JIO

jIO - Homepage

Sven Franck

 © Nexedi SA

Table of Contents

1. Figure-1 - jIO LogojIO 4

2. What is a jIO Storage? 5

3. Promises 6

4. Another JavaScript Framework? Why use jIO? 7

5. Getting Started 8
   - 5.1 Source Code 8
   - 5.2 What is a Document? 8
   - 5.3 Hello World 9
   - 5.4 Posting Attachments 9

6. API - Quickguide 11

7. API - Storage Types 14
   - 7.1 LocalStorage 14
   - 7.2 MemoryStorage 14
   - 7.3 IndexedDB 15
   - 7.4 WebSQL 15
   - 7.5 DavStorage 16
   - 7.6 Dropbox 16
   - 7.7 Google Drive 17
   - 7.8 ERP5 Storage 17
   - 7.9 ZipStorage (Handler) 18
   - 7.10 ShaStorage (handler) 18
   - 7.11 UuidStorage (handler) 19
   - 7.12 QueryStorage (handler) 19
   - 7.13 CryptStorage (handler) 20
   - 7.14 UnionStorage (handler) 21
   - 7.15 FileSystemBridgeStorage (handler) 21
   - 7.16 Document Storage (handler) 22
   - 7.17 Replicate Storage (Handler) 22

8. jIO Query Engine 24
8.1 How to use Queries with jIO? 24

9. Creating Your Own Storage 26

10. Tips and Tricks 29
10.1 CreateJIO is not Async 29
10.2 Using Queries Outside jIO 29
10.3 Wildchard Query Parameter 30
10.4 JiO Query JSON Schemas and Grammar 30
10.5 Customizing jIO Query Search Keys 32
10.6 Overriding jIO Query Operators and Sorting 34
10.7 Partial Date/Time Matching in jIO Queries 34
10.8 jIO Query Key Schemas 35
10.9 Using a schema 36

11. Tests 37

12. FAQ 38

13. Licence 39

14. Examples 40

1. jIO

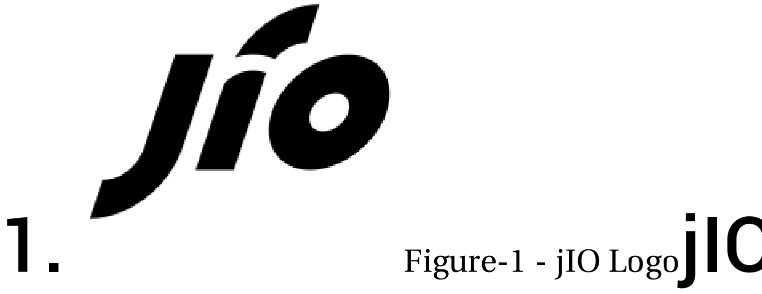
Figure-1 - jIO Logo

jIO is an abstract, promise-based JavaScript API that offers connectors to multiple storages (such as dropbox, s3, indexeddb) as well as special handlers for enhanced functionality (replication, encrpytion, querying). jIO allows to separate storage access from the application, provides a simple way to switch backends and create offline-capable, synchronizing applications. jIO is developed and maintained by Nexedi and used for the responsive ERP5 interface and as basis for applications in app stores like OfficeJS.

© Nexedi SA

2. What is a jIO Storage?

A storage is either a connector or a handler storage. The first one stores documents (metadata) and attachments (content) and provdies access to the documents through the jIO API. Setting up a storage connector is easy:

```
//create a jIO localStorage
jIO.createJIO({
  "type": "local",
  "sessiononly": false
});
```

just as adding a handler storage such as encryption on top:

```
//create a jIO encrypted localStorage
jIO.createJIO({
  "type": "crypt",
  "key": "json-web-key",
  "sub_storage": {
    "type": "local",
    "sessiononly": false
  }
});
```

3. Promises

jIO is fully asynchronous and use promises provided by a library called RSVP (original version, renderJS custom version below!). The main difference to the official Promise spec and RSVP is that jIO is not using `.then` for chaining. Instead chains are written using `RSVP.Queue()` with `.push(function () {...}).push(function (result) {...}).push(undefined, function (error) {...});` as custom extension which allows RenderJS promises to be cancelled.

 © Nexedi SA

4. Another JavaScript Framework? Why use jIO?

Nexedi's free software products and custom solutions developed from them are normally running for many years. As complexity of apps is usually very high, redevelopments to follow the current trending JS-framework or having to replace a framework being discontinued is out of our scope. Hence jIO (and renderJS), two non-frills libraries, that are:

- **sturdy**, small API, easy to use once understood.
- **flexible**, multiple storages and handlers.
- **extendable**, write your own storage if needed.

 © Nexedi SA

5. Getting Started

jIO is quick to setup and get working.

5.1. Source Code

The jIO source code is available on Gitlab (Github Mirror). To **build**,

```
> git clone https://lab.nexedi.com/nexedi/jio.git
> npm install
> grunt server
```

or just **download** the files directly:

- RenderJS latest version
- RenderJS latest version (minified)
- RSVP (custom version) (amd version)
- RSVP (custom version/minified)

The following file(s) might also be useful:

- renderJS latest version
- renderJS latest version (minified)

5.2. What is a Document?

A document is an association of metadata and attachment(s). The metadata is the set of properties of the document and the attachments are binary objects that represent the content of

 © Nexedi SA

the document. In jIO, the metadata is a dictionary with keys and values (a JSON object), and attachments are simple strings, for example:

```
{
  // document metadata
  title: 'A Title!',
  creator: 'Mr.Author'
}
```

5.3. Hello World

Create an html file with the specified js files and the following contents:

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="rsvp.latest.js"></script>
    <script type="text/javascript" src="jio.latest.js"></script>
  </head>
  <body>
    <p>jio example, see console</p>
    <script type="text/javascript">
      (function (jIO) {
        var storage = jIO.createJIO({"type":"local", "sessionstorage": false});
        console.log(storage);
        function foo(message) {
          return new RSVP.Queue()
            .push(function () {
              return storage.putAttachment("/", "start", new Blob([message], {type: "text/plain"}));
            })
            .push(function (my_answer) {
              console.log(my_answer);
              return storage.getAttachment("/", "start", {"format": "text"});
            })
            .push(function (my_answer) {
              console.log(my_answer)
            })
            .push(undefined, function (error) {
              console.log(error);
            });
        }
        return foo("hello");
      }(jIO));
    </script>
  </body>
</html>
```

Check your console to see how the jIO storage is created, the attachment is stored and fetched again. **Note**, that browser *localStorage* is treated as document and the actual documents as attachments and that attachments are always stored in blob format.

5.4. Posting Attachments

 © Nexedi SA

Below is an example of posting an attachment into indexeddb to show the handling of attachments.

```
(function (jIO) {
  // create a new jIO
  var jio_instance = jIO.createJIO({type: 'indexeddb', database: 'foo'});

  // post the document 'myVideo'
  return jio_instance.put('metadata', {
    title: 'My Video',
    type: 'MovingImage',
    format: 'video/ogg',
    description: 'Images Compilation'
  })
  // post a thumbnail attachment
  .push(function () {
    return jio_instance.putAttachment(
    'metadatda',
    'thumbnail',
    new Blob([my_image], {type: 'image/jpeg'})
  })
  // post video attachment
  .push(function () {
    return jio_instance.putAttachment(
    'metadatda',
    'video',
    new Blob([my_video], {type: 'video/ogg'})
  })
  // catch any errors and throw
  .push(undefined, function(err) {
    console.log(err);
    throw err
  });
}(jIO));
```


Company Confidential

© Nexedi SA

6. API - Quickguide

The original jIO interface was based on couchDB but has evolved to the current set of methods described below and in more detail afterwards.

Method	Example	Info
Create Storage	jIO.createJIO({storage_configuration});	[returns nothing]. Initialize a new storage or storage tree.
Post Document	storage.post({ "property": "dict" });	[returns Promise]. Create new document. Id is generated automatically.
Put Document	storage.put(id, { "property": "dict" });	[returns Promise]. Create/Update a document with predefined id.
Get Document	storage.get(id); // {"property": "dict"}	[returns Promise]. Retrieve a document.
Remove	storage.remove(id);	[returns Promise]. Deletes a document and its attachments.

© Nexedi SA

Method	Example	Info

Get all Documents	storage.allDocs({
	"query": [query-object],
	"limit": [[Integer], [Integer]],
	"sort_on": [["key1", "ascending"], ["key2", "descending"]],
	"select_list": ["key1", "key2", "key3"],
	"wildcard": "%",
	"include_docs": [Boolean],
	});
	
	// include_docs response
	// {
	// "total_rows": [n],
	// "rows": [{
	// "id": [id],
	// "value": {},
	// "doc": {"property": "key"}
	// }, {}...]
	//}
	
	// default response
	//{
	// "total_rows": [n],
	// "rows": [{
	// "id": [id],
	// "value": {"select_list_key": "select_list_value"}
	// }, {}, ...]
	//}
		[returns Promise]. Retrieves a list of all documents. If supported *queried*, *limit* to certain records, *select_list* to certain properties, or *sort_on* by specific keys.

Add Attachment to Document	storage.putAttachment(id, name, blob);	[returns Promise]. Updates/adds an blob attachment to a document
Remove Attachment	storage.removeAttachment(id, name);	[returns Promise]. Deletes a document's attachment

Get Attachment	storage.getAttachment(id, name, {
	"format": [format]
	});
	
	// response in format
		[returns Promise]. Retrieve a document attachment as blob, data_url, array_buffer, text or json.

Synchronize Storages	storage.repair();	[returns Promise]. Synchronize/repair storages

7. API - Storage Types

Below is a list of storages currently supported by jIO including the respective storage configurations. **Note** that authentication for a storage is not handled by jIO. You have to provide whatever tokens are required to access the storage when creating your jIO storage.

7.1. LocalStorage

This storage has only one document, so **post**, **put**, **remove** and **get** methods are not supported.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "local").
sessiononly	no	Boolean	False (default): create a storage with unlimited duration. True: the storage duration is limited to the user session.

Example:

```
var jio = jIO.createJIO({
  type: "local",
  sessiononly: true
});
```

7.2. MemoryStorage

Stores the data in a Javascript object, in memory. The storage's data isn't saved when your web

page is closed or reloaded. The storage doesn't take any argument at creation.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "memory").

Example:

```
var jio = jIO.createJIO({
  type: "memory"
});
```

7.3. IndexedDB

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "indexeddb").
database	yes	String	Name of the database.

Example:

```
{
  "type": "indexeddb",
  "database": "mydb"
}
```

7.4. WebSQL

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "websql").
database	yes	String	Name of the database.

Example:

```
{
  "type": "websql",
  "database": "mydb"
}
```

7.5. DavStorage

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "dav").
url	yes	String	Url of your webdav server.
basic_login	yes	String	Login and password of your dav, base64 encoded like this: btoa(username + ":" + password)
with_credentials	no	Boolean	True: send domain cookie. False (default): do not send domain cookie.

Example:

```
// No authentication
{
  "type": "dav",
  "url":  url
}

// Basic authentication
{
  "type":       "dav",
  "url":        url,
  "basic_login": btoa(username + ":" + password)
}

// Digest authentication is not implemented
```

Be careful: The generated description never contains a readable password, but for basic authentication, the password is just base64 encoded.

7.6. Dropbox

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "dropbox").
access_token	yes	String	Access token for your dropbox. See dropbox documentation how to generate an access_token.
root	no	String	"dropbox" (default) for full access to account files. "sandbox" for app limited file access.

Example:

{
 "type": "dropbox",
 "access_token": "sample_token"
 "root": "dropbox"
}

7.7. Google Drive

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "gdrive").
access_token	yes	String	Access token for your dropbox. See dropbox documentation how to generate an access_token.
trashing	no	Boolean	true (default): sends file to trash bin when calling "remove". false: delete files permantently when calling "remove""

Example:

{
 "type": "gdrive",
 "access_token": "sample_token"
 "trashing": true
}

7.8. ERP5 Storage

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "erp5").
url	yes	String	Url of your erp5 account.
default_view_reference	no	String	Reference of the action used for the delivering of the document.

Example:

```
{
  "type": "erp5",
  "url": erp5_url
}
```

7.9. ZipStorage (Handler)

This handler compresses and decompresses files (attachments only) to reduce network and storage usage.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "zip").
sub_storage	yes	Object	Definition of storage whose attachments should be zipped.

Example:

```
{
  "type": "zip",
  "sub_storage": {storage_definition}
}
```

7.10. ShaStorage (handler)

This handler provides a post method that creates a document that has for name the SHA-1 hash of his parameters.

© Nexedi SA

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "sha").
sub_storage	yes	Object	Definition of storage whose contents should be stored as SHA-1 hashes.

Example:

```
{
  "type": "sha",
  "sub_storage": {storage_definition}
}
```

7.11. UuidStorage (handler)

This handler provides a post method to create a document that has a unique ID for name.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "uuid").
sub_storage	yes	Object	Definition of storage whose post method should created UUID-ids.

Example:

```
{
  "type": "uuid",
  "sub_storage": {storage_definition}
}
```

7.12. QueryStorage (handler)

This handler provides an allDocs method with queries support to the substorage.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "query").
sub_storage	yes	Object	Definition of storage whose contents should be query-able on *allDocs* calls.

Example:

```
{
  "type": "query",
  "sub_storage": {storage_definition}
}
```

7.13. CryptStorage (handler)

This handler encrypts and decrypts attachments before storing them. You need to generate a Crypto key in JSON format to use the handler. (see here for more informations)

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "crypt").
key	yes	String	JSON crypto key.
sub_storage	yes	Object	Definition of storage whose contents should be encrypted.

Example:

```
var key, jsonKey, jio;

crypto.subtle.generateKey({name: "AES-GCM",length: 256},
                (true), ["encrypt", "decrypt"])
.then(function(res){key = res;});
window.crypto.subtle.exportKey("jwk", key)
.then(function(res){jsonKey = res})

jio = jIO.createJIO({
  "type": "crypt",
  "key": json_key
  "sub_storage": {storage_definition}
}
```

 © Nexedi SA

7.14. UnionStorage (handler)

This handler takes as list of storages as argument. When using a jio method, UnionStorage tries it on the first storage of the array, and, in case of failure, tries with the next storage, and repeats the operation until success, or end of storage's array.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "union").
storage_list	yes	Array	List of storage definitions.

Example:

```
{
 "type": "union",
 "storage_list": [
 {storage_definition},
 {storage_definition},
 ...
 {storage_definition}
]}
```

7.15. FileSystemBridgeStorage (handler)

This handler adds an abstraction level on top of the webDav Jio storage, ensuring that each document has only one attachment, and limiting the storage to one repertory.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "drivetojiomapping").
sub_storage	yes	Object	Definition of storage whose contents should be query-able on *allDocs* calls.

Example:

```
{
 "type": "drivetojiomapping",
 "sub_storage":
}
```

 © Nexedi SA

7.16. Document Storage (handler)

This handler creates a storage from a document in a storage, by filling his attachments with a new jIO storage.

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "document").
document_id	yes	String	id of the document to use.
repair_attachment	no	Boolean	Verify if the document is in good state. (default to false)

Example:

```
{
  "type": "document",
  "document_id": id,
  "repair_attachment": false
}
```

7.17. Replicate Storage (Handler)

Replicate Storage synchronizes documents between a local and a remote storage.

© Nexedi SA

Parameter	Required?	Type	Description
type	yes	String	Name of the storage type (here: "replicate").
local_sub_storage	yes	Object	Local sub_storage description.
remote_sub_storage	yes/td>	Object	Remote sub_storage description.
query_options	no/td>	Object	Query object to limit the synchronisation to specific files.
use_remote_post	no/td>	Boolean	true: at file modification, modifies the local file id. false (default): at file modification, modifies the remote file id.
conflict_handling	no/td>	Number	0 (default): no conflict resolution (throws error) 1: keep the local state. 2: keep the remote state. 3: keep both states (no signature update)
check_local_modification	no/td>	Boolean	Synchronise when local files are modified.
check_local_creation	no/td>	Boolean	Synchronise when local files are created.
check_local_deletion	no/td>	Boolean	Synchronise when local files are deleted.
check_remote_modification	no/td>	Boolean	Synchronise when remote files are modified.
check_remote_creation	no/td>	Boolean	Synchronise when local files are created.
check_remote_deletion	no/td>	Boolean	Synchronise when local files are deleted.

By default, synchronization parameters are set to true.

Example:

```
{
  type: 'replicate',
  local_sub_storage: { 'type': 'local'}
  remote_sub_storage: {
    'type': 'dav',
    'url': 'http://mydav.com',
    'basic_login': 'aGFwcHkgZWFzdGVy'
  }
  use_remote_post: false,
  conflict_handling: 2,
  check_local_creation: false,
  check_remote_deletion: false
}
```

 © Nexedi SA

8. jIO Query Engine

In jIO, a query can ask a storage server to select, filter, sort, or limit a document list before sending it back. If the server is not able to do so, the jio query tool can do the filtering by itself on the client. Only the .allDocs() method can use jio queries.

A query can either be a string (using a specific language useful for writing queries), or it can be a tree of objects (useful to browse queries). To handle queries, jIO uses a parsed grammar file which is compiled using JISON.

JIO queries can be used like database queries, for tasks such as:

- search a specific document
- sort a list of documents in a certain order
- avoid retrieving a list of ten thousand documents
- limit the list to show only N documents per page

For some storages (like localStorage), jio queries can be a powerful tool to query accessible documents. When querying documents on a distant storage, some server-side logic should be run to avoid returning too many documents to the client.

8.1. How to use Queries with jIO?

Queries can be triggered by including the option named query in the .allDocs() method call.

```
var options = {};

// search text query
options.query = '(creator:"John Doe") AND (format:"pdf")';

// OR query tree
options.query = {
```

```
  type: 'complex',
  operator: 'AND',
  query_list: [{
    type: 'simple',
    key: 'creator',
    value: 'John Doe'
  }, {
    type: 'simple',
    key: 'format',
    value: 'pdf'
  }]
};

// FULL example using filtering criteria
options = {
  query: '(creator:"% Doe") AND (format:"pdf")',
  limit: [0, 100],
  sort_on: [
    ['last_modified', 'descending'],
    ['creation_date', 'descending']
  ],
  select_list: ['title']
};

// execution
jio_instance.allDocs(options, callback);
```

9. Creating Your Own Storage

Extending jIO by adding own storages is fairly easy as you only have to implement the base methods plus the internal methods *hasCapacity* (for which *allDocs* parameters are supported) and *buildQuery* (for constructing actual queries).

For example if you would want to create a parallel storage which allows to maintain multiple storages on the same jIO gadget you could create a file named **jio.parallelstorage.js** (or similar) and add it after the jio file in your html. The file should contain:

```
/**
 * JIO Parallel Storage Type = "Parallel".
 * keep storages in parallel, without sync/replication
 */
/*jslint indent: 2 */
/*global jIO, RSVP, Array, Number*/
(function (jIO, RSVP, Array, Number) {
  "use strict";

  function testInteger(candidate) {
    if (Number.isInteger(candidate)) {
      return candidate;
    }
  }

  function handleArguments(argument_list) {
    if (testInteger(argument_list[0])) {
      return argument_list.splice(1);
    }
    return argument_list;
  }
```

The parallel storage only has a simple functionality which is running a jIO commands on one storage from a list of sub_storages denoted by an index number (default 0 = first storage in the list). The above methods handling picking the first parameter from the arguments passed to every jio method. So when getting an attachment from a parallelStorage, you would:

 © Nexedi SA

storage.getAttachment(2, id, name);

meaning running this method on the 3rd storage in a list of storages.

```
/**
 * The JIO Parallel Storage extension
 *
 * @class ParallelStorage
 * @constructor
 */
 function ParallelStorage (spec) {
  var i;

  if (spec.storage_list === undefined || !Array.isArray(spec.storage_list)) {
   throw new jIO.util.jIOError("storage_list is not an Array", 400);
  }

  this._storage_list = [];
  this._getStorage = function (index) {
   return this._storage_list[testInteger(index) || 0];
  };

  for (i = 0; i < spec.storage_list.length; i += 1) {
   this._storage_list.push(jIO.createJIO(spec.storage_list[i]));
  }
 }
```

Every storage needs a class constructor which sets up the storage. In this case validate the parameters passsed in the configuration and calling *createJIO* with the configurations passed in the *storage_list* parameter. `Note` this constructor does not return a promise. It's a synchronous call.

```
 ParallelStorage.prototype.post = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.post.apply(storage, handleArguments(arguments));
 };

 ParallelStorage.prototype.put = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.put.apply(storage, handleArguments(arguments));
 };

 ParallelStorage.prototype.get = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.get.apply(storage, handleArguments(arguments));
 };

 ParallelStorage.prototype.remove = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.remove.apply(storage, handleArguments(arguments));
 };

 ParallelStorage.prototype.allAttachments = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.allAttachments.apply(storage, handleArguments(arguments));
 };

 ParallelStorage.prototype.removeAttachment = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.removeAttachment.apply(storage, handleArguments(arguments));
 };
```

 © Nexedi SA

```
ParallelStorage.prototype.putAttachment = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.putAttachment.apply(storage, handleArguments(arguments));
};

ParallelStorage.prototype.getAttachment = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.getAttachment.apply(storage, handleArguments(arguments));
};

ParallelStorage.prototype.hasCapacity = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.hasCapacity.apply(storage, handleArguments(arguments));
};

ParallelStorage.prototype.allDocs = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.allDocs.apply(storage, handleArguments(arguments));
};

ParallelStorage.prototype.buildQuery = function () {
  var storage = this._getStorage(arguments[0]);
  return storage.buildQuery.apply(storage, handleArguments(arguments));
};
```

Afterwards all jIO methods that should be supported must be implemented on the class. **Note** how the first argument is picked on all calls to denote the storage to select.

```
  jIO.addStorage('parallel', ParallelStorage);

}(jIO, RSVP, Array, Number));
```

The storage closes with adding the storage to the jIO object. After this it is available like all other storages.

 © Nexedi SA

10. Tips and Tricks

10.1. CreateJIO is not Async

When creating new storages make sure you don't pass the *createJIO* call as a return value of a RSVP Promise, because the creating a new storage is not asynchronous, so the promise will resolve with an undefined return value instead of the storage.

10.2. Using Queries Outside jIO

Basic example:

```
// object list (generated from documents in storage or index)
var object_list = [
  {"title": "Document number 1", "creator": "John Doe"},
  {"title": "Document number 2", "creator": "James Bond"}
];

// the query to run
var query = 'title: "Document number 1"';

// running the query
var result = jIO.QueryFactory.create(query).exec(object_list);
// console.log(result);
// [ { "title": "Document number 1", "creator": "John Doe"} ]
```

Other example:

```
var result = jIO.QueryFactory.create(query).exec(
  object_list,
  {
    "select": ['title', 'year'],
    "limit": [20, 20], // from 20th to 40th document
    "sort_on": [['title', 'ascending'], ['year', 'descending']],
    "other_keys_and_values": "are_ignored"
  }
);
// this case is equal to:
```

```
var result = jIO.QueryFactory.
  create(query).exec(object_list);
jIO.Query.sortOn([
  ['title', 'ascending'],
  ['year', 'descending']
], result);
jIO.Query.limit([20, 20], result);
jIO.Query.select(['title', 'year'], result);
```

10.3. Wildchard Query Parameter

Queries select items which exactly match the value given in the query but you can also use wildcards (%). If you don’t want to use a wildcard, just set the operator to =.

```
var option = {
  query: 'creator:"% Doe"' // use wildcard
};

var option = {
  query: 'creator:="25%"' // don't use wildcard
};
```

10.4. JiO Query JSON Schemas and Grammar

Below you can find schemas for constructing queries.

Complex Query JSON Schema:

```
{
  "id": "ComplexQuery",
  "properties": {
    "type": {
      "type": "string",
      "format": "complex",
      "default": "complex",
      "description": "Type is used to recognize the query type"
    },
    "operator": {
      "type": "string",
      "format": "(AND|OR|NOT)",
      "required": true,
      "description": "Can be 'AND', 'OR' or 'NOT'."
    },
    "query_list": {
      "type": "array",
      "items": {
        "type": "object"
      },
      "required": true,
      "default": [],
      "description": "query_list is a list of queries which " +
              "can be in serialized format " +
              "or in object format."
```

```
   }
  }
}
```

Simple Query JSON Schema:

```
{
  "id": "SimpleQuery",
  "properties": {
    "type": {
      "type": "string",
      "format": "simple",
      "default": "simple",
      "description": "Type is used to recognize the query type."
    },
    "operator": {
      "type": "string",
      "default": "",
      "format": "(>=?|<=?|!?=|)",
      "description": "The operator used to compare."
    },
    "id": {
      "type": "string",
      "default": "",
      "description": "The column id."
    },
    "value": {
      "type": "string",
      "default": "",
      "description": "The value we want to search."
    }
  }
}
```

JIO Query Grammar:

```
search_text
    : and_expression
    | and_expression search_text
    | and_expression OR search_text

and_expression
    : boolean_expression
    | boolean_expression AND and_expression

boolean_expression
    : NOT expression
    | expression

expression
    : ( search_text )
    | COLUMN expression
    | value

value
    : OPERATOR string
    | string

string
```

```
  : WORD
  | STRING

terminal:
  OR           -> /OR[ ]/
  AND          -> /AND[ ]/
  NOT          -> /NOT[ ]/
  COLUMN       -> /[^> /"(\\.|[^\\"])*"/
  WORD         -> /[^> /(>=?|<=?|!?=)/
  LEFT_PARENTHESE  -> /\(/
  RIGHT_PARENTHESE -> /\)/

ignore: " "
```

10.5. Customizing jIO Query Search Keys

Features like case insensitive, accent-removing, full-text searches and more can be implemented by customizing jIO's query behavior.

Let's start with a simple search:

```
var query = {
  type: 'simple',
  key: 'someproperty',
  value: comparison_value,
  operator: '='
}
```

Each of the .someproperty attribute in objects' metadata is compared with comparison_value through a function defined by the '=' operator.

You can provide your own function to be used as '=' operator:

```
var strictEqual = function (object_value, comparison_value) {
  return comparison_value === object_value;
};

var query = {
  type: 'simple',
  key: {
    read_from: 'someproperty',
    equal_match: strictEqual
  },
  value: comparison_value
}
```

Inside equal_match, you can decide to interpret the wildcard character % or just ignore it, as in this case.

If you need to convert or preprocess the values before comparison, you can provide a conversion function:

 © Nexedi SA

```
var numberType = function (obj) {
  return parseFloat('3.14');
};

var query = {
  type: 'simple',
  key: {
    read_from: 'someproperty',
    cast_to: numberType
  },
  value: comparison_value
}
```

In this case, the operator is still the default '=' that works with strings. You can combine cast_to and equal_match:

```
var query = {
  type: 'simple',
  key: {
    read_from: 'someproperty',
    cast_to: numberType,
    equal_match: strictEqual
  },
  value: comparison_value
}
```

Now the query returns all objects for which the following is true:

```
strictEqual(numberType(metadata.someproperty),
      numberType(comparison_value))
```

For a more useful example, the following function removes the accents from any string:

```
var accentFold = function (s) {
  var map = [
    [new RegExp('[àáâãäå]', 'gi'), 'a'],
    [new RegExp('æ', 'gi'), 'ae'],
    [new RegExp('ç', 'gi'), 'c'],
    [new RegExp('[èéêë]', 'gi'), 'e'],
    [new RegExp('[ìíîï]', 'gi'), 'i'],
    [new RegExp('ñ', 'gi'), 'n'],
    [new RegExp('[òóôõö]', 'gi'), 'o'],
    [new RegExp('œ', 'gi'), 'oe'],
    [new RegExp('[ùúûü]', 'gi'), 'u'],
    [new RegExp('[ýÿ]', 'gi'), 'y']
  ];

  map.forEach(function (o) {
    var rep = function (match) {
      if (match.toUpperCase() === match) {
        return o[1].toUpperCase();
      }
      return o[1];
    };
    s = s.replace(o[0], rep);
  });
  return s;
```

```
};

...
 cast_to: accentFold
...
```

A more robust solution to manage diacritics is recommended for production environments, with unicode normalization, like unorm (untested).

10.6. Overriding jIO Query Operators and Sorting

The advantage of providing an equal_match function is that it can work with basic types; you can keep the values as strings or, if you use a cast_to function, it can return strings, numbers, arrays... and that's fine if all you need is the '=' operator.

It's also possible to customize the behavior of the other operators: <, >, !=...

To do that, the object returned by cast_to must contain a .cmp property, that behaves like the compareFunction described in Array.prototype.sort():

```
function myType (...) {
 ...
 return {
  ...
  'cmp': function (b) {
   if (a < b) {
    return -1;
   }
   if (a > b) {
    return +1;
   }
   return 0;
  }
 };
}

...
 cast_to: myType
...
```

If the < or > comparison makes no sense for the objects, the function should return undefined.

The .cmp() property is also used, if present, by the sorting feature of queries.

10.7. Partial Date/Time Matching in jIO Queries

As a real life example, consider a list of documents that have a start_task property.

 © Nexedi SA

The value of start_task can be an ISO 8601 string with date and time information including fractions of a second. Which is, honestly, a bit too much for most queries.

By using a cast_to function with custom operators, it is possible to perform queries like "start_task > 2010-06", or "start_task != 2011". Partial time can be used as well, so we can ask for projects started after noon of a given day: start_task = "2011-04-05" AND start_task > "2011-04-05 12"

The JIODate type has been implemented on top of the Moment.js library, which has a rich API with support for multiple languages and timezones. No special support for timezones is present (yet) in JIODate.

To use JIODate, include the jiodate.js and moment.js files in your application, then set cast_to = jiodate.JIODate.

10.8. jIO Query Key Schemas

Instead of providing the key object for each attribute you want to filter, you can group all of them in a schema object for reuse:

```
var key_schema = {
 key_set: {
  date_day: {
   read_from: 'date',
   cast_to: 'dateType',
   equal_match: 'sameDay'
  },
  date_month: {
   read_from: 'date',
   cast_to: 'dateType',
   equal_match: 'sameMonth'
  }
 },
 cast_lookup: {
  dateType: function (str) {
   return new Date(str);
  }
 },
 match_lookup: {
  sameDay: function (a, b) {
   return (
    (a.getFullYear() === b.getFullYear()) &&
     (a.getMonth() === b.getMonth()) &&
      (a.getDate() === b.getDate())
   );
  },
  sameMonth: function (a, b) {
   return (
```

 © Nexedi SA

```
        (a.getFullYear() === b.getFullYear()) &&
          (a.getMonth() === b.getMonth())
      );
    }
  }
}
```

With this schema, we have created two 'virtual' metadata attributes, date_day and date_month. When queried, they match values that happen to be in the same day, ignoring the time, or the same month, ignoring both time and day.

A key_schema object can have three properties:

- key_set - required.
- cast_lookup - optional, an object of the form {name: function} that is used if cast_to is a string. If cast_lookup is not provided, then cast_to must be a function.
- match_lookup - optional, an object of the form {name: function} that is used if equal_match is a string. If match_lookup is not provided, then equal_match must be a function.

10.9. Using a schema

A schema can be used:

- In a query constructor. The same schema will be applied to all the sub-queries:

```
jIO.QueryFactory.create({...}, key_schema).exec(...);
```

- In the jIO.createJIO() method. The same schema will be used by all the queries created with the .allDocs() method:

```
var jio = jIO.createJIO({
  type: 'local',
  username: '...',
  application_name: '...',
  key_schema: key_schema
});
```

 © Nexedi SA

11. Tests

You can run tests after installing and building jIO by opening the */test/* folder.

© Nexedi SA

12. FAQ

Q: What browsers does jIO support?

A: jIO will work on fully html5 compliant browsers. Thus, jIO should work well with the latest version of Chrome and Firefox. IE is a stretch and Safari as well. Run the tests tto find out if your browser is supported.

 © Nexedi SA

13. Licence

14. Examples

Most of the front end solutions created by Nexedi are based on RenderJS and jIO. For ideas and inspiration check out the following examples:

- OfficeJS - Office Productivity App Store (Chat client, task managers, various editors).

 © Nexedi SA