johoDB Documentation

Release 0.2.0

James Rakich

June 18, 2014

Contents

CHAPTER 1

Preamble

JohoDB is an abstraction library strictly for browser supported databases. It allows the developer to build a relational database for use on the client, completely independant of any server calls. JohoDB currently supports IndexedDB and WebSQL, which allows it to support the following browsers:

1.1 Desktop Browsers

- Safari 3.1+
- Opera 10.5+
- Chrome 4.0+
- Firefox 16.0+
- Internet Explorer 10.0+

1.2 Mobile/Tablet Browsers

- iOS Safari 3.2+
- Android 2.1+
- Chrome for Android 35+
- Blackberry 7.0+
- Opera Mobile 11.0+
- Firefox for Android 29.0+
- IE Mobile 10.0+

Local storage is not and **will never be** supported by JohoDB. It is merely a key value store and any attempt at implementation would be horribly inefficient, and looking to the future this browser support level is adequate for the purposes JohoDB is meant for, namely complex web applications.

CHAPTER 2

Quickstart

2.1 Installing JohoDB

JohoDB can be run on the client using a prebuilt javascript file, or bundled in as an npm package.:

```
npm install johodb
```

Once installed you can require JohoDB in your source code with:

```
var JohoDB = require('johodb');
```

Get started with your database with:

```
var db = new JohoDB('YOUR_DATABASE_NAME');
```

From here you'll need to add your schema. Learn to do this at *Schema and Migrations*.

Todo

Installation as a linked javascript file.

2.2 Schema and Migrations

JohoDB is a library that creates a *relational* database structure, and as such you must create a Schema. For those looking for a NoSQL approach to data there are other libraries out there, but you won't be able to support WebSQL with them. Anyway, lets get to the Schema.

2.2.1 Building the Schema

After creating your JohoDB object, but before running *db.init()* you need to add your schema. A basic schema looks like this:

```
var db = new JohoDB('YOUR_DATABASE_NAME');

db.addSchema('Person', {
    'id': {type: "string", primaryKey: true},
    'name': {type: "string", required: true},
    'age': {type: "int"}
});
```

```
8
9  db.addSchema('Message', {
10     'id': {type: "string", primaryKey: true},
11     'text': {type: "string", required: true},
12     'author': {type: "fk", required: true,
13                relation: "Person", relatedName: "messages"}
14 });
```

As you can see, we're using the addSchema method to build up our schema.

db.**addSchema**(*tableName*, *tableSchema*)

Arguments

- **tableName** (*string*) – The name of the table this schema represents.
- **tableSchema** (*object*) – The schema of the table, including all fields, their names and properties.

The Table Schema

The table schema is an object literal, with keys representing the names of the fields. The values are the attributes of that field, here are the basics you'll need:

type The type of value the field represents. Basics include *string*, *int*, and *fk* for a relation.

primaryKey Whether or not the field is the primary key for this table. Be aware that JohoDB *does not* create keys [1].

required The field is required to save a record. Primary keys are required by default.

relation For fields that are relations, the name of the table it is a relation too.

relatedName For fields that are relations, this is the name the other table should refer to it's related records by.

See the remaining attributes and field types available at: *Field Reference*

Notes for building a Schema

- You must define the entire schema before running *db.init()*.
- You can define the schema in any order. JohoDB will not attempt to validate the schema until you run *db.init()*.

2.2.2 Migrations

Once you've set up your schema, you might be tempted to run *db.init()*, however you'll run into a problem. You'll see this warning in your console:

```
1 UPDATED MIGRATION REQUIRED:
2
3 db.addMigration({ actions: [
4     new JohoDB.MigrationAction("add_table", {"tableName":"Person"}),
5     new JohoDB.MigrationAction("add_column", {"tableName":"Person","columnName":"id","columnTraits":
6     new JohoDB.MigrationAction("add_column", {"tableName":"Person","columnName":"name","columnTraits"
```

[1] JohoDB does not create primary keys for you automatically, the reason being it is an asynchronous library whether you use WebSQL or IndexedDB. As such it is better you take responsibility for key generation so you're not running into race conditions, whether it's a pseudo-UUID or some other system.

```
    new JohoDB.MigrationAction("add_column", {"tableName":"Person","columnName":"age","columnTraits"
    new JohoDB.MigrationAction("add_table", {"tableName":"Message"}),
    new JohoDB.MigrationAction("add_column", {"tableName":"Message","columnName":"id","columnTraits"
    new JohoDB.MigrationAction("add_column", {"tableName":"Message","columnName":"text","columnTraits
    new JohoDB.MigrationAction("add_column", {"tableName":"Message","columnName":"author","columnTra
]});
```

So what's happening here? JohoDB always checks on run time if the schema matches the migrations it has. It needs to know how to build the Schema it has been provided. This doesn't make much sense now, but is critical if you need to ship updates to the Schema down the track.

To make it easier on developers, if the migrations path provided (or lack of migrations in this case) does not arrive at a perfect match with the schema, it will provide you with a snippet that you can copy/paste into your code.

db.**addMigration**(*migrationObj*)

Arguments

- **migrationObj** (*object*) – An object with an attribute 'actions' which represents all the actions that must occurs within this migration.

Most of the time, you'll just be able to copy paste the code snippet in to some point before *db.init()*, but there's one thing to take care with: migration action order does matter in some key ways. Whenever you create a foreign key field, in WebSQL indexes will be created to the other table, so you need to ensure foreign key fields come after their related table is created.

Migrations and Versioning

Each migration represents a version of your database. When the database is connected with IndexedDB or WebSQL, the number of migrations is used to check whether the client is in or out of date. That way you can automatically upgrade their database, applying only the migrations that are required.

In order to do this versioning, you simply call *db.addMigration* multiple times with new actions each time. In the above example we could make the tables in two separate migrations like so:

```
db.addMigration({ actions: [
    new JohoDB.MigrationAction("add_table", {"tableName":"Person"}),
    new JohoDB.MigrationAction("add_column", {"tableName":"Person","columnName":"id","columnTraits":
    new JohoDB.MigrationAction("add_column", {"tableName":"Person","columnName":"name","columnTraits"
    new JohoDB.MigrationAction("add_column", {"tableName":"Person","columnName":"age","columnTraits"
});

db.addMigration({ actions: [
    new JohoDB.MigrationAction("add_table", {"tableName":"Message"}),
    new JohoDB.MigrationAction("add_column", {"tableName":"Message","columnName":"id","columnTraits"
    new JohoDB.MigrationAction("add_column", {"tableName":"Message","columnName":"text","columnTraits
    new JohoDB.MigrationAction("add_column", {"tableName":"Message","columnName":"author","columnTra
]});
```

Now, this would only matter if you shipped out a version of your app with only the Person table, and down the track decided to add the Message table. That said, hopefully it illustrates how migrations are intended to work.

Anyway, onward! Time to hook this up to a real database at *Initialising the Database*.

2.3 Initialising the Database

Once you've created your schema and have migrations to match, you're ready to connect to your database with:

db.**init**([*initSettings*])

:param object initSettings An object with optional settings for initialising the database.

Settings include:

storageType A Storage Adapter to use, if you want to override JohoDB's automatic detection. The only two available with JohoDB are: *JohoDB.WebSQLStorage* and *JohoDB.IndexedDBStorage*.

fieldRegistry A custom Field Registry to use. Only required if you need more field types than what JohoDB provides.

clobber Destroy the database if it exists and start from scratch. Good for development and testing.

dontLoadStore A testing setting that will stop JohoDB before it connects to the IndexedDB or WebSQL database.

While you can be specific here, for most use cases it's just fine to call *db.init()*.

JohoDB will do the following:

1. Go through the schema and build your API to save and query with, at *db.models*.
2. Connect to the Database (IndexedDB or WebSQL).
3. Check the database version (represented by number of migrations)
4. If the database is new, or older than the current version, apply migrations to bring it up to schema, creating tables and adding columns as required.

Once this is done, you will now be able to save and query data.

2.4 Saving and Querying Data

Now that you've initialised your database, you can now save and query data. All save and query operations are asynchronous, so you'll receive promises from any operation you start, which will eventually return your desired value or an error.

2.4.1 Saving Data

If we were using the schema from before, with a Person and a Message table, a save could look like this:

```
db.models.Person.save({
    'id': 'a-unique-id-of-some-sort',
    'name': 'Joe Bloggs',
    'age': 25
}).then(function(savedRecord) {
    // do something once the record is saved
}).catch(function(err) {
    // if you wish to catch any errors, throw catch on the end of
    // your promise chain
});
```

Model.**save**(*saveRecord*[, *options*])

Arguments

- **saveRecord** (*object*) – The object to be saved into the database. This can be any object, the only thing that matters is it has attributes matching the fields on the schema.

- **options** (*object*) – Options for the save operation, all are optional.

  updateRelations: Will update the keys for any relationship fields as required. This will ONLY update the keys, and not save the rest of the data in related records.

  updateRelatedRecords: Will update any related records for relationship fields if they are provided as objects or arrays.

When you use *save* whatever you pass to it will be validated against the schema and then saved in the data store. You can also just do a validation if you want by using *validate* instead in the same way.

`Model.validate(saveRecord[, options])`

Arguments

- **saveRecord** (*object*) – The object to be validated.
- **options** (*object*) – Options for the validation operation.

Returns Promise<validationResult> Promise resolves on end of validation.

The Validation Result returns the following information:

- **isValid**: is true if the record passed validation
- **errorCount**: number of invalid fields, if any
- **validating**: a reference to the validated record
- **errors**: an object with any field errors attached

2.4.2 Querying Data

Querying the data is done by building a query from chained methods and then evaluating that query. You can start a query by calling *.query()* from a model. An example where we get all of the available records:

```
db.models.Person.query().all().evaluate().then(function(records) {
    // do something with the records
});
```

If you want to get a single record you can use a get method in the query:

```
db.models.Person.query().get({
    'name': 'Joe Bloggs'
}).evaluate().then(function(record) {
    // do something with the record, if not found record will be null
});
```

If you want to get multiple records based on a query, you can use *filter* or *exclude*:

```
// get all persons with an age of 25
db.models.Person.query().filter({'age': 25}).evaluate();

// get all persons above the age of 25
db.models.Person.query().filter({'age__gt': 25}).evaluate();

// exclude all persons below of equal to the age of 25
db.models.Person.query().exclude({'age__lte': 25}).evaluate();
```

The object you pass to a *get*, *filter*, or *exclude* function contains the fields you want to query by. Providing the field name alone means you wish for an exact match to the value, otherwise you can provide a modifier after two underscores to do a different type of match.

__gt Greater than the query value.

__gte Greater than or equal to the query value.

__lt Less than the query value.

__lte Less than or equal to the query value.

__in Equals one of the query value, which must be an array.

Note: If you wish to query directly after a save operation, you should delay by a moment to ensure that the saved record shows up in the query. You can delay acting on a promise by calling *.delay(10)*.

CHAPTER 3

User API

3.1 Field Reference

See relational fields at: *Relations Reference*

3.1.1 Field Options

The following options are available to all field types and are optional.

primaryKey

If *true*, this field will be the primary key for identifying records in the table. ManyToMany tables will use this primary key as it's point of reference.

All tables must have at least **ONE** primary key.

required

If *true*, this field MUST be defined as something other than *null* or *undefined* when saved.

3.1.2 Field Types

Fields are specified by *type* in the schema. The *type* is case insensitive, and in some cases allow for abbreviated versions of the field's formal name.

String

Matching Types: str, string, text

A string field, for small to large sized strings. This is the only field type you can use for representing a primary key, which ideally would be similar to a UUID in structure to prevent collisions.

Integer

Matching Types: int, integer

An integer field, returned and saved as a Javascript number.

Boolean

Matching Types: bool, boolean

A boolean field, representing a *true*, *false* or *undefined* value.

JSON

Matching Types: json

A field that can represent any Javascript object literal. When stored in the database it will be serialised using *JSON.stringify()* and retrieved using *JSON.parse()*.

JSON field types are not indexed, so their ideal use is as a non searchable element of your database.

DateTime

Matching Types: datetime

A field representing a DateTime. Can be saved as any value that can be parsed by the momentjs.com library, however it will ALWAYS be returned as a moment for continuity. Timezone information is maintained when saving.

Date

Matching Types: date

A field representing a Date with no timezone information. As with DateTime, it is wrapped with momentjs.com when returned.

Time

Matching Types: time

A field representing a Time with timezone information. As with DateTime, it is wrapped with momentjs.com when returned.

3.2 Relations Reference

Relationships are established in the schema just like fields, but once created there is a little more to them.

3.2.1 Relation Options

The following options are available to all relation types.

relation

The name of the table this relation will be targeted at.

relatedName

The field the opposite table will use to refer to this relation.

onDelete

The behaviour that will occur when a relation subject is deleted. This defaults to *cascade*.

cascade Deletes will cascade down, where deleting the target of a foreign key relationship will also delete all records that link to it.

setNull When deleting the target of a foreign key relationship, all related records will have their foreign key field set to null.

3.2.2 Relation Types

Relations are specified by *type* in the schema. The *type* is case insensitive, and in some cases allow for abbreviated versions of the relations's formal name.

ForeignKey

Matching Types: fk, foreignkey

The table this relation is given to can be associated to one record of the target table by using their primary key.

The target table will be able to collect all records that have it as it's foreign key by using the name specified as the *relatedName*.

ManyToMany

Matching Types: many2many, m2m, manytomany

This table can be associated to as many of the relation table's records as you like, through a joining table that is automatically created by the underlying data store.

3.2.3 Saving Relations

When using the save method, you must set an option making it clear you wish to update the relationships of this record. An example:

```
db.models.Household.save({
    id: "thisHouseID",
    address: "52 Goldilock Road",
    suburb: "Manjimup",
    postCode: "8732",
    occupants: [
      {id: "jrakich", firstName: "James", lastName: "Rakich", dateOfBirth: new Date(1952, 10, 12)},
      {id: "joebloggs", firstName: "Joe", lastName: "Bloggs", dateOfBirth: new Date(1982, 9, 2)}
    ]
}, {updateRelatedRecords: true});
```

In this case the *Household* is the subject of a foreign key relationship from the *Person* table, with *occupants* as the *relatedName*. Because we've set *updateRelatedRecords* to true, this save will *also* save/update the occupants and set their foreign key to the correct id (*thisHouseID* in this case).

If you want to *only* update the foreign keys, without affecting the rest of the relation's data, set the option to *updateRelations* instead, and it won't touch the rest of the data, just the foreign key.

The inverse of this, with a *Person*, could look like this:

```
db.models.Person.save({
    id: "jrakich",
    firstName: "James",
    lastName: "Rakich",
    dateOfBirth: new Date(1952, 10, 12)},
    home: {
        id: "thisHouseID",
        address: "52 Goldilock Road",
        suburb: "Manjimup",
        postCode: "8732"
    }
}, {updateRelatedRecords: true});
```

Here the person would be saved, and so would their household, with the appropriate foreign key in it's place. If the household already existed this would be equivalent:

```
db.models.Person.save({
    id: "jrakich",
    firstName: "James",
    lastName: "Rakich",
    dateOfBirth: new Date(1952, 10, 12)},
    home: "thisHouseID"
});
```

As you can see in this case we can directly set the foreign key if we wish.

3.3 Model API

`Model.`**`validate`**(*validationRecord*[, *options*])

Validates the record, returning a validation result.

Arguments

- **saveRecord** (*object*) – The object to be saved into the database. This can be any object, the only thing that matters is it has attributes matching the fields on the schema.
- **options** (*object*) – Options for the save operation.

Returns Promise<validationResult> Promise resolves on end of validation.

The Validation Result returns the following information:

- **isValid**: is true if the record passed validation
- **errorCount**: number of invalid fields, if any
- **validating**: a reference to the validated record
- **errors**: an object with any field errors attached

`Model.`**`save`**(*saveRecord*[, *options*])

Save the record to the database.

Arguments

- **saveRecord** (*object*) – The object to be saved into the database. This can be any object, the only thing that matters is it has attributes matching the fields on the schema.
- **options** (*object*) – Options for the save operation, all are optional.

  updateRelations: Will update the keys for any relationship fields as required. This will ONLY update the keys, and not save the rest of the data in related records.

  updateRelatedRecords: Will update any related records for relationship fields if they are provided as objects or arrays.

Returns Promise<saveRecord> Promise resolves on full completion of saving, including all extra record saves.

`Model.delete`(*deletionRecord*[, *options*])
Delete a record from the database.

Arguments

- **deleteRecord** (*object*) – The object to be deleted from the database. This can be any object, the only thing that matters is that it has a primary key that corresponds to the schema.
- **options** (*object*) – Placeholder argument, there are no delete options at this time.

Returns Promise<saveRecord> Promise resolves on full completion of delete, including cascading deletes or nulling of foreign keys.

`Model.query`()
Begin a query object, for querying the database. See *Query API* for more details. Queries can also be immediately created with a first step by calling the relevant method first, for example, *Model.filter(args)*.

Returns Query

3.4 Query API

Queries are an object created from the Model API. Once created, you can add a number of steps to the query to get your desired results from the database. Queries are only executed after calling *Query.evaluate()*, which will return a promise with the appropriate results.

`Query.all`()
Add a step to the query that collects all records.

`Query.get`(*lookupArgs*)
Add a step to the query that collects a single record based on the lookup arguments.

Arguments

- **lookupArgs** (*object*) – An object literal with lookup arguments.

`Query.filter`(*lookupArgs*)
Add a step to the query that filters records based on the lookup arguments.

Arguments

- **lookupArgs** (*object*) – An object literal with lookup arguments.

`Query.exclude`(*lookupArgs*)
Add a step to the query that excludes records based on the lookup arguments.

Arguments

- **lookupArgs** (*object*) – An object literal with lookup arguments.

`Query.`**`evaluate`**`(`[*options*]`)`

Evaluate the query as built and return the record[s] collected in a promise.

Arguments

- **options** (*object*) – Options are provided as an object, but are not required. For now there is only one option available:

  depth: Decide how deeply the evaluation will go to collect relations. By default this argument is set at 1 and will collect the nearest relations only. Raising the depth can greatly increase the execution time and result in circular record collection, which is not optimised at present.

Return Promise<records> Promise resolves once the records and all depth recovered records are collected.

3.4.1 Lookup Arguments

Most query steps will require lookup arguments to be used. The lookup arguments are an object literal with keys that define which field to target, and values which define what to match against. By default lookup arguments query for an exact match, but you can also provide modifiers as a postfix to the fields, for example *age__gt* for the everything greater than on the field *age*.

Modifiers

__gt Greater than the query value.

__gte Greater than or equal to the query value.

__lt Less than the query value.

__lte Less than or equal to the query value.

__in Equals one of the query value, which must be an array.

Relational Queries

It is possible to perform lookups based on the fields of related objects. For example, if you had a person record with a relation *home*, you could query against it like so:

```
// filter all people who live in a home in the suburb 'Moonah'
db.models.Person.query().filter({'home__suburb': 'Moonah'}).evaluate();
```

Field names must be separated with two underscores. You can add modifiers to the end.

CHAPTER 4

Indices and tables

- *search*