jIO Documentation

Release 2.0.0-wip

Nexedi

Jun 12, 2018

Contents

jIO is a JavaScript library that allows to manage JSON documents on local or remote storages in asynchronous fashion. jIO is an abstracted API mapped after CouchDB, that offers connectors to multiple storages, special handlers to enhance functionality (replication, revisions, indexing) and a query module to retrieve documents and specific information across storage trees.

CHAPTER 1

How does it work?

jIO is composed of two parts - jIO core and storage libraries. The core makes use of storage libraries (connectors) to interact with the associated remote storage servers. Some queries can be used on top of the `.allDocs()` method to query documents based on defined criteria.

jIO uses a job management system, so each method call adds a job into a queue. The queue is copied in the browser's local storage (by default), so it can be restored in case of browser crash. Jobs are invoked asynchronously and ongoing jobs are not able to re-trigger to prevent conflicts.

CHAPTER 2

Copyright and license

jIO is open source and is licensed under the LGPL license.

CHAPTER 3

jIO documentation

3.1 Getting started

1. *Download* the core jIO, the storages you need and the dependencies required for them.
2. Add the scripts to your HTML page in the following order:

   ```
   <!-- jio core + dependencies -->
   <script src="rsvp.js"></script>
   <script src="jio.js"></script>

   <script ...>
   ```

3. jIO connects to a number of storages and allows adding handlers (or functions) to specific storages. You can use both handlers and available storages to build a storage tree across which all documents will be maintained and managed by jIO.

   ```
   // create your jio instance
   var my_jio = jIO.createJIO(storage_description);
   ```

   You have to provide a `storage_description` object, providing location and credentials.

   Its format depends on the type of storage, see *List of Available Storages*.

4. The jIO API provides ten main methods to manage documents across the storage(s) specified in your jIO storage tree.

   For details on the `document` and `attachment` objects, see *What is a document?*

Method	Example
`.post()`	`my_jio.post(document, [options]);` Creates a new document
`.put()`	`my_jio.put(document, [options]);` Creates/Updates a document
`.putAttachment()`	`my_jio.putAttachement(attachment, [options]);`` Updates/Adds an attachment to a document
`.get()`	`my_jio.get(document, [options]);` Reads a document
`.getAttachment()`	`my_jio.getAttachment(attachment, [options]);` Reads a document attachment
`.remove()`	`my_jio.remove(document, [options]);` Deletes a document and its attachments
`.removeAttachment()`	`my_jio.removeAttachment(attachment, [options]);` Deletes a document's attachment
`.allDocs()`	`my_jio.allDocs([options]);` Retrieves a list of existing documents
`.repair()`	`my_jio.repair(document, [options]);` Repairs the document

3.1.1 Download & Fork

You can get latest jIO release by using on of those links:

Full download Minified download

You can get latest RSVP release by using on of those links:

Full download Minified download

Fork jIO

Feel free to use the Gitlab repository:
GitLab: `git clone https://lab.nexedi.com/nexedi/jio.git`

3.2 How to manage documents?

jIO is mapped after the CouchDB APIs and extends them to provide unified, scalable and high performance access via JavaScript to a wide variety of storage backends.

If you are not familiar with Apache CouchDB: it is a scalable, fault-tolerant, and schema-free document-oriented database. It is used in large and small organizations for a variety of applications where traditional SQL databases are not the best solution for the problem at hand. CouchDB provides a RESTful HTTP/JSON API accessible by many programming libraries and tools (like curl or Pouchdb) and has its own conflict management system.

3.2.1 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 (or text) objects that represent the content of the document.

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

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

Here is a draft about metadata to use with jIO.

3.2.2 Basic Methods

Below you can see examples of the main jIO methods.

```
// Create a new jIO instance
var jio_instance = jIO.createJIO(storage_description);

// create and store new document
jio_instance.post({title: 'my document'}).
  then(function (response) {
    // console.log(response);
  });
```

(continues on next page)

(continued from previous page)

```
// create or update an existing document
jio_instance.put('document_name', {title: 'another document'}).
  then(function (response) {
    // console.log(response);
  });

// add an attachment to a document
jio_instance.putAttachment('document_name',
                           'attachment_name',
                           new Blob([data], {'type' : data_mimetype});
  ).
  then(function (response) {
    // console.log(response);
  });

// read a document
jio_instance.get('document_name').
  then(function (response) {
    // console.log(response);
  });

// read an attachment
jio_instance.getAttachment('document_name',
                           'attachment_name').
  then(function (response) {
    // console.log(response);
  });

// delete a document and its attachment(s)
jio_instance.remove('document_name').
  then(function (response) {
    // console.log(response);
  });

// delete an attachment
jio_instance.removeAttachment('document_name',
                              'attachment_name').
  then(function (response) {
    // console.log(response);
  });

// get all documents
jio_instance.allDocs().then(function (response) {
  // console.log(response);
});
```

3.2.3 Promises

Each jIO method (with the exception of `.createJIO()`) returns a Promise object, which allows us to get responses into callback parameters and to chain callbacks with other returned values.

jIO uses a custom version of RSVP.js, adding canceler and progression features.

You can read more about promises:

- RSVP.js on GitHub

- Promises/A+
- CommonJS Promises

3.2.4 Method Options and Callback Responses

To retrieve jIO responses, you have to provide callbacks like this:

```
jio_instance.post(metadata, [options]).
    then([responseCallback], [errorCallback], [progressionCallback]);
```

- On command success, `responseCallback` is called with the jIO response as first parameter.
- On command error, `errorCallback` is called with the jIO error as first parameter.
- On command notification, `progressionCallback` is called with the storage notification.

Here is a list of responses returned by jIO according to methods and options:

Available for	Option	Response (Callback first parameter)
`.post()`, `.put()`, `.remove()`	Any	id of the document affected (string)
`.putAttachment()`, `.removeAttachment()`	Any	no specific value
`.get()`	Any	document_metadata (object)

`.getAttachment()`	Any	`new Blob([data], {"type":␣`
		`↪content_type})`

`.allDocs()`	No option	`{`
		`    total_rows: 1,`
		`    rows: [{`
		`      id: 'mydoc',`
		`      value: {},`
		`    }]`
		`  }`

`.allDocs()`	include_docs: true	`{`
		`    total_rows: 1,`
		`    rows: [{`
		`      id: 'mydoc',`
		`      value: {`
		`        // Here, 'mydoc'␣`
		`↪metadata`
		`      }`
		`    }]`
		`  }`

In case of error, the `errorCallback` first parameter looks like:

```
{
  status_code: 404,
  message: 'Unable to get the requested document'
}
```

3.2.5 How to store binary data

The following example creates a new jIO in localStorage and then posts a document with two attachments.

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

// post the document 'myVideo'
jio_instance.put( 'metadata', {
  title       : 'My Video',
  type        : 'MovingImage',
  format      : 'video/ogg',
  description : 'Images Compilation'
})
.push(undefined, function(err) {
    // pushes error handler with RSVP.Queue push method
    // nothing to do with JIO itself
    return alert('Error posting the document metadata');
  });

  // post a thumbnail attachment
jio_instance.putAttachment('metadata',
  'thumbnail',
  new Blob([my_image], {type: 'image/jpeg'})
  ).push(undefined, function(err) {
  return alert('Error attaching thumbnail');
  });

  // post video attachment
  jio_instance.putAttachment('metadata',
    'video',
    new Blob([my_video], {type: 'video/ogg'})
  ).push(undefined, function(err) {
    return alert('Error attaching video');
  });
  alert('Video Stored');
```

indexedDB Storage now contains:

```
{
  "/myVideo/": {
    "title": "My Video",
    "type": "MovingImage",
    "format": "video/ogg",
    "description": "Images Compilation",
    "_attachments":{
      "thumbnail":{
        "digest": "md5-3ue...",
        "content_type": "image/jpeg",
        "length": 17863
      },
      "video":{
        "digest": "md5-0oe...",
        "content_type": "video/ogg",
        "length": 2840824
      }
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "/myVideo/thumbnail": "...",
  "/myVideo/video": "..."
}
```

3.3 Replicate Storage: Conflicts and Resolution

3.3.1 Why Conflicts can Occur

Using jIO you can store documents in multiple locations. With an increasing number of users working on a document and some storages not being available or responding too slow, conflicts are more likely to occur. jIO defines a conflict as multiple versions of a document existing in a storage tree and a user trying to save on a version that does not match the latest version of the document.

To keep track of document versions a replicate storage must be used. When doing so, jIO creates a document tree file for every document. This file contains all existing versions and their status and is modified whenever a version is added/updated/removed or when storages are being synchronized.

3.3.2 How conflicts are handled

The RemoteStorage takes in parameter two substorages, one "local" and one "remote". The "local" storage can be remote, but it will be used for all the requests like **get()**, **getAttachment()**, **allDocs()**...

Using the document tree, jIO tries to make every version of a document available on the two storages. When multiple versions of a document exist, Jio will follow the rule set by the conflict_handling option, given at storage creation. This option can one of the following numbers:

- 0: no conflict resolution (throws an error when conflict is occuring)
- 1: keep the local state. (overwrites the remote document with local content)
- 2: keep the remote state. (overwrites the local document with remote content)
- 3: keep both copies (leave documents untouched, no signature update)

3.3.3 Simple Conflict Example

You are keeping a namecard file on your PC updating from your smartphone. Your smartphone ran out of battery and is offline when you update your namecard on your PC with your new email adress. Someone else changes this email from your PC and once your smartphone is recharged, you go back online and the previous update is executed.

1. Set up the replicate storage:

```
var jio_instance = jIO.createJIO({
  // replicate storage
  type: 'replicate',
  local_sub_storage : {
      type: 'local',
      ...
    }
    remote_sub_storage: {
      type: 'dav',
      ...
    }
```

(continues on next page)

(continued from previous page)

```
    conflict_handling: ...
});
```

1. (a) Create the namecard on your smartphone:

```
jio_instance.put('myNameCard', {
  email: 'jb@td.com'
 }).then(function (response) {
  // response -> 'myNameCard'
});
```

   This will create the document on your WebDAV and local storage

2. (b) Someone else updates your shared namecard on WebDAV:

```
jio_instance.put('myNameCard', {
  email: 'kyle@td.com',
}).then(function (response) {
  // response -> 'myNameCard'
});
```

   Your smartphone is offline, so now you will have one version on your smartphone and another version on WebDAV on your PC.

3. (c) Later, your smartphone is online and you modify your email:

```
jio_instance.get('myNameCard').then(function (response) {
  // response.email -> 'jb@td.com'
  // the get() method checks only on your local storage
  // and doesn't warn you about remote modifications.

  return jio_instance.put('myNameCard', {
    email: 'jack@td.com'
  })
 .then(function (response) {
  // response -> 'myNameCard'
});
```

Your latest modification of the email is: "jack@td.com"
The modification from the other user is: "kyle@td.com"

If your conflict_handling option was:

- 0: the email is:
  -"kyle@td.com" on WebDAV
  -"jack@td.com" on your local storage
  The storage rejects your latest modification,
  you get an error because local and remote documents are desynchronized.
  The documents in local and remote state are left untouched.
- 1: the email is: "jack@td.com" on both storages
  The storage pushes the local modification, which is yours.
- 2: the email is: "kyle@td.com" on both storages

The storage keeps the remote modification, which is from the other user.
Your local storage is modified to fit the state of the remote storage.

- 3: the email is: “jack@td.com” on both storages
  The storage doesn’t do synchronization, and pushes your modification
  without checking if the remote storage has been changed or not

3.4 List of Available Storages

jIO saves his job queue in a workspace which is localStorage by default. Provided storage descriptions are also stored, and it can be dangerous to store passwords.

The best way to create a storage description is to use the (often) provided tool given by the storage library. The returned description is secured to avoid cleartext, readable passwords (as opposed to encrypted passwords for instance).

When building storage trees, there is no limit on the number of storages you can use. The only thing you have to be aware of is compatibility of simple and revision based storages.

3.4.1 Connectors

LocalStorage

This storage has only one document, so **post**, **put**, **remove** and **get** method are useless on it.

parameter	required?	type	description
`type`	yes	string	name of the storage type (here: “local”)
`sessiononly`	no	boolean	false: create a storage with unlimited duration. true: the storage duration is limited to the user session. (default to false)

Example:

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

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.

Example:

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

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"
}
```

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"
}
```

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`	no	string	login and password of your dav, base64 encoded like this: `btoa(username + ":" + password)`
`with_credentials`	no	boolean	true: send domain cookie false: do not send domain cookie default to false.

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.

Dropbox

parameter	required?	type	description
`type`	yes	string	name of the storage type (here: "dropbox")
`access_token`	yes	string	access token for your account. See specific documentation on how to retreive it.
`root`	no	string	"dropbox" for full access to account files, "sandbox" for app limited file access. default to "dropbox".

Example:

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

Google Drive

parameter	required?	type	description
`type`	yes	string	name of the storage type (here: "gdrive")
`access_token`	yes	string	access token for your account. See specific documentation on how to retreive it.
`trashing`	no	boolean	true: sends files to the trash bin when doing a "remove" false: deletes permanently files when doing a "remove" default to true.

Example:

```
{
  "type":         "gdrive",
  "access_token": "sample_token"
```

(continues on next page)

(continued from previous page)

```
  "trashing":     true
}
```

ERP5Storage

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
}
```

3.4.2 Handlers

Zipstorage

This handler compresses and decompresses files to reduce network and storage usage.

Usage:

```
{
  "type":        "zip",
  "sub_storage": <your storage>
}
```

ShaStorage

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

```
{
  "type":        "sha",
  "sub_storage": <your storage>
}
```

UUIDStorage

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

```
{
  "type":         "uuid",
  "sub_storage": <your storage>
}
```

QueryStorage

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

```
{
  "type":         "query",
  "sub_storage": <your storage>
}
```

CryptStorage

This handler encrypts and decrypts attachments before storing them.
You need to generate a Crypto key at the JSON format to use the handler.
(see https://developer.mozilla.org/fr/docs/Web/API/Window/crypto for more informations)

Usage:

```
var key,
  jsonKey,
  jio;

//creation of an encryption/decryption key.

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})

//creation of the storage

jio = jIO.createJIO({
{
  "type":         "crypt",
  "key":          json_key
  "sub_storage": <your storage>
}
```

UnionStorage

This handler takes in argument an array of storages. When using a 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.

```
{
  "type":         "union",
  "storage_list": [
  sub_storage_description_1,
  sub_storage_description_2,
  sub_storage_description_X
 ]
}
```

FileSystemBridgeStorage

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

```
{
  "type": "drivetojiomapping",
  "sub_storage": <your dav storage>
}
```

Document Storage

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`	no	string	id of the document to use.
`repair_attachment`	no	boolean	verify if the document is in good state. (default to false)

Replicate Storage

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

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	object	remote sub_storage description.
`query_options`	no	object	query object to limit the synchronisation to specific files.
`use_remote_post`	no	boolean	true: at file modification, modifies the local file id. false: at file modification, modifies the remote file id. default to false.
`conflict_handling`	no	number	0: no conflict resolution (throws error) 1: keep the local state. 2: keep the remote state. 3: keep both states (no signature update) default to 0.
`check_local_modification`	no	boolean	synchronise when local files are modified.
`check_local_creation`	no	boolean	synchronise when local files are created.
`check_local_deletion`	no	boolean	synchronise when local files are deleted.
`check_remote_modification`	no	boolean	synchronise when remote files are modified.
`check_remote_creation`	no	boolean	synchronise when local files are created.
`check_remote_deletion`	no	boolean	synchronise when local files are deleted.

synchronisation parameters are set by default to true.

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

(continues on next page)

(continued from previous page)

```
  conflict_handling :        2,
  check_local_creation:      false,
  check_remote_deletion:     false
}
```

3.5 jIO Query

3.5.1 What are Queries?

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 JSCC.

3.5.2 Why use JIO Queries?

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. If distant storages are static, an alternative would be to use an indexStorage with appropriate indices as jio queries will always try to run the query on the index before querying documents itself.

3.5.3 How to use Queries with jIO?

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

Example:

```
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'
  }, {
```

(continues on next page)

(continued from previous page)

```
    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);
```

3.5.4 How to use Queries outside jIO?

Refer to the JIO Query sample page for how to use these methods, in and outside jIO. The module provides:

```
jIO: {
  QueryFactory: { [Function: QueryFactory] create: [Function] },
  Query: { [Function: Query],
    parseStringToObject: [Function],
    stringEscapeRegexpCharacters: [Function],
    select: [Function],
    sortOn: [Function],
    limit: [Function],
    searchTextToRegExp: [Function],
  }
  SimpleQuery: {
    [Function: SimpleQuery] super_: [Function: Query]
  },
  ComplexQuery: {
    [Function: ComplexQuery] super_: [Function: Query]
  }
}
```

(Reference API coming soon.)

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"';
```

(continues on next page)

(continued from previous page)

```
// 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);
```

Query in storage connectors

The query exec method must only be used if the server is not able to pre-select documents. As mentioned before, you could use an indexStorage to maintain indices with key information on all documents in a storage. This index file will then be used to run queries, if all of the fields required in the query answer are available in the index.

Matching properties

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
};
```

Should default search types be defined in jIO or in user interface components?

Default search types should be defined in the application's user interface components because criteria like filters will be changed frequently by the component (change `limit: [0, 10]` to `limit: [10, 10]` or `sort_on: [['title', 'ascending']]` to `sort_on: [['creator', 'ascending']]`) and each component must have its own default properties to keep their own behavior.

Query into another type

Example, convert Query object into a human readable string:

```
var query = jIO.QueryFactory.
  create('year: < 2000 OR title: "%a"'),
  option = {
    limit: [0, 10]
  },
  human_read = {
    "": "matches ",
    "<": "is lower than ",
    "<=": "is lower or equal than ",
    ">": "is greater than ",
    ">=": "is greater or equal than ",
    "=": "is equal to ",
    "!=": "is different than "
  };

query.onParseStart = function (object, option) {
  object.start = "We need only the " +
    option.limit[1] +
    " elements from the number " +
    option.limit[0] + ". ";
};

query.onParseSimpleQuery = function (object, option) {
  object.parsed = object.parsed.key +
    " " + human_read[object.parsed.operator || ""] +
    object.parsed.value;
};

query.onParseComplexQuery = function (object, option) {
  object.parsed = "I want all document where " +
    object.parsed.query_list.join(
      " " + object.parsed.operator.toLowerCase() + " "
    ) + ". ";
};

query.onParseEnd = function (object, option) {
  object.parsed = object.start + object.parsed + "Thank you!";
};

console.log(query.parse(option));
// logged: "We need only the 10 elements from the number 0. I want all
// document where year is lower than 2000 or title matches %a. Thank you!"
```

3.5.5 JSON Schemas and Grammar

Below you can find schemas for constructing queries.

- Complex Query JSON Schema:

```
{
  "id": "ComplexQuery",
  "properties": {
```

(continues on next page)

(continued from previous page)

```
    "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            -> /[^><!= :\(\)"][^ :\(\)"]*:/
    STRING            -> /"(\\.|[^\\"])*"/
    WORD              -> /[^><!= :\(\)"][^ :\(\)"]*/
    OPERATOR          -> /(>=?|<=?|!?=)/
    LEFT_PARENTHESE   -> /\(/
    RIGHT_PARENTHESE  -> /\)/

ignore: " "
```

3.6 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:

```
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'],
```

(continues on next page)

(continued from previous page)

```
    [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 (untested): https://github.com/walling/unorm/

3.6.1 Overriding 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;
    }
  };
}

...
```

(continues on next page)

(continued from previous page)

```
  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.

3.6.2 Partial Date/Time match

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

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`.

3.6.3 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())
      );
    },
```

(continues on next page)

(continued from previous page)

```
    sameMonth: function (a, b) {
      return (
        (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.

3.6.4 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
});
```

3.7 Metadata

3.7.1 What is metadata?

The word "metadata" means "data about data". Metadata articulates a context for objects of interest – "resources" such as MP3 files, library books, or satellite images – in the form of "resource descriptions". As a tradition, resource description dates back to the earliest archives and library catalogs. During the Web revolution of the mid-1990s, Dublin Core has emerged as one of the prominent metadata standards.

3.7.2 Why use metadata?

Uploading a document to several servers can be very tricky, because the document has to be saved in a place where it can be easily found with basic searches in all storages (For instance: ERP5, XWiki and Mioga2 have their own way to save documents and to get them). So we must use metadata for *interoperability reasons*. Interoperability is the ability of diverse systems and organizations to work together.

3.7.3 How to format metadata with jIO

See below XML and its JSON equivalent:

XML	JSON
`<dc:title>My Title</dc:title>`	`{"title": "My Title"}`

`<dc:contributor>Me</dc:contributor>`
`<dc:contributor>And You</dc:contributor>`
	`{"contributor": ["Me", "And You"]}`

`<dc:identifier scheme="DCTERMS.URI">`
`  http://my/resource`
`</dc:identifier>`
`<dc:identifier>`
`  Xaoe41PAPNIWz`
`</dc:identifier>`
	`{"identifier": [`
	`  {`
	`    "scheme": "DCTERMS.URI",`
	`    "content": "http://my/resource"`
	`  },`
	`  "Xaoe41PAPNIWz"`
	`]}`

3.7.4 List of metadata to use

Identification

- **identifier**

```
{"identifier":  "http://domain/jio_home_page"}
{"identifier":  "urn:ISBN:978-1-2345-6789-X"}
{"identifier":  [{"scheme":  "DCTERMS.URI", "content":
"http://domain/jio_home_page"}]}
```

An unambiguous reference to the resource within a given context. Recommended best practice is to identify the resource with a string or number conforming to a formal identification system. Examples of formal identification systems include the Uniform Resource Identifier (URI) (including the Uniform Resource Locator (URL)), the Digital Object Identifier (DOI) and the International Standard Book Number (ISBN).

- **format**

```
{"format":  ["text/html", "52 kB"]}
{"format":  ["image/jpeg", "100 x 100 pixels", "13.2 KiB"]}
```

The physical or digital manifestation of the resource. Typically, Format may include the media-type or dimensions of the resource. Examples of dimensions include size and duration. Format may be used to determine the software, hardware or other equipment needed to display or operate the resource.

- **date**

```
{"date":  "2011-12-13T14:15:16Z"}
{"date":  {"scheme":  "DCTERMS.W3CDTF", "content":  "2011-12-13"}}
```

A date associated with an event in the life cycle of the resource. Typically, Date is associated with the creation or availability of the resource. Recommended best practice for encoding the date value is defined in a profile of ISO 8601 Date and Time Formats, W3C Note and follows the YYYY-MM-DD format.

- **type**

```
{"type":  "Text"}
{"type":  "Image"}
{"type":  "Dataset"}
```

The nature or genre of the content of the resource. Type includes terms describing general categories, functions, genres, or aggregation levels for content. Recommended best practice is to select a value from a controlled vocabulary. **This type is not a MIME Type!**

Intellectual property

- **creator**

```
{"creator":  "Tristan Cavelier"}
{"creator":  ["Tristan Cavelier", "Sven Franck"]}
```

An entity primarily responsible for creating the content of the resource. Examples of a Creator include a person, an organization, or a service. Typically the name of the Creator should be used to indicate the entity.

- **publisher**

```
{"publisher":  "Nexedi"}
```

The entity responsible for making the resource available. Examples of a Publisher include a person, an organization, or a service. Typically, the name of a Publisher should be used to indicate the entity.

- **contributor**

```
{"contributor":  ["Full Name", "Full Name", ...]}
```

An entity responsible for making contributions to the content of the resource. Examples of a Contributor include a person, an organization or a service. Typically, the name of a Contributor should be used to indicate the entity.

- **rights**

```
{"rights":  "Access limited to members"}
{"rights":  "https://www.j-io.org/documentation/jio-documentation/
#copyright-and-license"}
```

Information about rights held in and over the resource. Typically a Rights element should contain a rights management statement for the resource, or reference a service providing such information. Rights information often encompasses Intellectual Property Rights (IPR), Copyright, and various Property Rights. If the rights element is absent, no assumptions can be made about the status of these and other rights with respect to the resource.

Content

- **title**

```
{"title":  "jIO Home Page"}
```

The name given to the resource. Typically, a Title is a name by which the resource is formally known.

- **subject**

```
{"subject":  "jIO"}
{"subject":  ["jIO", "basics"]}
```

The topic of the content of the resource. Typically, a Subject is expressed as keywords or key phrases or classification codes that describe the topic of the resource. Recommended best practice is to select a value from a controlled vocabulary or formal classification scheme.

- **description**

```
{"description":  "Simple guide to show the basics of jIO"}
{"description":  {"lang":  "fr", "content":  "Ma description"}}
```

An account of the content of the resource. Description may include but is not limited to: an abstract, table of contents, reference to a graphical representation of content or a free-text account of the content.

- **language**

```
{"language":  "en"}
```

The language of the intellectual content of the resource. Recommended best practice for the values of the Language element is defined by RFC 3066 which, in conjunction with ISO 639, defines two- and three-letter primary language tags with optional subtags. Examples include "en" or "eng" for English, "akk" for Akkadian, and "en-GB" for English used in the United Kingdom.

- **source**

```
{"source":  ["Image taken from a drawing by Mr. Artist", "<phone
number>"]}
```

A Reference to a resource from which the present resource is derived. The present resource may be derived from the Source resource in whole or part. Recommended best practice is to reference the resource by means of a string or number conforming to a formal identification system.

- **relation**

```
{"relation":  "Resilience project"}
```

A reference to a related resource. Recommended best practice is to reference the resource by means of a string or number conforming to a formal identification system.

- **coverage**

```
{"coverage":  "France"}
```

The extent or scope of the content of the resource. Coverage would typically include spatial location (a place name or geographic co-ordinates), temporal period (a period label, date, or date range) or jurisdiction (such as a named administrative entity). Recommended best practice is to select a value from a controlled vocabulary (for example, the Getty Thesaurus of Geographic Names. Where appropriate, named places or time periods should be used in preference to numeric identifiers such as sets of co-ordinates or date ranges.

- **category**

```
{"category":  ["parent/26323", "resilience/javascript",
"javascript/library/io"]}
```

The category the resource is associated with. The categories may look like navigational facets, they correspond to the properties of the resource which can be generated with metadata or some other information (see faceted search).

- **product**

```
{"product":  "..."}
```

For e-commerce use.

- **custom**

```
{custom1:  value1, custom2:  value2, ...}
```

3.7.5 Examples

Posting a webpage for jIO

```
jio.post({
  "identifier" : "http://domain/jio_home_page",
  "format"     : ["text/html", "52 kB"],
  "date"       : new Date(),
  "type"       : "Text",
  "creator"    : ["Nexedi", "Tristan Cavelier", "Sven Franck"],
  "title"      : "jIO Home Page",
  "subject"    : ["jIO", "basics"],
  "description": "Simple guide to show the basics of jIO",
  "category"   : ["resilience/jio", "webpage"],
  "language"   : "en"
}); // send content as attachment
```

Posting jIO library

```
jio.post({
  "identifier" : "jio.js",
  "date"       : "2013-02-15",
  "format"     : "application/javascript",
  "type"       : "Software",
  "creator"    : ["Tristan Cavelier", "Sven Franck"],
  "publisher"  : "Nexedi",
  "rights"     :
    "https://www.j-io.org/documentation/" +
      "jio-documentation/#copyright-and-license",
  "title"      : "Javascript Input/Output",
  "subject"    : "jIO",
  "category"   : [
                   "resilience/javascript",
                   "javascript/library/io"
                 ]
  "description": "jIO is a client-side JavaScript library to " +
                 "manage documents across multiple storages."
}); // send content as attachment
```

Posting a webpage for interoperability levels

```
jio.post({
  "identifier" : "http://dublincore.org/documents/" +
                   "interoperability-levels/",
  "date"       : "2009-05-01",
  "format"     : "text/html",
  "type"       : "Text",
  "creator"    : [
                  "Mikael Nilsson",
                  "Thomas Baker",
```

(continues on next page)

(continued from previous page)

```
                  "Pete Johnston"
                 ],
  "publisher"   : "Dublin Core Metadata Initiative",
  "title"       : "Interoperability Levels for Dublin Core Metadata",
  "description": "This document discusses the design choices " +
                 "involved in designing applications for " +
                 "different types of interoperability. [...]",
  "language"    : "en"
}); // send content as attachment
```

Posting an image

```
jio.post({
  "identifier" : "new_york_city_at_night",
  "format"     : ["image/jpeg", "7.2 MB", "8192 x 4096 pixels"],
  "date"       : "1999",
  "type"       : "Image",
  "creator"    : "Mr. Someone",
  "title"      : "New York City at Night",
  "subject"    : ["New York"],
  "description": "A photo of New York City taken just after midnight",
  "coverage"   : ["New York", "1996-1997"]
}); // send content as attachment
```

Posting a book

```
jio.post({
  "identifier" : {
                     "scheme": "DCTERMS.URI",
                     "content": "urn:ISBN:0385424728"
                 },
  "format"     : "application/pdf",
  "date"       : {
                     "scheme": "DCTERMS.W3CDTF",
                     "content": getW3CDate()
                 }, // see tools below
  "creator"    : "Original Author(s)",
  "publisher"  : "Me",
  "title"      : {"lang": "en", "content": "..."},
  "description": {"lang": "en", "Summary: ..."},
  "language"   : {
                     "scheme": "DCTERMS.RFC4646",
                     "content": "en-GB"
                 }
}); // send content as attachment
```

Posting a video

```
jio.post({
  "identifier" : "my_video",
  "format"     : ["video/ogg", "130 MB", "1080p", "20 seconds"],
```

(continues on next page)

(continued from previous page)

```
  "date"       : getW3CDate(), // see tools below
  "type"       : "Video",
  "creator"    : "Me",
  "title"      : "My life",
  "description": "A video about my life"
}); // send content as attachment
```

Posting a job announcement

```
jio.post({
  "format"     : "text/html",
  "date"       : "2013-02-14T14:44Z",
  "type"       : "Text",
  "creator"    : "James Douglas",
  "publisher"  : "Morgan Healey Ltd",
  "title"      : "E-Commerce Product Manager",
  "subject"    : "Job Announcement",
  "description": "Announcement for e-commerce product manager job",
  "language"   : "en-GB",
  "source"     : "James@morganhealey.com",
  "relation"   : ["Totaljobs"],
  "coverage"   : "London, South East",
  "job_type"   : "Permanent",
  "salary"     : "£45,000 per annum"
}); // send content as attachment
```

Getting a list of document created by someone

With query:

```
jio.allDocs({"query": "creator: \"someone\""});
```

Getting all documents about jIO in the resilience project

With query:

```
jio.allDocs({
  "query": 'subject: "jIO" AND category: "resilience"'
});
```

3.7.6 Tools

Date functions

```
// Get RFC1123 date format "Tue, 13 Dec 2011 13:15:16 GMT"
new Date().toUTCString();

// Get ISO8601 date format "2011-12-13T13:15:16.433Z"
new Date().toISOString();
```

(continues on next page)

(continued from previous page)

```
/**
 * Tool to get the date in W3C date format.
 *
 *     "2011-12-13T14:15:16.433+01:00"
 *
 * @param  {Date} [date] The date to convert
 * @return {String} The date in W3C date format
 */
function getW3CDate(date) {
  var d = date || new Date(), offset = - d.getTimezoneOffset();
  return (
    d.getFullYear() + "-" +
      (d.getMonth() + 1) + "-" +
      d.getDate() + "T" +
      d.getHours() + ":" +
      d.getMinutes() + ":" +
      d.getSeconds() + "." +
      d.getMilliseconds() +
      (offset < 0 ?
       "-" + parseInt(-offset / 60, 10) + ":" + (-offset % 60) :
       "+" + parseInt(offset / 60, 10) + ":" + (offset % 60))
  ).replace(/[0-9]+/g, function (found) {
    if (found.length < 2) {
      return '0' + found;
    }
    return found;
  });
}
```

3.7.7 Sources

- Interoperability definition
- Faceted search
- DublinCore
  - Interoperability levels
  - Metadata elements
  - http://www.chu-rouen.fr/documed/eahilsantander.html
  - http://openweb.eu.org/articles/dublin_core (French)
- CouchDB
- Resource Description Framework (RDF)
- Five Ws
- Metadata
- MIME Types
  - https://en.wikipedia.org/wiki/Internet_media_type
  - https://www.iana.org/assignments/media-types

3.8 For developers

3.8.1 Quick start

The source repository includes ready-to-use files, so in case you do not want to build jIO yourself, just use *sha256.amd.js*, *rsvp.js*, *jio.js* plus the storages and dependencies you need and you will be good to go.

If you want to modify or build jIO yourself, you need to

- Clone from a repository

  ```
  $ git clone https://lab.nexedi.com/nexedi/jio.git
  ```

- Install NodeJS (including `npm`)
- Install the Grunt command line with `npm`.

  ```
  # npm -g install grunt-cli
  ```

- Install the dependencies.

  ```
  $ npm install
  ```

- Compile the JS/CC parser.

  `$ make` (until we find out how to compile it with grunt)

- Run build.

  ```
  $ grunt
  ```

3.8.2 Naming Conventions

All the code follows this *JavaScript Style Guide*.

3.8.3 How to design your own jIO Storage Library

Create a constructor:

```
function MyStorage(storage_description) {
  this._value = storage_description.value;
  if (typeof this._value !== 'string') {
    throw new TypeError("'value' description property " +
                        "is not a string");
  }
}
```

Your storage must be added to jIO. This is done with the `jIO.addStorage()` method, which requires two parameters: the storage type (string) and a constructor (function). It is called like this:

```
// add custom storage to jIO
jIO.addStorage('mystoragetype', MyStorage);
```

Please refer to *localstorage.js* implementation for a good example on how to setup a storage and what methods are required.

3.9 JavaScript Style Guide

This document defines JavaScript style conventions, which are split into essential, coding and naming conventions.

3.9.1 Essential Conventions

Essential conventions include generic patterns that you should adhere to in order to write *readable*, *consistent* and *maintainable* code.

Minimizing Globals

Variable declarations should always be done using *var* to not declare them as global variables. This avoids conflicts from using a variable name across different functions as well as conflicts with global variables declared by third party plugins.

Good Example

```
function sum(x, y) {
  var result = x + y;
  return result;
}
```

Bad Example

```
function sum(x, y) {
  // missing var declaration, implied global
  result = x + y;
  return result;
}
```

Using JSLint

JSLint is a quality tool that inspects code and warns about potential problems. It can be used online and can also be integrated into several development environments, so errors can be highlighted while writing code.

Before validating your code in JSLint, you should use a code beautifier to fix basic syntax errors (like indentation) automatically. There are a number of beautifiers available online. The following ones seem to work best:

- JSbeautifier.org
- JS-Beautify

In this project, JavaScript sources have to begin with the header:

```
/*jslint indent: 2, maxlen: 80, nomen: true */
```

which means it uses two spaces indentation, 80 maximum characters per line and allows variable names starting with '_'. Other JSLint options can be added in sub functions if necessary.

Some allowed options are:

- `ass: true` if assignment should be allowed outside of statement position.
- `bitwise: true` if bitwise operators should be allowed.
- `continue: true` if the continue statement should be allowed.

- `newcap`: `true` if Initial Caps with constructor function is optional.
- `regexp`: `true` if `.` and `[^...]` should be allowed in RegExp literals. They match more material than might be expected, allowing attackers to confuse applications. These forms should not be used when validating in secure applications.
- `unparam`: `true` if warnings should be silenced for unused parameters.

3.9.2 Coding Conventions

Coding conventions include generic patterns that ensure the written code is consistently formatted.

Using two-space indentation

Tabs and 2-space indentation are being used equally. Since a lot of errors on JSLint often result from mixed use of space and tab, using 2 spaces throughout prevents these errors up front.

Good Example

```
function outer(a, b) {
  var c = 1,
    d = 2,
    inner;
  if (a > b) {
    inner = function () {
      return {
        "r": c - d
      };
    };
  } else {
    inner = function () {
      return {
        "r": c + d
      };
    };
  }
  return inner;
}
```

Bad Example

```
function outer(a, b) {
var c = 1,
d = 2,
inner;

if (a > b) {
inner = function () {
return {
r: c - d
}}};
```

Using shorthand for conditional statements

An alternative for using braces is the shorthand notation for conditional statements. When using multiple conditions, the conditional statement can be split on multiple lines.

Good Example

```
// single line
var results = test === true ? alert(1) : alert(2);

// multiple lines
var results = (test === true && number === undefined ?
               alert(1) : alert(2));

var results = (test === true ?
               alert(1) : number === undefined ?
               alert(2) : alert(3));
```

Bad Example

```
// multiple conditions
var results = (test === true && number === undefined) ?
  alert(1) :
  alert(2);
```

Opening Brace Location

Always put the opening brace on the same line as the previous statement.

Bad Example

```
function func()
{
  return
  {
    "name": "Batman"
  };
}
```

Good Example

```
function func() {
  return {
    "name": "Batman"
  };
}
```

Closing Brace Location

The closing brace should be on the same indent level as the original function call.

Bad Example

```
function func() {
  return {
           "name": "Batman"
         };
}
```

Good Example

```
function func() {
  return {
    "name": "Batman"
  };
}
```

Variable Declaration Location

Every variables should be declared at the top of its function.

Bad Example

```
function first() {
  var a = 1, b = 2,
    c, d;

  // ...
}

function second() {
  var a = 1, b = 2, c;
  var d;

  // ...
}
```

Good Example

```
function third() {
  var a = 1, b = 2, c, d;

  // ...
}

function fourth() {
  var a = 1,
    b = 2,
    c,
    d;

  // ...
}
```

Function Declaration Location

Non anonymous functions should be declared before use and before every statements.

Bad Example

```
if (...) {
  return {
    "namedFunction": function namedFunction() { ... }
  };
}
```

(continues on next page)

(continued from previous page)

```
// or
if (...) {
  function namedFunction() { ... }
  return {
    "namedFunction": namedFunction
  };
}
```

Good Example

```
function namedFunction() { ... }

if (...) {
  return {
    "namedFunction": namedFunction
  };
}
```

Anonymous Function Location

Execept if you want to keep your function without name, anonymous functions must not be declared in the same place as the variables.

Bad Example

```
function first() {
  var a = 1, b = 2, func = function () {
    return a;
  };
  // ...
}
```

Good Example

```
function second() {
  var a = 1, b = 2;

  function func() {
    return a;
  };
  // ...
}
```

You can assign a variable to an anonymous function inside **non-loop** statements.

Bad Example

```
function third() {
  var a = 1, b = 2, func;

  for (...) {
    b.forEach(function () { ... });
  }
  // ...
}
```

Good Example

```
function fourth() {
  var a = 1, b = 2, func;

  if (...) {
    func = function () { ... };
  } else {
    func = function () { ... };
  }
  // ...
}

function fifth() {
  var a = [], b = [];

  function func() { ... }

  for (...) {
    b.forEach(func);
  }
  // ...
}
```

3.9.3 Naming Conventions

Naming conventions include generic patterns for setting names and identifiers throughout a script.

Constructors

Constructor functions (called with the `new` statement) should always start with a capital letter:

```
// bad example
var test = new application();

// good example
var test = new Application();
```

Methods/Functions

A method/function should always start with a small letter.

```
// bad example
function MyFunction() {...}

// good example
function myFunction() {...}
```

TitleCase, camelCase

Follow the camel case convention, typing the words in lower-case, only capitalizing the first letter in each word.

```
// Good example constructor = TitleCase
var test = new PrototypeApplication();

// Bad example constructor
var test = new PROTOTYPEAPPLICATION();

// Good example functions/methods = camelCase
myFunction();
calculateArea();

// Bad example functions/methods
MyFunction();
CalculateArea();
```

Variables

Variable names with multiple words should always use an underscore between them.

```
// bad example
var deliveryNote = 1;

// good example
var delivery_note = 1;
```

Confusing variable names should end with the variable type.

```
// implicit type
var my_callback = doSomething();
var Person = require("./person");

// confusing names + var type
var do_something_function = doSomething.bind(context);
var value_list = getObjectOrArray();
// value_list can be an object which can be cast into an array
```

To use camelCase, when sometimes it is not possible to declare a function directly, the function variable name should match some pattern which shows that it is a function.

```
// good example
var doSomethingFunction = function () { ... };
// or
var tool = {"doSomething": function () { ... }};

// bad example
var doSomething = function () { ... };
```

Element Classes and IDs

JavaScript can access elements by their ID attribute and class names. When assigning IDs and class names with multiple words, these should also be separated by an underscore (same as variables).

Example

```
// bad example
test.setAttribute("id", "uniqueIdentifier");

// good example
test.setAttribute("id", "unique_identifier");
```

Discuss - checked with jQuery UI/jQuery Mobile, they don't use written name conventions, only

- events names should fit their purpose (pageChange for changing a page)
- element classes use "-" like in ui-shadow
- "ui" should not be used by third party developers
- variables and events use lower camel-case like pageChange and activePage

Underscore Private Methods

Private methods should use a leading underscore to separate them from public methods (although this does not technically make a method private).

Good Example

```
var person = {
  "getName": function () {
    return this._getFirst() + " " + this._getLast();
  },
  "_getFirst": function () {
    // ...
  },
  "_getLast": function () {
    // ...
  }
};
```

Bad Example

```
var person = {
  "getName": function () {
    return this.getFirst() + " " + this.getLast();
  },
  // private function
  "getFirst": function () {
    // ...
  }
};
```

No Abbreviations

Abbreviations should not be used to avoid confusion.

Good Example

```
// delivery note
var delivery_note = 1;
```

Bad Example

```
// delivery note
var del_note = 1;
```

No Plurals

Plurals should not be used as variable names.

```
// good example
var delivery_note_list = ["one", "two"];

// bad example
var delivery_notes = ["one", "two"];
```

Use Comments

Comments should be used within reason but include enough information so that a reader can get a first grasp of what a part of code is supposed to do.

Good Example

```
var person = {
  // returns full name string
  "getName": function () {
    return this._getFirst() + " " + this._getLast();
  }
};
```

Bad Example

```
var person = {
  "getName": function () {
    return this._getFirst() + " " + this._getLast();
  }
};
```

Documentation

You can use YUIDoc and its custom comment tags together with Node.js to generate the documentation from the script file itself. Comments should look like this:

Good Example

```
/**
 * Reverse a string
 *
 * @param  {String} input_string String to reverse
 * @return {String} The reversed string
 */
function reverse(input_string) {
  // ...
  return output_string;
};
```

Bad Example

```
function reverse(input_string) {
  // ...
  return output_string;
};
```

3.9.4 Additional Readings

Resources, additional reading materials and links:

- JavaScript Patterns, main resource used.
- JSLint, code quality tool.
- JSLint Error Explanations, a useful reference.
- YUIDoc, generate documentation from code.

3.10 Authors

- Francois Billioud
- Tristan Cavelier
- Sven Franck
- Romain Courteaud