JamDB Documentation

Release 0.3.0

Center for Open Science

Oct 22, 2018

Contents

JamDB is a schema-less, immutable database that can optionally enforce a schema and stores provenance. It supports efficient full-text search, filtering by nested keys, and is accessible a REST API.

It has pluggable storage backends. It defaults to using both MongoDB and Elasticsearch.

The full BDD test suite lives in the features directory and describes in human-readable prose how each endpoint works. What it's expecting for input and what it expects for output.

Contents:

CHAPTER 1

Prerequisites

This tutorial assumes that the JamDB server you're interacting with will be at `http://localhost:1212`. It also assumes access to a terminal shell on either OSX or Linux with the `curl` command installed and executable.

CHAPTER 2

Installing JamDB

JamDB is written in Python using the Tornado Web Framework.

It requires MongoDB version >= 3.2 and Elasticsearch version 1.7.

2.1 Steps

1. Clone the JamDB git repo
2. Create a new virtual environment for JamDB called `jam`
3. Setup your new virtual environment for JamDB
   - Once in your virtual env, change into the directory you cloned JamDB into and execute `python setup.py develop`
   - Then execute `pip install -r requirements.txt`
4. Install MongoDB and Elasticsearch
5. Confirm they're both running: `ps aux | grep -i 'elasticsearch\|mongod'`
6. Run `jam server`
7. In another terminal, run `curl http://localhost:1212/v1/namespaces/` to confirm you can connect to the server. The response should be: `{"data": [], "links": {}, "meta": {"perPage": 50, "total": 0}}`

Next you should see the Namespaces section for instructions on creating your first namespace.

CHAPTER 3

Namespaces

A namespace is the equivalent of a database in MongoDB or PostgreSQL.

It will act as a top-level container for any collections you make and store permissions that apply to itself and cascade to anything inside of it.

Administrator privileges are required to make any modification to a namespace.

3.1 Creating a namespace

Currently, there is no way to create a namespace through the API. If you're working with a remote instance of JamDB, contact the server administrator to create a namespace. If you're running a local instance of JamDB, you can create a namespace by running `jam create <namespace id> -u 'jam-ProgrammingLanguages:Programmers-Ash`.

> We'll be using `ProgrammingLanguages` as the example namespace id for the rest of this document. Namespace ids are case-sensitive.

Once your namespace is setup, you'll need to send the proper `Authorization` header to access it.

3.2 Authorizing against a namespace

JamDB uses json web tokens, JWT for short, in the `Authorization` header or the `token` query string parameter.

There are three ways to acquire a JWT:

1. Contact the server administrator and request a temporary token.
2. Authenticate via the Auth Endpoint
3. If you are running a JamDB server locally you can generate a token by running `jam token 'jam-ProgrammingLanguages:Programmers-Ash'`

   We'll be using `mycooljwt` as the example JWT for the rest of this document.

3.3 Investigating a namespace

You can get information about your namespace by making an HTTP request using curl, Paw, or a similar program.

HTTP Request:

```
GET /v1/namespaces/ProgrammingLanguages HTTP/1.1
Authorization: mycooljwt
```

HTTP Response:

```
{
  "data": {
    "id": "ProgrammingLanguages",
    "type": "namespaces",
    "attributes": {
      "name": "ProgrammingLanguages",
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN"
      }
    },
    "meta": {/*...*/},
    "relationships": {/*...*/}
  }
}
```

Permissions may be different depending on how you got your JWT.

3.4 Namespace Permissions

Giving other users permissions to a namespace is easy.

We can update our namespace in two ways.

We can use jsonpatch to add just the field we want.

HTTP Request:

```
PATCH /v1/namespaces/ProgrammingLanguages HTTP/1.1
Authorization: mycooljwt
Content-Type: Content-Type: application/vnd.api+json; ext=jsonpatch

[{"op": "add", "path": "/permissions/jam-ProgrammingLanguages:Programmers-*", "value
↪": "READ"}]
```

HTTP Response:

```
{
  "data": {
    "id": "ProgrammingLanguages",
    "type": "namespaces",
    "attributes": {
      "name": "ProgrammingLanguages",
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-*": "READ",
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN"
```

(continues on next page)

(continued from previous page)

```
        }
      },
      "meta": {/*...*/},
      "relationships": {/*...*/}
    }
}
```

Many jsonpatch objects may be sent at once.

HTTP Request:

```
PATCH /v1/namespaces/ProgrammingLanguages HTTP/1.1
Authorization: mycooljwt
Content-Type: Content-Type: application/vnd.api+json; ext=jsonpatch

[
  {"op": "add", "path": "/permissions/jam-ProgrammingLanguages:Programmers-*", "value
↪": "READ"},
  {"op": "add", "path": "/permissions/jam-ProgrammingLanguages:Programmers-Misty",
↪"value": "ADMIN"},
  {"op": "add", "path": "/permissions/jam-ProgrammingLanguages:Programmers-Brock",
↪"value": "ADMIN"}
]
```

HTTP Response:

```
{
  "data": {
    "id": "ProgrammingLanguages",
    "type": "namespaces",
    "attributes": {
      "name": "ProgrammingLanguages",
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-*": "READ",
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN",
        "jam-ProgrammingLanguages:Programmers-Misty": "ADMIN",
        "jam-ProgrammingLanguages:Programmers-Brock": "ADMIN",
      }
    },
    "meta": {/*...*/},
    "relationships": {/*...*/}
  }
}
```

Or we can just PATCH up our updated data and let the JamDB server figure it out.

This is potentially a destructive action. Any existing permissions will be completely replaced. If you want to do a partial update use the JSONPatch method above.

HTTP Request:

```
PATCH /v1/namespaces/ProgrammingLanguages HTTP/1.1
Authorization: mycooljwt

{
  "data": {
    "id": "ProgrammingLanguages",
```

(continues on next page)

(continued from previous page)

```
    "type": "namespaces",
    "attributes": {
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-*": "READ",
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN"
      }
    }
  }
}
```

HTTP Response:

```
{
  "data": {
    "id": "ProgrammingLanguages",
    "type": "namespaces",
    "attributes": {
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-*": "READ",
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN"
      }
    },
    "meta": {/*...*/},
    "relationships": {/*...*/}
  }
}
```

Collections are the next step in the documentation.

CHAPTER 4

Collections

A collection is a bucket for arbitrary data. It's the equivalent of a table in a SQL or NoSQL database. It **may** enforce a schema on its data. It also **may** extend the permissions of the namespace.

4.1 Creating a collection

To create a collection we just have to POST the data about our collection to our namespace's collections endpoint.

HTTP Request:

```
POST /v1/namespaces/ProgrammingLanguages/collections HTTP/1.1
Authorization: mycooljwt

{
  "data": {
    "id": "Functional",
    "type": "collections",
    "attributes": {}
  }
}
```

HTTP Response:

```
{
  "data": {
    "id": "ProgrammingLanguages.Functional",
    "type": "collections",
    "attributes": {
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN"
      }
    },
    "meta": {/*...*/},
```

(continues on next page)

(continued from previous page)

```
    "relationships": {/*...*/}
  }
}
```

Please Note:

- We have been given ADMIN access to this collection because we created it.
- The id has been extended to `ProgrammingLanguages.Functional` because the `Functional` collection belongs to the `ProgrammingLanguages` namespace.
- The full id (`ProgrammingLanguages.Functional`) or the truncated id (`Functional`) may be used when sending update requests.
- The truncated id will be used for the rest of this document.

Now our fellow programmers are free to browse through the Functional collection, which we will add information into later.

It's a lot of work to load all this data into our collection by ourselves. Let's get some help!

4.2 Adding collection permissions

We want to give a couple of our friends access to insert data into this collection but we don't want to grant them access to all of our collection.

Using collection-level permissions, we can do just that.

Collections can be updated the same way that namespace are, either POSTing or PATCHing data.

Please note:

- The JSONPatching format is a bit nicer to look at so we'll be using that method for the rest of this document.
- Keep in mind that you could just as easily PATCH the updated document instead.

```
PATCH /v1/namespaces/ProgrammingLanguages/collections/Functional HTTP/1.1
Authorization: mycooljwt

[
  {"op": "add", "path": "/permissions/jam-ProgrammingLanguages:Programmers-Gary",
↪"value": "CREATE, UPDATE"},
  {"op": "add", "path": "/permissions/jam-ProgrammingLanguages:Programmers-
↪ProfessorOak", "value": "CREATE, UPDATE"},
  {"op": "add", "path": "/permissions/jam-ProgrammingLanguages:Programmers-
↪ProfessorBirch", "value": "CREATE, UPDATE"}
]
```

```
{
  "data": {
    "id": "ProgrammingLanguages.Functional",
    "type": "collections",
    "attributes": {
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN",
        "jam-ProgrammingLanguages:Programmers-Gary": "CU",
```

(continues on next page)

(continued from previous page)

```
        "jam-ProgrammingLanguages:Programmers-ProfessorOak": "CU",
        "jam-ProgrammingLanguages:Programmers-ProfessorBirch": "CU"
      }
    },
    "meta": {/*...*/},
    "relationships": {/*...*/}
  }
}
```

Please note:

- Our permissions got compressed from CREATE, UPDATE to CU. This is the format JamDB stores permissions. CREATE, UPDATE and CU are equivalent. We could have set Gary's, Professor Oak's, and Professor Birch's permissions to CU but CREATE, UPDATE is a bit easier to read.
- Remember that we gave `jam-ProgrammingLanguages:Programmers-*` READ permissions earlier.
- Whenever Gary, Professor Oak, or Professor Birch access the Functional collection they will have that permission added to their CREATE, UPDATE permissions.

While we trust our friends, we may want to enforce data validation.

We are going to leverage the power of JSONSchema and JamDB's schema validation for this.

Note: For the sake of length and readability we are going to use an abbreviated schema. The actual Functional schema is much longer because we're huge nerds.

HTTP Request:

```
PATCH /v1/namespaces/ProgrammingLanguages/collections/Functional HTTP/1.1
Authorization: mycooljwt

[
  {
    "op": "add",
    "path": "/schema",
    "value": {
      "type": "jsonschema",
      "schema": {
        "id": "/",
        "type": "object",
        "properties": {
          "name": {
            "id": "name",
            "type": "string"
          },
          "type": {
            "id": "type",
            "type": "string"
          },
          "Number": {
            "id": "Number",
            "type": "integer"
          },
          "Interpreted": {
            "id": "Interpreted",
            "type": "boolean"
          }
```

(continues on next page)

(continued from previous page)

```
        },
        "required": [
          "name",
          "type",
          "Number",
          "Interpreted"
        ]
      }
    }
  }
]
```

Please Note:

- `schema.type` must be set to the type of the schema. The actual schema lives at `schema.schema`. This is so that JamDB may support other forms of schema validation in the future. Currently JSONSchema is the only supported validator.
- `$` are illegal in JamDB key names.
- Make sure not to use `$schema` or `$ref` in your JSONSchema.

HTTP Response:

```
{
  "data": {
    "id": "ProgrammingLanguages.Functional",
    "type": "collections",
    "attributes": {
      "permissions": {
        "jam-ProgrammingLanguages:Programmers-Ash": "ADMIN",
        "jam-ProgrammingLanguages:Programmers-Gary": "CU",
        "jam-ProgrammingLanguages:Programmers-ProfessorOak": "CU",
        "jam-ProgrammingLanguages:Programmers-ProfessorBirch": "CU"
      },
      "schema": {
        "type": "jsonschema",
        "schema": {
          "id": "/",
          "type": "object",
          "properties": {
            "type": {
              "id": "type",
              "type": "string"
            },
            "Number": {
              "id": "Number",
              "type": "integer"
            },
            "Interpreted": {
              "id": "Interpreted",
              "type": "boolean"
            }
          },
          "required": [
            "name",
            "type",
            "Number",
```

(continues on next page)

(continued from previous page)

```
                "Interpreted"
              ]
            }
          }
        },
        "meta": {/*...*/},
        "relationships": {/*...*/}
    }
}
```

Documents would be a good place to continue on to.

CHAPTER 5

Documents

A document is any **JSON object** with a string identifier that lives in a collection.

> Strings, numbers, and arrays are all valid JSON but the root of a document must be a JSON object.

Time for the fun part: Filling out the functional collection!

5.1 Creating Documents

Documents are created like anything else: by POSTing to functional collection's documents endpoint.

HTTP Request:

```
POST /v1/namespaces/ProgrammingLanguages/collections/Functional/documents HTTP/1.1
Authorization: mycooljwt

{
  "data": {
    "id": "Clojure",
    "type": "documents",
    "attributes": {
      "Number": 35,
      "Interpreted": true,
      "type": "JVM"
    }
  }
}
```

HTTP Response:

```
{
  "data": {
    "id": "ProgrammingLanguages.Functional.Clojure",
    "type": "documents",
```

(continues on next page)

(continued from previous page)

```
    "attributes": {
      "Number": 35,
      "Interpreted": true,
      "type": "JVM"
    },
    "meta": {/*...*/},
    "relationships": {/*...*/}
  }
}
```

For this next portion, we're going to assume that our friends have filled out the rest of the Functional collection for us. Such nice friends.

5.2 Filtering, Pagination, and Sorting

Now that we have all our data loaded up, let's search it. We'll start with finding all entries of the type `JVM`.

5.2.1 Filtering

- Filtering is available on the `documents` endpoint
- The query string parameter is `filter[{key}]={value}`
- `{key}` is the key that you want to filter on
- `{value}` is the value that you want to filter the key by
- `.s` are used to separate keys when referring to a nested object, `filter[nested.keys.like.this]=value`

5.2.2 Page size

- To save space we'll be using a page size of 2
- Page size may be anywhere between 0 and 100, inclusive, and defaults to 50
- The query string parameter is `page[size]={value}`

HTTP Request:

```
GET /v1/namespaces/ProgrammingLanguages/collections/Functional/documents?
↪filter[type]=JVM&page[size]=2 HTTP/1.1
Authorization: mycooljwt
```

HTTP Response:

```
{
  "data": [
    {
      "id": "ProgrammingLanguages.Functional.Clojure",
      "type": "documents",
      "attributes": {
        "Number": 35,
        "Interpreted": true,
```

(continues on next page)

(continued from previous page)

```
        "type": "JVM"
      },
      "meta": {/*...*/},
      "relationships": {/*...*/}
    }, {
      "id": "ProgrammingLanguages.Functional.Haskell",
      "type": "documents",
      "attributes": {
        "Number": 60,
        "Interpreted": false,
        "type": "native"
      },
      "meta": {/*...*/},
      "relationships": {/*...*/}
    },
  ],
  "links": {/*...*/}
}
```

5.2.3 Sorting

Next let's find the entry with the highest Number that is an JVM type.

This can be achieved by filtering on type and then sorting on Number.

Please note:

- Sorts may be done ascending or descending by prefixing the key you wish to sort on with + or -, respectively
- Sort order defaults to ascending
- Sort defaults to id
- If you want to sort on id, descending use `sort=-ref`. This is subject to change
- The query string parameter is `sort={order}{value}`

HTTP Request:

```
GET /v1/namespaces/ProgrammingLanguages/collections/Functional/documents?
↪filter[type]=JVM&page[size]=1&sort=Number HTTP/1.1
Authorization: mycooljwt
```

HTTP Response:

```
{
  "data": [
    {
      "id": "ProgrammingLanguages.Functional.Elixir",
      "type": "documents",
      "attributes": {
        "Number": 90,
        "Interpreted": false,
        "type": "erlang"
      },
      "meta": {/*...*/},
      "relationships": {/*...*/}
```

(continues on next page)

(continued from previous page)

```
    }
  ],
  "links": {/*...*/}
}
```

5.3 Searching

Finally, Gary is trying to remember the id of a specific entry but only remembers that is ends with "oq"

What an excellent opportunity for us to tap into JamDB's Elastic Search API.

Please note:

- The power of elasticsearch's query string syntax is exposed as the `q` query string parameter on the _search endpoint.
- In accordance with the other query string parameters, to query the id of a document use the `ref` key instead of id.
- The query string parameter is `q={url_escaped_elasticsearch_query}`

HTTP Request:

```
GET /v1/namespaces/ProgrammingLanguages/collections/Functional/documents?q=ref:*oq ␣
↪HTTP/1.1
Authorization: mycooljwt
```

HTTP Response:

```
{
  "data": [
    {
      "id": "ProgrammingLanguages.Functional.Coq",
      "type": "documents",
      "attributes": {
        "Number": 106,
        "Interpreted": true,
        "type": "OCaml"
      },
      "meta": {/*...*/},
      "relationships": {/*...*/}
    }
  ],
  "links": {/*...*/}
}
```

CHAPTER 6

Authentication

JamDB uses Json Web Tokens for authentication.

6.1 Authenticating

JamDB allows authentication through many providers. Currently `osf` and `self` are the only available providers.

A user may authenticate to JamDB by sending a properly formatted `POST` request to Jam's auth endpoint, `/v1/auth`

```
POST /v1/auth HTTP/1.1

{
  "data": {
    "type": "users",
    "attributes": {
      "provider": ...
      ...
    }
  }
}
```

Note: The elements of `attributes` have been left blank in this example as they vary per provider. The following sections will cover what each provider needs to properly authenticate

A successful authentication request will return the following data

```
{
    "data": {
        "id": "<type>-<provider>-<id>",
        "type": "users",
        "attributes": {
```

(continues on next page)

(continued from previous page)

```
            "id": "<id>",
            "type": "<type>",
            "provider": "<provider>",
            "token": "<jwt>",
        }
    }
}
```

`data.id` is the *user id* it will be matched against *user selectors* to calculate it's permissions.

`data.attributes.id` is the provider specific id for this user.

`data.attributes.type` is the *type of user* for this user.

`data.attributes.provider` is the provider that was used to authenticate as this user.

`data.attributes.token` is the jwt used to authorize requests to JamDB

6.1.1 OSF

You will need an OSF account and an OAuth2 access token to authenticate via the OSF provider.

You may sign up for an account at osf.io.

To acquire an access token you may either generate a personal access token in user settings or via an OAuth2 authorization flow of an OSF app.

```
GET /v1/auth HTTP/1.1

{
  "data": {
    "type": "users",
    "attributes": {
      "provider": "osf",
      "access_token": "<token>",
    }
  }
}
```

6.2 Authorizing

Authorization may be provided for an HTTP request in either the `Authorization` header or the `token` query parameter.

Note: The `Authorization` header takes precedence over the `token` query parameter

```
GET /v1/namespaces/ProgrammingLanguages HTTP/1.1
Authorization: mycooljwt
```

```
PUT /v1/namespaces/ProgrammingLanguages?token=mycooljwt HTTP/1.1
```

6.3 User Ids

User Ids are made of three parts separated by `-`s.

`<type>-<provider>-<id>`

> Note: `*`, `-` and `.` are illegal characters in user ids

6.3.1 Type

Currently there are 3 types, `user`, `anon`, and `jam`.

`user` indicates that the user was authenticated via a 3rd party service, such as the OSF, Google, or even Facebook.

`anon` indicates that the user simply requested a token to access JamDB, **anyone may be a anon user**.

`jam` indicates that the user was authenticated via a collection existing in jam.

6.3.2 Provider

A provider is simply the service that was used to authenticate.

In the case of the `user` type this may be `osf`, `google`, `facebook`, etc.

`anon` users do not have a provider.

For the `jam` user type, provider is the namespace and collection that the user "logged into" separated by a `:`. ie `ProgrammingLanguages:Functional`

6.3.3 Id

An id is any given string used by their provider to identify a user.

6.4 User Selectors

Selector	Meaning
`*`	Matches **ALL** users, authenticated or not
`<type>-*`	Matches all authenticated users with the type `<type>`
`<type>-<provide r>-*`	Matches all users of the given type that have authenticated via `<provider>`
`<type>-<provide r>-<id>`	Matches an exact user

6.4.1 User Selectors

Objective	Selector
Match everyone	`*`
Match all users authenticated via OSF	`user-osf-*`
Match all users authenticated via a 3rd party service	`user-*`
Match anonymous users	`anon-*`
Match a specific user	`user-osf-juwia`

CHAPTER 7

Data Modeling

JamDB is non-relational. A document is any **JSON object** with a string identifier that lives in a collection.

Strings, numbers, and arrays are all valid JSON but the root of a document must be a JSON object.

Your data model is up to you. You can use IDs to create pseudo-relationships.

```
+---------------------------------------------------------+
|AuthorizationNamespace                                   |
+---------------------------------------------------------+
|                                                         |
|   +---------------+                                     |
|   |UserCollection |                                     |
|   +---------------+                                     |
|   |Id             +----------+                          |
|   |Name           |          |                          |
|   |FavoriteColor  |          |                          |
|   |ShirtSize      |          |                          |
|   |Gender         |          +--------------------+     |
|   |Email          |          ||UserGroupCollection|     |
|   +---------------+          |--------------------+     |
|                              ||UserId             |     |
|   +---------------------------+GroupId            |     |
|   |GroupCollection||          |                   |     |
|   +---------------|          +-------------------+     |
|   |Id             ++                                    |
|   |Name           |                                     |
|   +---------------+                                     |
|                                                         |
+---------------------------------------------------------+
```

CHAPTER 8

Permissions

Please see:

- Namespace Permissions
- Collection Permissions

CHAPTER 9

Contributing

We welcome contributions via [GitHub](https://github.com/CenterForOpenScience/jamdb).

Before submitting your pull request, please make sure that all unit tests are passing, by running the command below:

```
behave
```

To preview changes to the documentation, install the requirements in dev-requirements.txt, then run the following:

```
cd docs
make html
```

CHAPTER 10

Limitations

Due to some of the underlying technologies, limits exist on the amount and types of data that may be stored.

10.1 Imposed by JamDB

- Namespace, collection, and document IDs may not excede 64 characters

10.2 Elasticsearch

- String values may not exceed 32766 bytes
- The string values `Infinity` and `-Infinity` may not be used where other documents would have a numeric value

10.3 MongoDB

- Integer values may not exceed 8 bytes
- Floating point values may not exceed 8 bytes
- Object keys may not start with `$`s
- Object keys may not contain `.`s
- Documents may not exceed 16 megabytes
- Documents may not exceed 100 levels of nesting

CHAPTER 11

API Semantics

11.1 Notes

- ids are must match the regex [\d\w-]{3,64}

11.2 Routes list

- `/v1/namespaces`
- `/v1/namespaces/<namespace_id>`
- `/v1/namespaces/<namespace_id>/collections`
- `/v1/namespaces/<namespace_id>/collections/<collection_id>`
- `/v1/namespaces/<namespace_id>/collections/<collection_id>/_search`
- `/v1/namespaces/<namespace_id>/collections/<collection_id>/documents`
- `/v1/namespaces/<namespace_id>/collections/<collection_id>/documents/<document_id>`
- `/v1/namespaces/<namespace_id>/collections/<collection_id>/documents/<document_id>/history`
- `/v1/namespaces/<namespace_id>/collections/<collection_id>/documents/<document_id>/history/<history_id>`
- `/v1/id/namespaces`
- `/v1/id/namespaces/<namespace_id>`
- `/v1/id/namespaces/<namespace_id>/collections`
- `/v1/id/collections/<namespace_id>.<collection_id>`
- `/v1/id/collections/<namespace_id>.<collection_id>/_search`

- `/v1/id/collections/<namespace_id>.<collection_id>/documents`
- `/v1/id/documents/<namespace_id>.<collection_id>.<document_id>`
- `/v1/id/documents/<namespace_id>.<collection_id>.<document_id>/history`
- `/v1/id/history/<namespace_id>.<collection_id>.<document_id>.<history_id>`

11.3 Extensions

11.3.1 JSONPatch

JamDB implements JSONAPI's jsonpatch extension as described here.

Example payloads and responses may be seen here, here, or here.

11.3.1.1 Deviations

JSONPatch is not currently supported with bulk operations.

11.3.2 Bulk

JamDB implements JSONAPI's bulk extension as described here.

Example payloads and responses may be seen here.

11.3.2.1 Deviations

Bulk deletes are not currently supported.

Bulk operations are not transactional. If a document creation fails for any reason it will not impede the creation of other documents. The failure will be returned in the `errors` key of the response JSON corresponding to it's index in the POSTed `data` field. The behavior is demonstrated here.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search