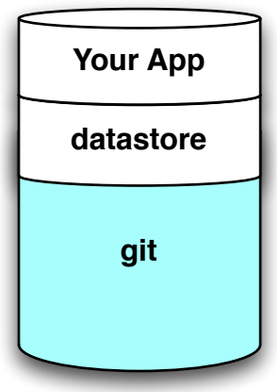
examples: filesystem

```
>>> import datastore
>>> from datastore.impl.filesystem import FileSystemDatastore
>>>
>>> ds = FileSystemDatastore('/tmp/.test_datastore')
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```



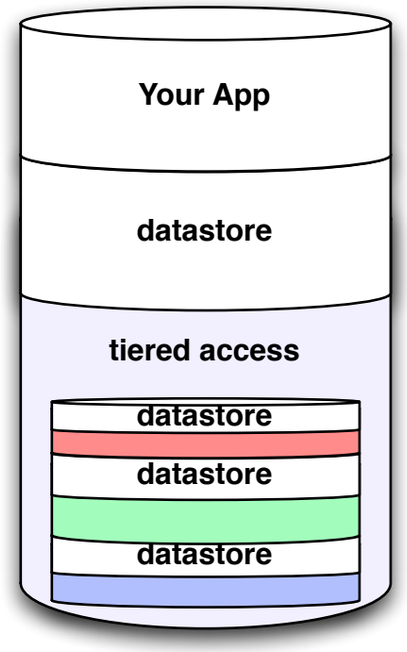
examples: git

```
>>> import datastore
>>> from datastore.impl.git import GitDatastore
>>>
>>> ds = GitDatastore('/tmp/.test_datastore')
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```



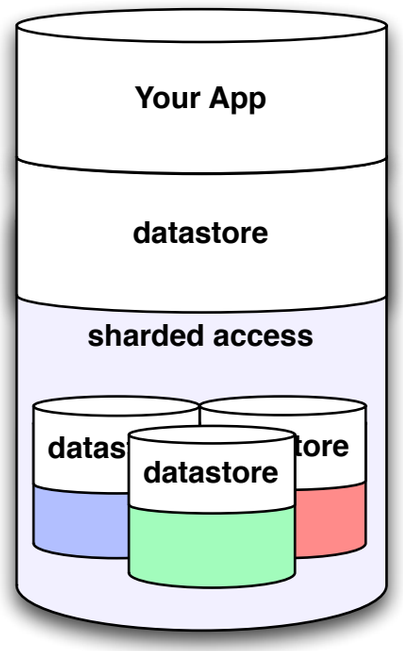
examples: tiered access

```
>>> import pymongo
>>> import datastore
>>>
>>> from datastore.impl.mongo import MongoDatastore
>>> from datastore.impl.lrucache import LRUCache
>>> from datastore.impl.filesystem import FileSystemDatastore
>>>
>>> conn = pymongo.Connection()
>>> mongo = MongoDatastore(conn.test_db)
>>>
>>> cache = LRUCache(1000)
>>> fs = FileSystemDatastore('/tmp/.test_db')
>>>
>>> ds = datastore.TieredDatastore([cache, mongo, fs])
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```



examples: shard access

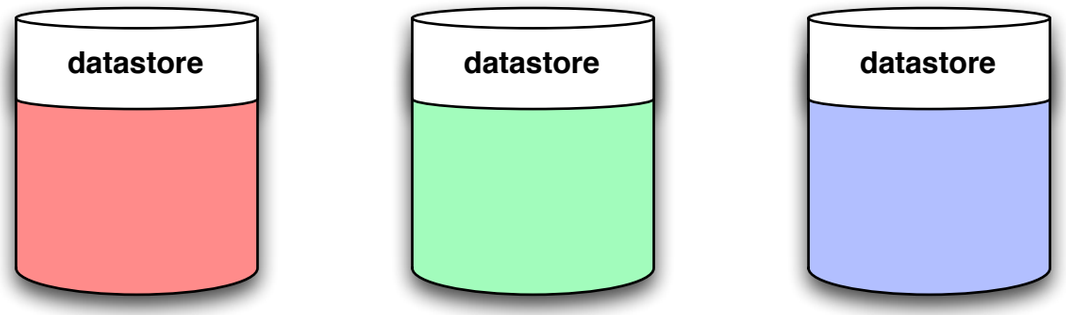
```
>>> import datastore
>>>
>>> shards = [datastore.DictDatastore() for i in range(0, 10)]
>>>
>>> ds = datastore.ShardedDatastore(shards)
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```



datastore

unified API for multiple data stores

https://github.com/jbenet/datastore



Thank You! Questions?

Juan Batiz-Benet
jbenet@cs.stanford.edu