Big Friendly Datastore

Release 0.0.1

Nicholas H.Tollervey

Oct 02, 2022

CONTENTS

Organic data collaboration, from the ground up.

This is an experiment in APIs, data storage, aggregation and discovery.

CHAPTER

ONE

DEVELOPER SETUP

This project uses Python 3.8+.

1. Clone the repository.
2. Create and start a new virtual environment.
3. `pip install -r requirements.txt`
4. Type `make` to see a list of common developer related tasks. For instance, to run the full test suite and code checks type: `make check`.

CHAPTER

TWO

CORE CONCEPTS

- Objects represent things and have a unique unicode name. BFG doesn't impose any further constraints on the name, except uniqueness. However, naming conventions are likely to evolve and/or be specified by mutual agreement of users working together across domains.
- Namespaced tags annotate data on openly writeable objects. The combination of namespace (who) and tag (what) provide meaningful context for the data tagged onto the object.
- Tag values containing data about objects are typed and can be queried via predicate based comparison operations (`and`, `or`, `not`, `<`, `>`, `=`, `<=`, `>=`, `(`, `)`), or naive string matching a la SQL `like` (and case insensitive `ilike`) pattern matching operator on string values. Queries return specified tags on matching objects.
- Namespaces have admins and tags have users and readers. Admins configure the namespaces and tags belonging to their namespaces, users may annotate objects with the namespaces/tags and readers can see the namespaces/tags and their associated values.
- Interrogate individual objects for readable namespaces/tags (that may match a pattern).
- Events are raised when specific changes happen in the datastore. These are configured to call web-hooks so third parties can follow what's going on. The event log can be used observe how the object and associated values changed through time (i.e. versioning).
- Data types understood by BFD: string, boolean, integer, floats, datetime and duration. Geospatial types may also be added soon. Blobs of arbitrary bytes may also be stored (as a URL that references raw data identified by mime-type). There is no such thing as "null". If a value isn't known, the tag is removed (but its historic presence is retained in the event log).

CHAPTER

THREE

IMPLEMENTATION

- Delivered via a REST API. Query results returned as either JSON or CSV.
- Admins, users and readers are expressed as a "whitelist", where an empty list means "everyone". For instance, if the readers are set to, `[]`, then everyone can see the namespace/tag. If the users are set to, `["nicholas", ]`, then only the user identified as "nicholas" can annotate with the namespace/tag. If the admins are set to, `["mary", "penelope", ]`, then only the users identified as "mary" and "penelope" may change the behaviour of the namespace and the tags contained therein.

CHAPTER

FOUR

ACKNOWLEDGEMENTS

Many of the ideas found herein have evolved from those used in FluidDB by Fluidinfo, a defunct startup project I was involved with between 2009-2012 (when it folded). Special mention to Terry Jones for much of the original thought behind this, and to Nick Radcliffe for subsequent stimulating exploration of the concepts involved.

Why this? Why now?

I find myself in need of such a data store, and since FluidDB is no more, I need to reheat the ocean with my own plastic kettle.

4.1 Contents

4.1.1 Contributing

Hey! Many thanks for wanting to improve BFD.

Contributions are welcome without prejudice from *anyone* irrespective of age, gender, religion, race or sexuality. If you're thinking, "but they don't mean me", *then we especially mean YOU*. Good quality code and engagement with respect, humour and intelligence wins every time.

- If you're from a background which isn't well-represented in most geeky groups, get involved - *we want to help you make a difference*.
- If you're from a background which *is* well-represented in most geeky groups, get involved - *we want your help making a difference*.
- If you're worried about not being technical enough, get involved - *your fresh perspective will be invaluable*.
- If you think you're an imposter, get involved.
- If your day job isn't code, get involved.
- This isn't a group of experts, just people. Get involved!

We expect contributors to follow the spirit of *being together*.

Feedback may be given for contributions and, where necessary, changes will be politely requested and discussed with the originating author. Respectful yet robust argument is most welcome.

Contributions are subject to the following caveats: the contribution was created by the contributor who, by submitting the contribution, is confirming that they have the authority to submit the contribution and place it under the license as defined in the *LICENSE* file found within this repository. If this is a significant contribution the contributor should add themselves to the AUTHORS file found in the root of BFD's repository. Contributors agree, for the sake of convenience, that copyright passes exclusively to Nicholas H.Tollervey on behalf of the BFD project.

Checklist

- If your contribution includes non-obvious technical decision making please make sure you document this.
- Your code should be commented in *plain English* (British spelling).
- If your contribution is for a major block of work and you've not done so already, add yourself to the AUTHORS file following the convention found therein.
- We have 100% test coverage - include tests to maintain this!
- **Before submitting code ensure coding standards and test coverage by running**::

```
make check
```

- If in doubt, ask a question. The only stupid question is the one that's never asked.
- Most importantly, **Have fun!** :-)

4.1.2 On Being Together

From here. Here's why.

4.1.3 BFD API

Interactions with the Big Friendly Datastore are via a REST-ful API described below.

The five core concepts needed to understand the API are:

1. *Users* - real humans who interact with the BFD. Admin users are able to create new users. All users interact with the BFD via the API and use either token-based HTTP authentication via an HTTP `Authorization` header in all requests, or via session management if interacting with the BFD via its browser based frontend.
2. *Namespaces* - define who is annotating data. Namespaces have a unique name and a free text description to provide context. Users are automatically given a namespace with the same name as their username. Further namespaces may be created to represent organisational units. Namespaces must have at least one user, and potentially many users, to act as administrators. They define the tags and the permissions relating to tags that belong to the Namespace.
3. *Tags* - define what is annotated to objects. Tags belong to a parent namespace, so it's possible to understand who is annotating what within the BFD. Tags have a name that is unique within the parent namespace and a free text description to provide context. Tags can be either public (anyone can read values associated with them) or private (only whitelisted individuals can read values annotated with them). Furthermore, only namespace admins or those users listed in a "users" whitelist are able to use the tag to annotate values onto an object. Finally, tags are typed so if a tag is for storing a date-time value, an operation to use it to annotate a string value will fail.
4. *Objects* - represent things in the universe. Objects are identified by a unique name. What an object represents can be found out by looking at the aggregation of values annotated to the object via BFD's user's namespaces and tags.
5. *Values* - are the data associated with an object via a namespace/tag combination. Values can be used with *the query language* to find objects of interest.

Users

`GET /u/<username>`

Returns limited information about the user identified by `username`. Admin users and the user identified by `username` see a more comprehensive profile.

`POST /u/<username>`

Use JSON data to update the referenced user. Only works if the request is made by an admin user of the user identified by `username`.

`POST /u/new`

Creates a new user from the JSON data provided by an admin user.

Namespaces

`GET /n/<namespace>`

Returns the description associated with the referenced `namespace`.

`PUT /n/<namespace>`

Use JSON data to update the referenced `namespace`. Only works if the request is made by a global admin user or a user listed as the referenced namespace's admin.

`POST /n/new`

Creates a new namespace from the JSON data provided by an admin user.

Tags

`GET /t/<namespace>/<tag>`

Returns the description associated with the referenced `namespace` / `tag` pair.

`PUT /t/<namespace>/<tag>`

Use JSON data to update the referenced `namespace` / `tag` pair. Only works if the request is made by a global admin user or a user listed as the referenced namespace's admin.

`POST /t/<namespace>/new`

Creates a new namespace from the JSON data provided by a global admin user or a user listed as the referenced namespace's admin.

Objects, Tags and Values

`GET /o/<object_id>/<namespace>/<tag>`

Get the value annotated by the referenced `namespace` / `tag` pair on the referenced `object`. Will return 404 if the value doesn't exist or if the callee doesn't have read permission for the referenced `tag`.

`POST /o/<object_id>/<namespace>/<tag>`

Annotate the value in the request, using the `namespace` / `tag` pair onto the referenced `object`. Will only work for global admins, namespace admins or those with `use` permission for the referenced tag.

`DELETE /o/<object_id>/<namespace>/<tag>`

Remove the value annotated on the referenced object by the namespace/tag pair. Will only work if the user has "use" permission on the referenced tag.

`GET /o/<object_id>`

Get a list of visible namespace/tag pairs on the referenced object.

`POST /o/<object_id>`

Annotate a list of `namespace/tag: value` pairs onto the referenced object. Will only work if the user has permission to annotate with all the referenced namespace / tag pairs.

`POST /o/<object_id>/delete`

Delete a list of `namespace/tag` annotations from the referenced object.

`POST /o/select`

Select a list of namespace/tag pairs from objects matching a query. Expressed as a JSON object.

`POST /o/update`

Update a list of `namespace/tag: value` pairs onto objects matching a query. Expressed as a JSON object.

`POST /o/delete`

Delete a list of `namespace/tag` pairs from objects matching a query. Expressed as a JSON object.

`POST /o/bulk`

Bulk annotate a collection of specified objects with associated object specific `namespace/tag: value` pairs. Expressed as a JSON object. This is how to do bulk imports/updates.

4.1.4 BFQL

The Big Friendly Query Language (BFQL - "buffquill") is a very simple way to match objects by tag-values. BFQL is designed to be expressive and easy to read from a human perspective. BFQL is an English query language in that its keywords come from English natural language.

BFQL is used in three ways, to match:

1. objects where a list of specified tag-values are returned,
2. objects that require a bulk update of specific tag-values, or
3. objects from which specified tags should be deleted.

The BFQL is heavily inspired by the query language created for FluidInfo.

Tags are written as a unique `namespace/tag` path.

The following kinds of queries are possible and depend on the type of the tag used in the query.

All Tags

The type of the tag makes no difference to these sorts of query.

- Presence: `has library/title` will return all objects that have values annotated by the `library/title` tag.
- Absence: `missing library/title` will return all objects that have NOT been annotated with the `library/title` tag. Due to the potential for huge numbers of results, this query can only be used in conjunction with another via the `and` operator.
- Conjugation (and): `library/summary matches "whales" and library/pages < 100` will return all objects that have the `library/summary` and `library/pages` tags and whose values match `"whales"` and `less than 100` respectively.
- Disjunction (or): `library/summary matches "whales" or library/summary matches "dolphins"` will return all objects that have the `library/summary` tag and whose value matches either `"whales"` or `"dolphins"`.
- Grouping: `has library/title and (nicholas/rating > 5 or terry/rating > 7)` returns all objects where the queries within the parenthesis are evaluated together before being conjugated with the results of the outer query to produce the final result set of object_ids.

String Based Tags

The following queries are possible on tags that are a type of `string` (representing arbitrary textual data) or `pointer` (representing a URL pointing to something elsewhere on the internet).

When writing strings, enclose them within double quotes.

- Equality: `library/title is "Moby Dick"` will return all objects with the `library/title` tag whose value is exactly `"Moby Dick"`.
- Case insensitive equality: `library/author iis "moby dick"` will return all objects with the `library/title` tag whose value is a case insensitive match for `"moby dick"`.
- Text matching: `library/summary matches "whales"` will return all objects with the `library/summary` tag whose value contains a case-sensitive match of the word `"whales"`.
- Case insensitive text matching: `library/summary imatches "whales"` will return all objects with the `library/summary` tag whose value contains a case-INsensitive match of the word `"whales"`.

Boolean Tags

The following queries work on tags that are `boolean`.

- Truth: `nicholas/has_met is true` will return all objects with the `nicholas/has_met` tag whose value is true.
- Falsity: `nicholas/has_met is false` will return all objects with the `nicholas/has_met` tag whose value is false.

Scalar Tags

The following queries only work with scalar tags: `integer`, `float`, `datetime` and `duration`.

Integer and float values can be used with each other for comparison.

Integers are written as a sequence of digits: `1234` Floats are written as a sequence of digits with a decimal point: `1.234` Datetimes are written precisely to the second: `2020-09-24T15:30:30` (`YYYY-MM-DDTHH:MM:SS` with the `T` separating the date and time portions for readability reasons) or with just the date: `2020-09-24` (`YYYY-MM-DD`). Timezone offset may also be appended `2020-09-24T15:30:30-08:00` (`YYYY-MM-DDTHH:MM:SS[+|-]HH:MM`). These patterns follow the recommendations in the W3C's Date and Time Formats Note. Durations are expressed as exact numbers of days (denoted by an integer followed by `d`) or seconds (an integer followed by `s`): `12d` or `360s`. Other durations should be constructed by multiplying days or seconds to the right value.

- Equal: `game/score = 1000` will return all objects with the `game/score` tag whose value is exactly the integer `1000`.
- Not equal: `game/score != 1000` will return all objects with the `game/score` tag whose value is NOT equal to the integer `1000`.
- Greater than: `bookshop/delivery_weight_kg > 1.2` will return all objects with the `bookshop/delivery_weight_kg` tag whose value is greater than the float `1.2`.
- Less than: `bookshop/delivery_weight_kg < 1.5` will return all objects with the `bookshop/delivery_weight_kg` tag whose value is less than the float `1.5`.
- Greater than or equal to: `employee/dob >= 1973-08-19` will return all objects with the `employee/dob` tag whose value is greater than or equal to the date `1973-08-19` (19th August, 1973).
- Less than or equal to: `employee/probation_period <= 365d` will return all objects with the `employee/probation_period` tag whose value is less than or equal to a duration of 365 days (`365d`).

Binary Tags

Binary tags are, by their nature opaque (they store arbitrary binary information). However they do automatically store the value's MIME type.

- Type of: `library/audiobook is mime:audio/mpeg` will return all objects with the `library/audiobook` tag whose mime type is listed as `audio/mpeg` (i.e. MP3). MIME type information is always treated as case insensitive and must start with `mime:` (to differentiate MIME values from tag paths).

The current list of valid MIME types (and what file types they represent) can be found on the IANA's website.

4.1.5 Architecture

TBD

4.1.6 Design Decisions

Select implementation language

Criteria:

- Mature.
- Well known.
- Stable and comprehensive eco-system.

Candidates:

- Elixir
- Python
- JavaScript (Node)

Commentary:

N/A

- Status: approved.
- Decision: Python.
- Author: ntoll.

Select web framework

Criteria:

- Mature / feature-ful.
- Easy to implement REST.
- Support for relational databases.

Candidates:

- Flask
- Django

Commentary:

I prefer the all-in-one / tightly integrated approach of Django.

- Status: approved.
- Decision: Django (with Django Rest Framework).
- Author: ntoll.

Select database backend

Criteria:

- Mature / feature-ful.
- Good performance with known routes to scaling.
- Potential for Geo-data.
- Supported by Django.

Candidates:

- Sqlite
- Postgresql
- MySql

Commentary:

Postgresql is the only solution that meets all the criteria.

- Status: approved.
- Decision: Postgresql.
- Author: ntoll.

Tag Values

Criteria:

Namespaced tags attach values to objects. What is the "type" system for the BFD. It needs to:

- Be clear.
- Promote consistency.
- Promote thoughtful definition of data.
- Avoid unexpected value related side effects.

Candidates:

Tag/value types are:

- Static.
- Dynamic.

In addition, should the BFD allow for null values to be associated with a tag on an object?

Commentary:

Knowing and enforcing the type of data at the tag level ensures clarity and consistency. Furthermore, forcing the creator of a tag to define the tag's type ensures they think carefully about the sort of value the tag will be used to represent. Finally, by disallowing null values users can be certain that any value they retrieve will be guaranteed to be a value of the type of the tag. If a value of a tag for a certain object is unknown, then that tag should not exist on that object (otherwise, what's stopping every tag from being attached to every object with the value null, until someone updates the value?).

- Status: approved.
- Decision: static / no null.
- Author: ntoll.

Naming constraints

Criteria:

Namespaces, tags, objects and users are named things in the BFD. Simple and URL friendly conventions are needed so endpoints are readable and accessible to multiple locales.

Candidates:

- Limited by length.
- Limited to alpha-numeric / URL friendly characters (_ and −).
- UTF-8.

Commentary:

Some REST based services end up with huge illegible URLs (because of escape characters and other limitations).

If users wish to share BFD related URLs, the URL should make it obvious what data they're going to get. Furthermore, such URL related names should be easy to write. As a result, the names that could be found in URLs should be limited in such a way that they encourage read/write-ability.

Limiting the length of names ensures readability, limiting to alpha-numeric and URL friendly characters (− and _) ensures the URL isn't full of hard-to-decypher escape sequences and allowing UTF-8 means "alpha-numeric" includes a wide range of characters from non-English/Latin character sets.

- Status: approved.
- Decision: limited length / limited to alpha-numeric / UTF-8.
- Author: ntoll.

Select parser framework

Criteria:

- Simple.
- Easy.
- Well documented.

Candidates:

- SLY
- The various parsers listed here: https://tomassetti.me/parsing-in-python/

Commentary:

I've already used SLY, but wanted to look around to see if there were better alternatives. It turns out, for a quick and simple solution with great docs, nothing yet beats SLY.

- Status: approved.

- Decision: SLY.
- Author: ntoll.

4.1.7 License

The Big Friendly Datastore is licensed under the MIT License modified by the "Commons Clause" License Condition v1.0.

Both are reproduced below.

Commons Clause License Condition v1.0

The Software is provided to you by the Licensor under the License, as defined below, subject to the following condition.

Without limiting other conditions in the License, the grant of rights under the License will not include, and the License does not grant to you, the right to Sell the Software.

For purposes of the foregoing, "Sell" means practicing any or all of the rights granted to you under the License to provide to third parties, for a fee or other consideration (including without limitation fees for hosting or consulting/support services related to the Software), a product or service whose value derives, entirely or substantially, from the functionality of the Software. Any license notice or attribution required by the License must also include this Commons Clause License Condition notice.

Software: Big Friendly Datastore (BFD)

License: MIT (Copied Below)

Licensor: Nicholas H.Tollervey

MIT License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4.1.8 Authors

The following folks have made significant contributions to BFD's development by contributing code:

Nicholas H.Tollervey - ntoll@ntoll.org (creator and maintainer).