Javascript Input/Output

Documentation Index

Introduction

What is jIO?

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.

How does it work?

JIO is separated into three parts - jIO core and storage library(ies). The core is using storage libraries

(connectors) to interact with the accociated remote storage servers. Some queries can be used on top of the jIO allDocs method to query documents based on defined criteria.

JIO uses a job management system, so every method called adds a job into a queue. The queue is copied in the browsers local storage (by default), so it can be restored in case of a browser crash. Jobs are being invoked asynchronously with ongoing jobs not being able to re-trigger to prevent conflicts.

Getting started

This walkthrough is designed to get you started using a basic jIO instance.

1. Download jIO core, the storages you want to use as well as the complex-queries scripts as well as the dependencies required for the storages you intend to use. [Download & Fork]

2. Add the scripts to your HTML page in the following order:

```
<!-- jio core + dependency -->
<script src="sha256.amd.js"></script>
<script src="rsvp-custom.js"></script>
<script src="jio.js"></script>


<!-- storages + dependencies -->
<script src="complex_queries.js"></script>
<script src="localstorage.js"></script>
<script src="davstorage.js"></script>


<script ...>
```

   With require js, the main.js will be like this:

```
require.config({
  "paths": {
    // jio core + dependency
    "sha256": "sha256.amd", // the AMD compatible version of sha256.js -> see Download and Fork
    "rsvp": "rsvp-custom",
    "jio": "jio",
    // storages + dependencies
    "complex_queries": "complex_queries",
    "localstorage": "localstorage",
    "davstorage": "davstorage"
  }
});
```

3. jIO connects to a number of storages and allows to add handlers (or functions) to specifc 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. See the list of available storages.

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

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

Method	Sample Call	Description
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 attachment
allDocs	my_jio.allDocs([options]);	Retrieves a list of existing documents
check	my_jio.check(document, [options]);	Check the document state
repair	my_jio.repair(document, [options]);	Repair the document

How to manage documents?

JIO is mapped after the CouchDB API and extends it to provide unified, scalable and high performance access via Javascript to a wide variety of different storage backends.

If you are unfamiliar with ApacheCouchDB: it is a scalable, fault-tolerant, and schema-free **document-oriented** database. It's used in large and small organizations for a variety of applications where traditional SQL databases aren't the best solution for the problem at hand. CouchDB provides a **REST**ful HTTP/JSON API accessible from many programming libraries and tools (like 'curl' or 'Pouchdb') and has it's own **conflict management system**.

What is a document?

A document an association with metadata and attachment(s). The metadata are the properties of the document and the attachments are the binaries of the content of the document.

In jIO, metadata are just a dictionnary with keys and values (JSON object), and attachments are just simple strings.

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

You can also retrieve document attachment metadata in this object.

```
{ // document metadata
  "_id"   : "Identifier",
  "title" : "A Title!",
  "creator": "Mr.Author",
  "_attachments": { // attachment metadata
    "body.html": {
      "length": 12893,
      "digest": "sha256-XXXX...",
      "content_type": "text/html"
    }
  }
}
```

Here is a draft about metadata to use with jIO.

Basic Methods

Below you can find sample calls of the main jIO methods. All examples are using revisions (as in revision storage or replicate revision storage), so you can see, how method calls should be made with either of these storages.

```
// Create a new jIO instance
var jio_instance = jIO.newJio(storage tree description);

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

// create or update an existing document
jio_instance.put({"_id": "my_document", "title": "New Title"}).
  then(function (response) {
    // console.log(response);
  });

// add an attachement to a document
jio_instance.putAttachment({"_id": "my_document", "_attachment": "its_attachment",
                            "_data": "abc", "_mimetype": "text/plain"}).
  then(function (response) {
    // console.log(response);
  });

// read a document
jio_instance.get({"_id": "my_document"}).
  then(function (response) {
    // console.log(response);
  });

// read an attachement
jio_instance.getAttachment({"_id": "my_document", "_attachment": "its_attachment"}).
  then(function (response) {
    // console.log(response);
  });
```

```
// delete a document and its attachment(s)
jio_instance.remove({"_id": "my_document"}).
  then(function (response) {
    // console.log(response);
  });

// delete an attachement
jio_instance.removeAttachment({"_id": "my_document", "_attachment": "its_attachment"}).
  then(function (response) {
    // console.log(response);
  });

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

Promises

Each JIO methods return 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:

- github RSVP.js
- Promises/A+
- CommonJS Promises

Method Options and Callback Responses

To retrieve JIO responses, you have to give callbacks like this:

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

- On command success, responseCallback will be called with the JIO response as first parameter.
- On command error, errorCallback will be called with the JIO error as first parameter.
- On command notification, progressionCallback will be called with the storage notification.

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

- Available Options (Methods)

Option	Available for	Response (Callback first parameter)
No options	post, put, remove	{ "result": "success", "method": "post", // put or remove "id": "my_doc_id", "status": 204, "statusText": "No Content" }
No options	putAttachment, removeAttachment	{ "result": "success", "method": "putAttachment", // or removeAttachment "id": "my_doc_id", "attachment": "my_attachment_id", "status": 204, "statusText": "No Content" }
No options	get	{ "result": "success", "method": "get", "id": "my_doc_id", "status": 200, "statusText": "Ok", "data": { // Here, the document metadata } }
No options	getAttachment	{ "result": "success", "method": "getAttachment", "id": "my_doc_id", "attachment": "my_attachment_id", "status": 200, "statusText": "Ok",

Option	Available for	Response (Callback first parameter)
		"data": Blob // Here, the attachment content }

No option	allDocs	{
		"result": "success",
		"method": "allDocs",
		"id": "my_doc_id",
		"status": 200,
		"statusText": "Ok",
		"data": {
		"total_rows": 1,
		"rows": [{
		"id": "mydoc",
		"key": "mydoc", // optional
		"value": {},
		}]
		}
		}

include_docs: true	allDocs	{
		"result": "success",
		"method": "allDocs",
		"id": "my_doc_id",
		"status": 200,
		"statusText": "Ok",
		"data": {
		"total_rows": 1,
		"rows": [{
		"id": "mydoc",
		"key": "mydoc", // optional
		"value": {},
		"doc": {
		// Here, "mydoc" metadata
		}
		}]
		}
		}

In case of error, the errorCallback first parameter will look like:

```
{
  "result": "error",
  "method": "get",
  "status": 404,
  "statusText": "Not Found",
  "error": "not_found",
  "reason": "document missing",
  "message": "Unable to get the requseted document"
}
```

Example: How to store a video on localStorage

The following shows how to create a new jIO in localStorage and then post a document with two attachments.

```
// create a new jIO
var jio_instance = jIO.createJIO({
  "type": "local",
  "username": "usr",
  "application_name": "app"
});

function postMyVideoMetadata() {
  return jio_instance.post({
    "title"       : "My Video",
    "type"        : "MovingImage",
    "format"      : "video/ogg",
    "description" : "Images Compilation"
  }, putThumbnailAttachment);
}

function putThumbnailAttachment(err, response) {
  var id;
  if (err) {
    return alert('Error posting the document meta');
  }
  id = response.id;
  // post a thumbnail attachment
  return jio_instance.putAttachment({
    "_id": id,
    "_attachment": "thumbnail",
    "_data": my_image,
    "_mimetype": "image/jpeg"
  }, putVideoContent);
}
```

```
function putVideoContent(err, response) {
  if (err) {
    return alert('Error attaching thumbnail');
  }
  // put video attachment
  return jio_instance.putAttachment({
    "_id": id,
    "_attachment": "video",
    "_data": my_video,
    "_mimetype":"video/ogg"
  }, checkResult);
}

function checkResult(err, response) {
  if (err) {
    return alert('Error attaching the video');
  }
  alert('Video Stored');
}

postMyVideoMetadata();
```

localStorage contents:

```
{
  "jio/local/usr/app/12345678-1234-1234-1234-123456789012": {
    "_id": "12345678-1234-1234-1234-123456789012",
    "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
      }
    }
  },
  "jio/local/usr/app/myVideo/thumbnail": "/9j/4AAQSkZ...",
  "jio/local/usr/app/myVideo/video": "..."
}
```

Revision Storages: Conflicts and Resolution

Why Conflicts can Occur

Using jIO you can store documents in multiple storage locations. With 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 revision 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.

How to solve conflicts

Using the document tree, jIO tries to make every version of a document available on every storage. When multiple versions of a document exist, jIO will select the **latest, left-most** version on the document tree, along with the conflicting versions (when option **conflicts: true** is set in order for developers to setup a routine to solve conflicts.

Technically a conflict is solved by deleting alternative versions of a document ("cutting leaves off from the document tree"). When a user decides to keep a version of a document and manually deletes all conflicting versions, the storage tree is updated accordingly and the document is available in a single version on all storages.

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 change this email from your PC and once your smartphone is recharged, you go back online and the previous update is executed.

1. **Setting up the storage tree**

```
var jio_instance = jIO.newJio({
  // replicate revision storage
  "type":"replicaterevision",
  "storagelist":[{
    "type": "revision",
    "sub_storage": {
      "type": "dav",
      ...
    }
  }, {
    "type": "revision",
    "sub_storage": {
      "type": "local",
      ...
    }
  }]
});
```

2. **Create your namecard on your smartphone**

```
jio_instance.post({
  "_id": "myNameCard",
  "email": "me@web.com"
}).then(function (response) {
  // response.id -> "myNameCard"
  // response.rev -> "1-5782E71F1E4BF698FA3793D9D5A96393"
});
```

   This will create the document on your webDav and local storage

3. **Someone else updates your shared namecard on Webdav**

```
jio_instance.put({
  "email": "my_new_me@web.com",
  "_id": "myNameCard"
  "_rev": "1-5782E71F1E4BF698FA3793D9D5A96393"
}).then(function (response) {
  // response.id -> "myNameCard"
  // response.rev -> "2-068E73F5B44FEC987B51354DFC772891"
});
```

   Your smartphone is offline, so you will now have one version (*1-578...*) on your smartphone and another version on webDav (*2-068...*) on your PC.

4. **You modify your namecard while being offline**

```
jio_instance.get({"_id": "myNameCard"}).then(function (response) {
  // response.id -> "myNameCard"
  // response.rev -> "1-5782E71F1E4BF698FA3793D9D5A96393"
  // response.data.email -> "me@web.com"


  return jio_instance.put({
    "_id": "myNameCard",
    "email": "me_again@web.com"
  });


}).then(function (response) {
  // response.id -> "myNameCard"
  // response.rev -> "2-3753476B70A49EA4D8C9039E7B04254C"
});
```

5. **Later, your smartphone is online and you retrieve your namecard.**

```
jio_instance.get({"_id": "myNameCard"}).then(function (response) {
  // response.id -> "myNameCard"
  // response.rev -> "2-3753476B70A49EA4D8C9039E7B04254C"
  // response.data.email -> "me_again@web.com"
});
```

   When multiple versions of a document are available, jIO returns the lastest, left-most version on the document tree (*2-375...* and labels all other versions as conflicting*2-068...*).

6. **Retrieve conflicts by setting option**

```
jio_instance.get({"_id": "myNameCard"}, {
```

```
  "conflicts": true
}).then(function (response) {
  // response.id -> "myNameCard"
  // response.rev -> "2-3753476B70A49EA4D8C9039E7B04254C",
  // response.conflicts -> ["2-068E73F5B44FEC987B51354DFC772891"]
});
```

The conflicting version (*2-068E...*) is displayed, because *{conflicts: true}* was specified in the GET call. Deleting either version will solve the conflict.

7. **Delete conflicting version**

```
jio_instance.remove({
  "_id": "myNameCard",
  "_rev": "2-068E73F5B44FEC987B51354DFC772891"
}).then(function (response) {
  // response.id -> "myNameCard"
  // response.rev -> "3-28910A4937537B5168E772896B70EC98"
});
```

When deleting the conflicting version of your namecard, jIO removes this version from all storages and sets the document tree leaf of this version to *deleted*. All storages now contain just a single version of your namecard (*2-3753...*). Note, that the on the document tree, removing a revison will create a new revision with status set to *deleted*.

List of Available Storages

JIO save his job queue in a workspace which is localStorage by default. Provided storage descirption are also stored, and it can be dangerous if we 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 clear readable password. (enciphered password 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 compatability of *simple* and *revision based* storages.

Connectors

LocalStorage

Three methods are provided:

- createDescription(username, [application_name], [mode="localStorage"])
- createLocalDescription(username, [application_name])
- createMemoryDescription(username, [application_name])

All parameters are strings.

Examples:

```
// to work on browser localStorage
var jio = jIO.createJIO(local_storage.createDescription("me"));

// to work on browser memory
var jio = jIO.createJIO(local_storage.createMemoryDescription("me"));
```

DavStorage

The tool dav_storage.createDescription generates a dav storage description for *no*, *basic* or *digest* authentication (*digest* is not implemented yet).

```
dav_storage.createDescription(url, auth_type, [realm], [username], [password]);
```

All parameters are strings.

Only url and auth_type are required. If auth_type is equal to "none", then realm, username and password are useless. username and password become required if auth_type is equal to "basic". And realm also becomes required if auth_type is equal to "digest".

digest **is not implemented yet**

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

S3Storage

Updating to v2.0

XWikiStorage

Updating to v2.0

Handlers

IndexStorage

This handler indexes documents metadata into a database (which is a simple document) to increase the speed of allDocs requests. However, it is not able to manage the `include_docs` option.

The sub storages have to manage `query` and `include_docs` options.

Here is the description:

```
{
  "type": "index",
  "indices": [{
    "id": "index_title_subject.json", // doc id where to store indices
    "index": ["title", "subject"], // metadata to index
    "attachment": "db.json", // default "body"
    "metadata": { // additional metadata to add to database, default undefined
      "type": "Dataset",
      "format": "application/json",
      "title": "My index database",
      "creator": "Me"
    },
    "sub_storage": <sub storage where to store index>
                   // default equal to parent sub_storage field
  }, {
    "id": "index_year.json",
    "index": "year"
    ...
  }],
  "sub_storage": <sub storage description>
}
```

GIDStorage

Full description here.

Updating to v2.0

SplitStorage

Updating to v2.0

Replicate Storage

Comming soon

Revision Based Handlers

A revision based handler is a storage which is able to do some document versionning using simple storages listed above.

On JIO command parameter, `_id` is still used to identify a document, but another id `_rev` must be defined to use a specific revision of this document.

On command responses, you will find another field `rev` which will represent the new revision produced by your action. All the document history is kept unless you decide to delete older revisions.

Another fields `conflicts`, `revisions` and `revs_info` can be returned if the options `conflicts: true`, `revs: true` and `revs_info: true` are set.

Revision Storage

Updating to v2.0

Replicate Revision Storage

Updating to v2.0

JIO Complex Queries

Only one dependency is needed: <script src="rsvp-custom.js"></script>

What are Complex Queries?

In jIO, a complex query can tell 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 complex query tool can act on the retreived list by itself. Only the *allDocs* method can use complex 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 complex queries, jIO uses a parsed grammar file which is complied using JSCC [link].

Why use Complex Queries?

Complex queries can be used similar to database queries. So they are useful to:

- search a specific document
- sort a list of documents in a certain order
- avoid retreiving a list of ten thousand documents
- limit the list to show only xy documents by page

For some storages (like localStorage), complex 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 having to request large amount of documents to run a query on the client. If distant storages are static, an alternative would be to use an indexStorage with appropriate indices as complex queries will always try to run the query on the index before querying documents itself.

How to use Complex Queries with jIO?

Complex queries can be triggered by including the option named*query* in the *allDocs* method call. An example would be:

```
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'],
  wildcard_character: '%'
};

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

How to use Complex Queries outside jIO?

Complex Queries provides an API - which namespace is`complex_queries`. Please also refer to the Complex Queries sample page on how to use these methods in- and outside jIO. The module provides:

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

(Reference API comming 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"';

// running the query
complex_queries.QueryFactory.create(query).exec(object_list).then(function (result) {
  // here: object_list and result are the same object. It means that object_list has been modified.
  // console.log(result);
  // [ { "title": "Document number 1", "creator": "John Doe"} ]
});
```

Other example:

```
complex_queries.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"
  }
).then(operateResult);
// this case is equal to:
complex_queries.QueryFactory.create(query).exec(object_list).then(function (result) {
  complex_queries.sortOn([['title', 'ascending'], ['year', 'descending']], result);
  complex_queries.limit([20, 20], result);
  complex_queries.select(['title', 'year'], result);
  return result;
}).then(operateResult);
```

Complex Queries 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 on if all fields, required in the query answer are available in the index.

Matching properties

Complex Queries select items which exactly match with the value given in the query. You can use wildcards ('*%*' is the default wildcard character), and you can change the wildcard character in the query options object. If you don't want to use a wildcard, just set the wildcard character to an empty string.

```
var query = {
  "query": 'creator:"* Doe"',
  "wildcard_character": "*"
};
```

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 their own default properties to keep their own behavior.

Convert Complex Queries into another type

Example, convert Query object into human readable string:

```
var query = complex_queries.QueryFactory.create('year: < 2000 OR title: "*a"'),
  option = {
    "wildcard_character": "*",
    "limit": [0, 10]
  },
  human_read = {
    "<": "is lower than ",
    "<=": "is lower or equal than ",
    ">": "is greater than ",
    ">=": "is greater or equal than ",
    "=": "matches ",
    "!=": "doesn't match "
  };

query.onParseStart = function (object, option) {
  object.start = "The wildcard character is '" +
    (option.wildcard_character || "%") +
    "' and we need only the " + option.limit[1] + " elements from the numero " +
    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: "The wildcard character is '*' and we need only the 10 elements
// from the numero 0. I want all document where year is lower than 2000 or title
// matches *a. Thank you!"
```

JSON Schemas and Grammar

Below you can find schemas for constructing complex queries

- Complex Queries JSON Schema
- Simple Queries JSON Schema
- Complex Queries Grammar

For developers

Quick start

To get started with jIO, clone one of the repositories link inDownload & Fork tab.

To build your library you have to:

- Install NodeJS (including NPM)
- Install Grunt command line with npm. # npm -g install grunt-cli
- Install dev dependencies. $ npm install
- Compile JS/CC parser. $ make (until we found how to compile it with grunt)
- And run build. $ grunt

The repository also includes the built ready-to-use files, so in case you do not want to build jIO yourself, just use *jio.js* as well as *complex_queries.js* plus the storages and dependencies you need and you will be good to go.

Naming Conventions

All the code follows this Javascript Naming Conventions.

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");
  }
}
```

Create 10 methods: post, put, putAttachment, get, getAttachment, remove, removeAttachment, allDocs, check and repair .

```
MyStorage.prototype.post = function (command, metadata, option) {
  var document_id = metadata._id;
  // [...]
};

MyStorage.prototype.get = function (command, param, option) {
  var document_id = param._id;
  // [...]
};

MyStorage.prototype.putAttachment = function (command, param, option) {
  var document_id = param._id;
  var attachment_id = param._attachment;
  var attachment_data = param._blob;
  // [...]
};

// [...]
```

(To help you to design your methods, some tools are provided by JIO.util.)

The first parameter command provides some methods to act on the JIO job:

- success, to tell JIO that the job is successfully terminated

  command.success(status[Text], [{custom key to add to the response}]);

- resolve, is equal to success
- error, to tell JIO that the job cannot be done

  command.error(status[Text], [reason], [message], [{custom key to add to the response}])

- retry, to tell JIO that the job cannot be done now, but can be retried later. (same API than error)
- reject, to tell JIO that the job cannot be done, let JIO to decide to retry or not. (same API than error)

The second parameter metadata or param is the first parameter given by the JIO user.

The third parameter option is the option parameter given by the JIO user.

Detail of what should return a method:

- post --> success("created", {"id": new_generated_id})
- put, remove, putAttachment or removeAttachment --> success(204)
- get --> success("ok", {"data": document_metadata})
- getAttachment -->

  ```
  success("ok", {"data": binary_string, "content_type": content_type})
  // or
  success("ok", {"data": new Blob([data], {"type": content_type})})
  ```

- allDocs --> success("ok", {"data": row_object})
- check -->

  ```
  // if metadata provides "_id" -> check document state
  // if metadata doesn't promides "_id" -> check storage state
  success("no_content")
  // or
  error("conflict", "corrupted", "incoherent document or storage")
  ```

- repair -->

  ```
  // if metadata provides "_id" -> repair document state
  // if metadata doesn't promides "_id" -> repair storage state
  success("no_content")
  // or
  error("conflict", "corrupted", "impossible to repair document or storage")
  // DON'T DESIGN STORAGES IF THEIR IS NO WAY TO REPAIR INCOHERENT STATES
  ```

After setting up all methods, your storage must be added to jIO. This is done using the `jIO.addStorage()` method, which requires two parameters: the storage type (string) add 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. Also keep in mind, that jIO is a job-based library, so whenever you trigger a method, a job is created, which after being processed returns a response.

Job rules

jIO job manager will follow several rules set at the creation of a new jIO instance. When you try to call a method, jIO will create a job and will make sure the job is really necessary and will be executed. Thanks to these job rules, jIO knows what to do with the new job before adding it to the queue. You can add your own rules like this:

These are the jIO **default rules**:

```
var jio_instance = jIO.createJIO(storage_description, {
  "job_rules": [{
    "code_name": "readers update",
    "conditions": [
      "sameStorageDescription",
      "areReaders",
      "sameMethod",
      "sameParameters",
      "sameOptions"
    ],
    "action": "update"
  }, {
    "code_name": "metadata writers update",
    "conditions": [
      "sameStorageDescription",
      "areWriters",
      "useMetadataOnly",
      "sameMethod",
      "haveDocumentIds",
      "sameParameters"
    ],
    "action": "update"
  }, {
    "code_name": "writers wait",
    "conditions": [
      "sameStorageDescription",
      "areWriters",
      "haveDocumentIds",
      "sameDocumentId"
    ],
    "action": "wait"
  }]
});
```

The following actions can be used:

- `ok` - accept the job
- `wait` - wait until the end of the selected job
- `update` - bind the selected job to this one
- `deny` - reject the job

The following condition function can be used:

- `sameStorageDescription` - check if the storage descriptions are different.
- `areWriters` - check if the commands are `post`, `put`, `putAttachment`, `remove`, `removeAttachment`, or `repair`.
- `areReaders` - check if the commands are `get`, `getAttachment`, `allDocs` or `check`.
- `useMetadataOnly` - check if the commands are `post`, `put`, `get`, `remove` or `allDocs`.
- `sameMethod` - check if the commands are equal.
- `sameDocumentId` - check if the document ids are equal.
- `sameParameters` - check if the metadata or param are equal in deep.
- `sameOptions` - check if the command options are equal.
- `haveDocumentIds` - test if the two commands contain document ids

Create Job Condition

You can create 2 types of function: job condition, and job comparison.

```
// Job Condition
// Check if the job is a get command
jIO.addJobRuleCondition("isGetMethod", function (job) {
  return job.method === 'get';
});

// Job Comparison
// Check if the jobs have the same 'title' property only if they are strings
jIO.addJobRuleCondition("sameTitleIfString", function (job, selected_job) {
  if (typeof job.kwargs.title === 'string' &&
      typeof selected_job.kwargs.title === 'string') {
    return job.kwargs.title === selected_job.kwargs.title;
  }
  return false;
});
```

Add job rules

You just have to define job rules in the jIO options:

```
// Do not accept to post or put a document which title is equal to another
// already running post or put document title
var jio_instance = jIO.createJIO(storage_description, {
  "job_rules": [{
    "code_name": "avoid similar title",
    "conditions": [
      "sameStorageDescription",
      "areWriters",
      "sameTitleIfString"
    ],
    "action": "deny",
    "before": "writers update" // optional
    // "after": also exists
  }]
});
```

Clear/Replace default job rules

If a job which `code_name` is equal to `readers update`, then add this rule will replace it:

```
var jio_instance = jIO.createJIO(storage_description, {
  "job_rules": [{
    "code_name": "readers update",
    "conditions": [
      "sameStorageDescription",
      "areReaders",
      "sameMethod",
      "haveDocumentIds"
      "sameParameters"
      // sameOptions is removed
    ],
    "action": "update"
  }]
});
```

Or you can just clear all rules before adding other ones:

```
var jio_instance = jIO.createJIO(storage_description, {
  "clear_job_rules": true,
  "job_rules": [{
    // ...
  }]
});
```

Authors

- Francois Billioud
- Tristan Cavelier
- Sven Franck

Copyright and license

Last update: Thu Oct 31 2013