
Store.js Documentation

JOOR Inc.

Oct 27, 2018

Contents:

1	Canonical repo	3
2	Edge (unstable) repo	5
3	Articles	7
4	Features	11
5	Implemented Patterns	13
6	Used programming paradigms	15
7	Store	17
7.1	Store public API	17
7.2	Store events	20
7.3	Store observers	20
8	Result	23
8.1	Result public API	23
8.2	Result events	24
9	Registry	25
9.1	Registry public API	25
9.2	Registry events	26
9.3	Registry observers	26
10	Observable Interface	29
11	StoreObservable Interface	31
12	Result Observable Interface	33
13	Query Object	35
13.1	Comparison operators	35
13.2	Logical operators	36
13.3	Relational operators	36
13.4	Query Modifiers	37

14 Examples	39
14.1 Query	39
14.2 Simple relations	41
14.3 Composite relations	43
14.4 Many to many relations	45
14.5 Compose	47
14.6 Decompose	50
14.7 Observable object	53
14.8 StoreObservable	55
14.9 Reaction of Result on changes in Store	56
15 Contributing	65
16 Indices and tables	67

Store.js is a super lightweight implementation of [Repository](#) pattern for relational data and aggregates. The library allows you to use Domain-Driven Design (DDD) on client-side as well as reactive programming.

This is similar to Object-Relational Mapping (ORM) for JavaScript, including the Data Mapper pattern (the data can be mapped between objects and a persistent data storage).

CHAPTER 1

Canonical repo

- Home Page and Source Code: <https://github.com/joor/store-js-external>
- Docs: TODO

CHAPTER 2

Edge (unstable) repo

- Home Page and Source Code: <https://github.com/emacsway/store>
- Docs: <https://edge-storejs.readthedocs.io/>

- Article (in English) “Implementation of the pattern Repository for browser’s JavaScript”
- Article (in Russian): “ Repository JavaScript”

Contents

- *Welcome to Store.js’ documentation!*
 - *Canonical repo*
 - *Edge (unstable) repo*
 - *Articles*
- *Features*
- *Implemented Patterns*
- *Used programming paradigms*
- *Store*
 - *Store public API*
 - *Store events*
 - * *Events by ObservableStoreAspect*
 - * *Store events by PreObservableStoreAspect*
 - *Store observers*
- *Result*
 - *Result public API*
 - *Result events*
- *Registry*

- *Registry public API*
 - *Registry events*
 - *Registry observers*
- *Observable Interface*
- *StoreObservable Interface*
- *Result Observable Interface*
- *Query Object*
 - *Comparison operators*
 - * *\$eq*
 - * *\$ne*
 - * *\$in*
 - * *\$callable*
 - *Logical operators*
 - * *\$and*
 - * *\$or*
 - *Relational operators*
 - * *\$rel*
 - *Query Modifiers*
 - * *\$query*
 - * *\$orderby*
 - * *\$limit*
 - * *\$offset*
- *Examples*
 - *Query*
 - *Simple relations*
 - *Composite relations*
 - *Many to many relations*
 - *Compose*
 - *Decompose*
 - *Observable object*
 - *StoreObservable*
 - *Reaction of Result on changes in Store*
- *Contributing*
- *Indices and tables*

The `IStore()` class is a super lightweight implementation of [Repository](#) pattern for relational data and composed nested aggregates. The main goal of [Repository](#) pattern is to hide the data source.

The `IStore()` class has simple interface, so, this abstract layer allows you easy to change the policy of data access. For example, you can use as data source:

- [REST API](#)
- [CORS REST API](#)
- [JSON-RPC](#)
- [html](#)
- [Indexed Database API](#)
- etc.

An essential attribute of Repository pattern is the pattern [Query Object](#), which is necessary to hide the data source. This class was developed rapidly, in limited time, thus there is used the simplest query syntax similar to [MongoDB Query](#).

CHAPTER 4

Features

- Store is easy to debug, since its code is written with a [KISS principle](#), and thus is easy to understand.
- Store handles composed primary keys and composite relations with ease (no need for surrogate keys).
- Store supports cascade deleting and updating with changeable cascade behavior.
- Store uses event system extensively.
- Store has reactive result which synchronizes his state when the observed subject (store or parent result collection) is changed.
- Store has easy query syntax similar to [MongoDB Query](#).
- Store allows you to keep models FULLY clean without any service logic, - only business rules. This is an important point when you use [DDD](#), thus your product team (or customer) will be able to read the business rules from code.
- Store allows you to work with stream of composed aggregates easily, regardless of the depth of nesting of aggregates. See method `Store.prototype.decompose()`.
- Store allows you to compose composed aggregates from stores using information about relations. See method `Store.prototype.compose()`.
- Store has implemented pattern [Identity Map](#), thus you can easily to work with model instances by [reference](#). You always will have the single instance of entity in a memory.
- Store does not have any external dependencies except [RequireJS](#).
- Written in ES3 and should be fully compatible with ES3 (not really tested).

Implemented Patterns

- Repository
- Query Object
- Identity Map
- Data Mapper
- Gateway
- Unit Of Work
- Observer
- Mediator
- Adapter

Used programming paradigms

- Reactive Programming
- Event-driven programming
- Aspect-oriented programming (Cross-Cutting Concerns)
- Declarative programming

7.1 Store public API

```
class Store ([options])
```

Arguments

- **options** (*Object*) – the keyword arguments.

The `options` object can have the next keys:

Arguments

- **options.pk** (*string or Array[string]*) – the name of Primary Key or list of names of composite Primary Key.Optional. The default value is 'id'.
- **options.objectAccessor** (*ObjectAccessor*) – an instance of `ObjectAccessor()`. Optional. By default will be created on fly using `options.pk`.
- **options.indexes** (*Array[string]*) – the array of field names to be indexed for fast finding or instance of local store.Note, all field used by relations or primary key will be indexed automatically.Optional.
- **options.remoteStore** (*IStore*) – an instance of `IStore()`. Optional.By default will be created on fly using `options`
- **options.model** (*function*) – the model constructor, which should be applied before to add object into the store.Can be usefull in combination with `Store.prototype.decompose()`.Optional. The default value is `DefaultModel()`
- **options.serializer** (*Serializer*) – an instance of `Serializer()`. Optional.By default will be created on fly using `options.model`
- **options.relations** (*Object*) – the dictionary describes the schema relations.

The format of `options.relations` argument:

```

{
  foreignKey: {
    firstForeignKeyName: {
      [field: fieldNameOfCurrentStore,] // (string | Array[string]),
      // optional for Fk, in this case the relation name will be used.
↪as field name
      relatedStore: nameOfRelatedStore, // (string)
      relatedField: fieldNameOfRelatedStore, // (string | Array[string])
      [onAdd: callableOnObjectAdd,] // (function) compose
      [onDelete: callableOnObjectDelete,] // (function) cascade|setNull
      [onUpdate: callableOnObjectUpdate,] // (function)
    },
    secondForeignKeyName: ...,
    ...
  },
  [oneToMany: {
    firstOneToManyName: {
      field: fieldNameOfCurrentStore, // (string | Array[string]),
      relatedStore: nameOfRelatedStore, // (string)
      relatedField: fieldNameOfRelatedStore, // (string | Array[string])
      [relatedName: nameOfReverseRelationOfRelatedStore,]
      [onAdd: callableOnObjectAdd,] // (function)
      [onDelete: callableOnObjectDelete,] // (function)
↪cascade|setNull|decompose
      [onUpdate: callableOnObjectUpdate,] // (function)
    },
    secondOneToManyName: ...,
    ...
  },]
  manyToMany: {
    firstManyToManyName: {
      relation: relationNameOfCurrentStore, // (string)
      // the name of foreignKey relation to middle M2M store.
      relatedStore: nameOfRelatedStore, // (string)
      relatedRelation: relationNameOfRelatedStore, // (string)
      // the name of oneToMany relation from related store to middle.
↪M2M store.
      [onAdd: callableOnObjectAdd,] // (function) compose
      [onDelete: callableOnObjectDelete,] // (function)
↪cascade|setNull|decompose
      [onUpdate: callableOnObjectUpdate,] // (function)
    },
    secondManyToManyName: ...,
    ...
  }
}

```

If `oneToMany` is not defined, it will be built automatically from `foreignKey` of related store. In case the `foreignKey` don't has `relatedName` key, a new `relatedName` will be generated from the store name and "Set" suffix.

If `options.objectAccessor` is provided, the `options.pk` will be ignored.

If `options.serializer` is provided, the `options.model` and `options.objectAccessor` will be ignored.

If `options.localStorage` is provided, the `options.indexes` will be ignored.

The public method of Store:

`Store.Store.prototype.pull (query, options)`
Populates local store from remote store.

Arguments

- **query** (*Object*) – the Query Object.
- **options** (*Object*) – options to be passed to the remote store.

Return type `Promise<Array[Object], Error>`

`Store.Store.prototype.get (pkOrQuery)`
Retrieves a Model instance by primary key or by Query Object.

Arguments

- **pkOrQuery** (*number or string or Array or Object*) – the primary key of required Model instance or Query Object.

`Store.Store.prototype.add (obj)`
Adds a Model instance into the Store instance.

Arguments

- **obj** (*Object*) – the Model instance to be added.

Return type `Promise<Object, Error>`

`Store.Store.prototype.update (obj)`
Updates a Model instance in the Store instance.

Arguments

- **obj** (*Object*) – the Model instance to be updated.

Return type `Promise<Object, Error>`

`Store.Store.prototype.save (obj)`
Saves a Model instance into the Store instance. Internally the function call will be delegated to `Store.prototype.update ()` if `obj` has primary key, else to `Store.prototype.add ()`

Arguments

- **obj** (*Object*) – the Model instance to be saved.

Return type `Promise<Object, Error>`

`Store.Store.prototype.delete (obj)`
Deletes a Model instance from the Store instance.

Arguments

- **obj** (*Object*) – the Model instance to be deleted.

Return type `Promise<Object, Error>`

`Store.Store.prototype.find (query)`
Returns a `Result ()` instance with collection of Model instances meeting the selection criteria.

Arguments

- **query** (*Object*) – the Query Object.

`Store.Store.prototype.compose (obj)`
Builds a nested hierarchical composition of related objects with the `obj` top object. Example: *Compose*.

Arguments

- **obj** (*Object*) – the Model instance to be the top of built nested hierarchical composition

`Store.Store.prototype.decompose(obj)`

Populates related stores from the nested hierarchical composition of related objects. Example: *Decompose*.

Arguments

- **obj** (*Object*) – the nested hierarchical composition of related objects with the `obj` top object

`Store.Store.prototype.observd()`

Returns the *StoreObservable()* interface of the store.

Return type StoreObservable

The service public methods (usually you don't call these methods):

`Store.Store.prototype.register(name, registry)`

`Store.Store.prototype.destroy()`

`Store.Store.prototype.clean()`

7.2 Store events

7.2.1 Events by ObservableStoreAspect

Event	When notified
“add”	on object is added to store, triggered by <code>Store.prototype.add()</code>
“update”	on object is updated in store, triggered by <code>Store.prototype.update()</code>
“delete”	on object is deleted from store, triggered by <code>Store.prototype.delete()</code>
“restoreObject”	on object is restored, triggered by <code>Store.prototype.delete()</code>
“destroy”	immediately before store is destroyed, triggered by <code>Store.prototype.destroy()</code> Usually used to kill reference cycles .

7.2.2 Store events by PreObservableStoreAspect

Event	When notified
“preAdd”	before object is added to store, triggered by <code>Store.prototype.add()</code>
“preUpdate”	before object is updated in store, triggered by <code>Store.prototype.update()</code>
“preDelete”	before object is deleted from store, triggered by <code>Store.prototype.delete()</code>

7.3 Store observers

Store functional-style observer signature:

storeObserver (*aspect, obj*)

this variable inside observer is setted to the notifier `IStore()` instance.

Arguments

- **aspect** (*string*) – the event name
- **obj** (*Object*) – the Model instance.

Store OOP-style Observer interface:

```
class IStoreObserver ()
```

```
IStoreObserver.update (subject, aspect, obj)
```

Arguments

- **subject** (*Store*) – the notifier
- **aspect** (*string*) – the event name
- **obj** (*Object*) – the Model instance.

An observer of the events “update” has one extra argument “oldObjectState”.

8.1 Result public API

class Result (*subject, reproducer, objectList* [, *relatedSubjects*])

The Result is a subclass of Array (yes, a composition would be better than the inheritance, but it was written by ES3).

Arguments

- **subject** (*Store*) – the subject of result
- **reproducer** (*function*) – the reproducer of actual state of result
- **objectList** (*Array[Object]*) – the list of model instances
- **relatedSubjects** (*Array[Store]*) – the list of subjects which can affect the result

`Result.Result.prototype.observe` (*enabled*)

Makes observable the result, and attaches it to its subject.

Arguments

- **enabled** (*Boolean or undefined*) – if enabled is false, the all observers of the result will be detached form its subject.

Return type

 Result

`Result.Result.prototype.observd` ()

Returns the *ResultObservable* () interface of the result.

Return type

 ResultObservable

`Result.Result.prototype.addRelatedSubject` (*relatedSubject*)

Adds subject on which result should be dependent.

Arguments

- **relatedSubject** (*Array[Store or Result or SubResult]*) – the subject on which result should be dependent

Return type `Result`

8.2 Result events

Event	When notified
“add”	on object is added to result
“update”	on object is updated in result
“delete”	on object is deleted from result

An observer of the event “update” has one extra argument “oldObjectState”.

class `SubResult` (*subject*, *reproducer*, *objectList* [, *relatedSubjects*])

The `SubResult` is a subclass of `Result` (). The difference is only the subject can be `Result` or another `SubResult`.

Arguments

- **subject** (*Result* or *SubResult*) – the subject of result
- **reproducer** (*function*) – the reproducer of actual state of result
- **objectList** (*Array[Object]*) – the list of model instances
- **relatedSubjects** (*Array[Result* or *SubResult]*) – the list of subjects which can affect the result

9.1 Registry public API

class Registry()

The Registry class is a **Mediator** between *stores* and has goal to lower the **Coupling**. The public methods of Registry:

`Registry.register(name, store)`

Links the `IStore()` instance and the `Registry()` instance.

Arguments

- **name** (*string*) – the name of `IStore()` instance to be registered. This name will be used in relations to the store from related stores.
- **store** (*Store*) – the instance of `IStore()`

`Registry.Registry.prototype.has(name)`

Returns true if this store name is registered, else returns false.

Arguments

- **name** (*string*) – the name of `IStore()` instance the presence of which should be checked.

Return type Boolean

`Registry.Registry.prototype.get(name)`

Returns `IStore()` instance by name.

Arguments

- **name** (*string*) – the name of `IStore()` instance the presence of which should be checked.

Return type Store

`Registry.Registry.prototype.getStores()`

Returns mapping of name and `IStore()` instances

`Registry.Registry.prototype.keys()`

Returns list of names.

Return type Array[String]

`Registry.Registry.prototype.ready()`

Notifies the attached observers that all stores are registered. Usually used to attach observers of registered *stores* one another.

`Registry.Registry.prototype.begin()`

Delays save objects by remote storage until `Registry.prototype.commit()` will be called.

`Registry.Registry.prototype.commit()`

Runs delayed saving for all objects which has been added, updated, deleted since `Registry.prototype.begin()` has been called.

`Registry.Registry.prototype.rollback()`

Discards all uncommitted changes since `Registry.prototype.begin()` has been called.

`Registry.Registry.prototype.destroy()`

Notifies the attached observers when the data will be destroyed. The method calls `Store.prototype.destroy()` method for each registered store.

`Registry.Registry.prototype.clean()`

Cleans all registered *stores*.

`Registry.Registry.prototype.observable()`

Returns the *Observable()* interface of the registry.

Return type Observable

9.2 Registry events

Event	When notified
“register”	on store registered
“ready”	on all stores are registered
“begin”	on begin of transaction
“commit”	on commit of transaction
“rollback”	on rollback of transaction
“destroy”	on all data will be destroyed

9.3 Registry observers

Registry functional-style observer signature:

registryObserver (*aspect*, *store*)

this variable inside observer is setted to the notifier *Registry()* instance.

Arguments

- **aspect** (*string*) – the event name
- **store** (*Store*) – the *IStore()* instance. This argument is omitted for “ready” event.

Registry OOP-style Observer interface:

class `IRegistryObserver()`

`IRegistryObserver.update(subject, aspect, store)`

Arguments

- **subject** (`Registry`) – the notifier
- **aspect** (`string`) – the event name
- **store** (`Store`) – the `IStore()` instance. This argument is omitted for “ready” event.

Observable Interface

class Observable (*obj*)

Creates an observable interface for object.

Arguments

- **obj** (*Object*) – the object to be observable

`Observable.Observable.prototype.set` (*name, newValue*)

Sets the new value of attribute of the object by the name of the attribute.

Arguments

- **name** (*string*) – the name of the object attribute to be updated
- **newValue** – the new value of the object attribute

`Observable.Observable.prototype.get` (*name*)

Returns the current value of the object attribute by name.

Arguments

- **name** (*string*) – the name of the object attribute

`Observable.Observable.prototype.attach` (*[aspect], observer*)

Attaches the observer to the specified aspect(s). If aspect is omitted, the observer will be attached to the global aspect which is notified on every aspect. Returns instance of `Disposable()`. So, you can easily detach the attached observer by calling the `Disposable.prototype.dispose()`.

Arguments

- **aspect** (*string or Array[string]*) – the aspect name(s).
- **observer** (*function or Object*) – the observer

Return type `Disposable`

`Observable.Observable.prototype.detach` (*[aspect], observer*)

Detaches the observer to the specified aspect(s). If aspect is omitted, the observer will be detached from the global aspect which is notified on every aspect.

Arguments

- **aspect** (*string or Array[string]*) – the aspect name(s).
- **observer** (*function or Object*) – the observer

`Observable.Observable.prototype.notify` (*aspect*[[*argument*], ...])

Notifies observers attached to specified and global aspects. All arguments of this function are passed to each observer.

Arguments

- **aspect** (*string*) – the aspect name.

`Observable.Observable.prototype.isObservable` ()

Returns True if class of current instance is not DummyObservable.

Return type Boolean

StoreObservable Interface

class StoreObservable (*store*)

Creates an observable interface for `IStore()` instance. Inherited from the `Observable()` class.

Arguments

- **store** (`Store`) – the `IStore()` instance to be observable.

`StoreObservable.StoreObservable.prototype`

An `Observable()` instance.

`StoreObservable.StoreObservable.prototype.attachByAttr` (*attr*, *defaultValue*, *observer*)

Attaches observer to “add”, “update”, “delete” events of the *store*. The *observer* will be notified only if value attribute is changed with the arguments:

- attribute name
- old value
- new value

Arguments

- **attr** (*string* or *Array[string]*) – the aspect name(s).
- **defaultValue** – default value (used as attribute value when object is added or deleted)
- **observer** (*function* or *Object*) – the observer

Return type `CompositeDisposable`

Result Observable Interface

class ResultObservable (*subject*)

Creates an observable interface for *Result()* instance. Inherited from the *Observable()* class.

Arguments

- **store** (*Store*) – the *IStore()* instance to be observable.

ResultObservable.ResultObservable.prototype.**prototype**

An *Observable()* instance.

ResultObservable.ResultObservable.prototype.**attachByAttr** (*attr*, *defaultValue*,
observer)

Attaches observer to “add”, “update”, “delete” events of the *result*. The *observer* will be notified only if value attribute is changed with the arguments:

- attribute name
- old value
- new value

Arguments

- **attr** (*string* or *Array[string]*) – the aspect name(s).
- **defaultValue** – default value (used as attribute value when object is added or deleted)
- **observer** (*function* or *Object*) – the observer

Return type CompositeDisposable

13.1 Comparison operators

13.1.1 \$eq

Specifies equality condition. The *\$eq* operator matches objects where the value of a field equals the specified value.

```
{<field>: {$eq: <value>}}
```

The *\$eq* expression is equivalent to `{field: <value>}`

13.1.2 \$ne

Specifies not equality condition. The *\$ne* operator matches objects where the value of a field doesn't equal the specified value.

```
{<field>: {$ne: <value>}}
```

13.1.3 \$in

The *\$in* operator selects the objects where the value of a field equals any value in the specified array.

```
{field: {$in: [<value1>, <value2>, ... <valueN> ]}}
```

13.1.4 \$callable

Function arguments: value, obj, field.

```
{field: {$callable: <function>}}
```

The short form:

```
{field: <function>}
```

Another way to use *\$callable* operator:

```
{$callable: <function>}
```

In this case the function accepts `obj` as single argument.

13.2 Logical operators

13.2.1 \$and

\$and performs a logical AND operation on an array of two or more expressions (e.g. `<expression1>`, `<expression2>`, etc.) and selects the objects that satisfy all the expressions in the array.

```
{$and: [{<expression1>}, {<expression2>}, ... , {<expressionN>}]}
```

In short form you can simple list expressions in single object. These two expressions are equivalent:

```
{$and: [{firstName: 'Donald'}, {lastName: 'Duck'}]}
```

```
{firstName: 'Donald', lastName: 'Duck'}
```

13.2.2 \$or

The *\$or* operator performs a logical OR operation on an array of two or more `<expressions>` and selects the objects that satisfy at least one of the `<expressions>`.

```
{$or: [{<expression1>}, {<expression2>}, ... , {<expressionN>}]}
```

13.3 Relational operators

All relation operators can be nested, for example, this expression is valid:

```
tagStore.find({'posts.author.country.code': 'USA'})
```

13.3.1 \$rel

Delegates expression to related store by relation. The type of relation will be detected automatically. The relation should be described by one of:

- `Store.relations.foreignKey`
- `Store.relations.oneToMany`
- `Store.relations.manyToMany`


```
{relation: {$rel: {<expression>}}}
```

In short form you can use dot in the field (the left part). These two expressions are equivalent:

```
{author: {$rel: {firstName: 'Donald'}}
```

```
{'author.firstName': 'Donald'}
```

13.4 Query Modifiers

13.4.1 \$query

Selection criteria.

```
{$query: {title: 'Donald Duck'}}
```

13.4.2 \$orderby

Warning: This operator is not implemented yet!

The \$orderby operator sorts the results of a query in ascending or descending order.

```
{$query: {title: 'Donald Duck'}, $orderby: [{age: -1}, {title: 1}]}
```

This example return all objects sorted by the “age” field in descending order and then by the “title” field in ascending order. Specify a value to \$orderby of negative one (e.g. -1, as above) to sort in descending order or a positive value (e.g. 1) to sort in ascending order.

13.4.3 \$limit

Warning: This operator is not implemented yet!

Limit.

```
{$query: {title: 'Donald Duck'}, $limit: 10}
```

13.4.4 \$offset

Warning: This operator is not implemented yet!

Offset.

```
{ $query: { title: 'Donald Duck' }, $offset: 10 }
```

14.1 Query

```
1 define(['../store', './utils'], function(store, utils) {
2
3     'use strict';
4
5     var assert = utils.assert,
6         expectPks = utils.expectPks;
7
8
9     function testQuery(resolve, reject) {
10        var registry = new store.Registry();
11
12        function Post(attrs) {
13            store.clone(attrs, this);
14        }
15        Post.prototype = {
16            constructor: Post,
17            getSlug: function() {
18                return this.slug;
19            }
20        };
21
22        var postStore = new store.Store({
23            indexes: ['slug', 'author'],
24            remoteStore: new store.DummyStore(),
25            model: Post
26        });
27        registry.register('post', postStore);
28
29        registry.ready();
30
31    }
```

(continues on next page)

(continued from previous page)

```

32     var posts = [
33         new Post({id: 1, slug: 's11', title: 't11', author: 1}),
34         new Post({id: 2, slug: 's11', title: 't12', author: 1}), // slug can be
↪unique per date
35         new Post({id: 3, slug: 's13', title: 't11', author: 2}),
36         new Post({id: 4, slug: 's14', title: 't14', author: 3})
37     ];
38     store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
39
40     var r;
41
42     r = registry.get('post').find({slug: 's11'});
43     assert(expectPks(r, [1, 2]));
44
45     r = registry.get('post').find({getSlug: 's11'});
46     assert(expectPks(r, [1, 2]));
47
48     r = registry.get('post').find({slug: 's11', author: 1});
49     assert(expectPks(r, [1, 2]));
50
51     r = registry.get('post').find({author: {'$ne': 1}});
52     assert(expectPks(r, [3, 4]));
53
54     r = registry.get('post').find({'$callable': function(post) { return post.
↪author === 1; }});
55     assert(expectPks(r, [1, 2]));
56
57     r = registry.get('post').find({author: function(author_id) { return author_id
↪=== 1; }});
58     assert(expectPks(r, [1, 2]));
59
60     r = registry.get('post').find({'$and': [{slug: 's11'}, {author: 1}]});
61     assert(expectPks(r, [1, 2]));
62
63     r = registry.get('post').find({'$or': [{slug: 's11'}, {author: 2}]});
64     assert(expectPks(r, [1, 2, 3]));
65
66     r = registry.get('post').find({'$or': [{slug: 's11'}, {title: 't11'}]}); //
↪No index
67     assert(expectPks(r, [1, 2, 3]));
68
69     r = registry.get('post').find({
70         '$and': [
71             {
72                 '$or': [
73                     {slug: 's11'},
74                     {slug: 's12'}
75                 ]
76             },
77             {author: 1}
78         ]
79     });
80     assert(expectPks(r, [1, 2]));
81
82     r = registry.get('post').find({slug: {'$in': ['s11', 's13']}});
83     assert(expectPks(r, [1, 2, 3]));

```

(continues on next page)

(continued from previous page)

```

84     registry.destroy();
85     resolve();
86   }
87   }
88   return testQuery;
89 });

```

14.2 Simple relations

```

1  define(['../store', './utils'], function(store, utils) {
2
3    'use strict';
4
5    var assert = utils.assert,
6        expectPks = utils.expectPks;
7
8
9    function testSimpleRelations(resolve, reject) {
10     var registry = new store.Registry();
11
12     var postStore = new store.Store({
13       indexes: ['slug', 'author'],
14       relations: {
15         foreignKey: {
16           author: {
17             field: 'author',
18             relatedStore: 'author',
19             relatedField: 'id',
20             relatedName: 'posts',
21             onDelete: store.cascade
22           }
23         }
24       },
25       remoteStore: new store.DummyStore()
26     });
27     registry.register('post', postStore);
28
29     var authorStore = new store.Store({
30       indexes: ['firstName', 'lastName'],
31       remoteStore: new store.DummyStore()
32     });
33     registry.register('author', authorStore);
34
35     registry.ready();
36
37     var authors = [
38       {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
39       {id: 2, firstName: 'Fn1', lastName: 'Ln2'},
40       {id: 3, firstName: 'Fn3', lastName: 'Ln1'}
41     ];
42     store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });
43
44     var posts = [

```

(continues on next page)

(continued from previous page)

```

45         {id: 1, slug: 'sl1', title: 'tl1', author: 1},
46         {id: 2, slug: 'sl1', title: 'tl2', author: 1}, // slug can be unique per_
↳date
47         {id: 3, slug: 'sl3', title: 'tl1', author: 2},
48         {id: 4, slug: 'sl4', title: 'tl4', author: 3}
49     ];
50     store.whenIter(posts, function(post) { return postStore.getLocalStore().
↳add(post); });
51
52     var r = registry.get('post').find({slug: 'sl1'});
53     assert(expectPks(r, [1, 2]));
54
55     var author = registry.get('author').get(1);
56     r = registry.get('post').find({'author': author});
57     assert(expectPks(r, [1, 2]));
58
59     r = registry.get('post').find({'author.firstName': 'Fn1'});
60     assert(expectPks(r, [1, 2, 3]));
61
62     r = registry.get('post').find({'author': {'$rel': {firstName: 'Fn1'}}});
63     assert(expectPks(r, [1, 2, 3]));
64
65     r = registry.get('author').find({'posts.slug': {'$in': ['sl1', 'sl3']}});
66     assert(expectPks(r, [1, 2]));
67
68     r = registry.get('author').find({'posts': {'$rel': {slug: {'$in': ['sl1', 'sl3
↳']}}}});
69     assert(expectPks(r, [1, 2]));
70
71
72     // Add
73     var post = {id: 5, slug: 'sl5', title: 'tl5', author: 3};
74     return registry.get('post').add(post).then(function(post) {
75         assert(5 in registry.get('post').getLocalStore().pkIndex);
76         assert(registry.get('post').getLocalStore().indexes['slug']['sl5'].
↳indexOf(post) !== -1);
77
78
79     // Update
80     post = registry.get('post').get(5);
81     post.slug = 'sl5.2';
82     return registry.get('post').update(post).then(function(post) {
83         assert(5 in registry.get('post').getLocalStore().pkIndex);
84         assert(registry.get('post').getLocalStore().indexes['slug']['sl5.2'].
↳indexOf(post) !== -1);
85         assert(registry.get('post').getLocalStore().indexes['slug']['sl5'].
↳indexOf(post) === -1);
86
87
88     // Delete
89     var author = registry.get('author').get(1);
90     post = registry.get('post').find({'author': 1})[0];
91     assert(registry.get('post').getLocalStore().indexes['slug']['sl1'].
↳indexOf(post) !== -1);
92     assert(1 in registry.get('post').getLocalStore().pkIndex);
93     return registry.get('author').delete(author).then(function() {
94         assert(registry.get('post').getLocalStore().indexes['slug']['sl1
↳'].indexOf(post) === -1);

```

(continues on next page)

(continued from previous page)

```

95         assert(!(1 in registry.get('post').getStore().pkIndex));
96         var r = registry.get('author').find();
97         assert(expectPks(r, [2, 3]));
98         r = registry.get('post').find();
99         assert(expectPks(r, [3, 4, 5]));
100
101         registry.destroy();
102         // resolve();
103     });
104 });
105 });
106 }
107 return testSimpleRelations;
108 });

```

14.3 Composite relations

```

1 define(['../store', './utils'], function(store, utils) {
2
3     'use strict';
4
5     var assert = utils.assert,
6         expectPks = utils.expectPks;
7
8
9     function testCompositeRelations(resolve, reject) {
10        var registry = new store.Registry();
11
12        // Use reverse order of store creation.
13        var authorStore = new store.Store({
14            pk: ['id', 'lang'],
15            indexes: ['firstName', 'lastName'],
16            remoteStore: new store.DummyStore()
17        });
18        registry.register('author', authorStore);
19
20        var postStore = new store.Store({
21            pk: ['id', 'lang'],
22            indexes: ['lang', 'slug', 'author'],
23            relations: {
24                foreignKey: {
25                    author: {
26                        field: ['author', 'lang'],
27                        relatedStore: 'author',
28                        relatedField: ['id', 'lang'],
29                        relatedName: 'posts',
30                        onDelete: store.cascade
31                    }
32                }
33            },
34            remoteStore: new store.DummyStore()
35        });
36        registry.register('post', postStore);
37

```

(continues on next page)

(continued from previous page)

```

38 registry.ready();
39
40 var authors = [
41   {id: 1, lang: 'en', firstName: 'Fn1', lastName: 'Ln1'},
42   {id: 1, lang: 'ru', firstName: 'Fn1-ru', lastName: 'Ln1-ru'},
43   {id: 2, lang: 'en', firstName: 'Fn1', lastName: 'Ln2'},
44   {id: 3, lang: 'en', firstName: 'Fn3', lastName: 'Ln1'}
45 ];
46 store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });
47
48 var posts = [
49   {id: 1, lang: 'en', slug: 's11', title: 't11', author: 1},
50   {id: 1, lang: 'ru', slug: 's11-ru', title: 't11-ru', author: 1},
51   {id: 2, lang: 'en', slug: 's11', title: 't12', author: 1}, // slug can_
↪be unique per date
52   {id: 3, lang: 'en', slug: 's13', title: 't11', author: 2},
53   {id: 4, lang: 'en', slug: 's14', title: 't14', author: 3}
54 ];
55 store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
56
57 var compositePkAccessor = function(o) { return [o.id, o.lang]; };
58
59 var r = postStore.find({slug: 's11'});
60 assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));
61
62 var author = registry.get('author').get([1, 'en']);
63 r = registry.get('post').find({'author': author});
64 assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));
65
66 r = postStore.find({'author.firstName': 'Fn1'});
67 assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));
68
69 r = postStore.find({'author': {'$rel': {firstName: 'Fn1'}}});
70 assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));
71
72 r = authorStore.find({'posts.slug': {'$in': ['s11', 's13']}});
73 assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));
74
75 r = authorStore.find({'posts': {'$rel': {slug: {'$in': ['s11', 's13']}}}});
76 assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));
77
78 // Add
79 var post = {id: 5, lang: 'en', slug: 's15', title: 't15', author: 3};
80 postStore.add(post).then(function(post) {
81   assert([5, 'en'] in postStore.getLocalStore().pkIndex);
82   assert(postStore.getLocalStore().indexes['slug']['s15'].indexOf(post) !==
↪-1);
83
84
85 // Update
86 var post = postStore.get([5, 'en']);
87 post.slug = 's15.2';
88 postStore.update(post).then(function(post) {
89   assert([5, 'en'] in postStore.getLocalStore().pkIndex);
90   assert(postStore.getLocalStore().indexes['slug']['s15.2'].
↪indexOf(post) !== -1);

```

(continues on next page)

(continued from previous page)

```

91     assert(postStore.getLocalStore().indexes['slug']['sl5'].indexOf(post) ↵
↵ === -1);
92
93
94     // Delete
95     var author = authorStore.get([1, 'en']);
96     post = postStore.find({author: 1, lang: 'en'})[0];
97     assert(postStore.getLocalStore().indexes['slug']['sl1'].indexOf(post) ↵
↵ !== -1);
98     assert([1, 'en'] in postStore.getLocalStore().pkIndex);
99     authorStore.delete(author).then(function(post) {
100         assert(postStore.getLocalStore().indexes['slug']['sl1'].
↵ indexOf(post) === -1);
101         assert(!([1, 'en'] in postStore.getLocalStore().pkIndex));
102         var r = authorStore.find();
103         assert(expectPks(r, [[1, 'ru'], [2, 'en'], [3, 'en']], ↵
↵ compositePkAccessor));
104         r = postStore.find();
105         assert(expectPks(r, [[1, 'ru'], [3, 'en'], [4, 'en'], [5, 'en']], ↵
↵ compositePkAccessor));
106
107         registry.destroy();
108         resolve();
109     });
110 });
111 });
112 }
113 return testCompositeRelations;
114 });

```

14.4 Many to many relations

```

1  define(['../store', './utils'], function(store, utils) {
2
3      'use strict';
4
5      var assert = utils.assert,
6          expectPks = utils.expectPks;
7
8
9      function testManyToMany(resolve, reject) {
10         var registry = new store.Registry();
11
12         var tagStore = new store.Store({
13             pk: ['id', 'lang'],
14             indexes: ['slug'],
15             remoteStore: new store.DummyStore()
16         });
17         registry.register('tag', tagStore);
18
19         var tagPostStore = new store.Store({
20             relations: {
21                 foreignKey: {
22                     post: {

```

(continues on next page)

```

23         field: ['postId', 'postLang'],
24         relatedStore: 'post',
25         relatedField: ['id', 'lang'],
26         relatedName: 'tagPostSet',
27         onDelete: store.cascade
28     },
29     tag: {
30         field: ['tagId', 'tagLang'],
31         relatedStore: 'tag',
32         relatedField: ['id', 'lang'],
33         relatedName: 'tagPostSet',
34         onDelete: store.cascade
35     }
36 }
37 },
38 remoteStore: new store.DummyStore()
39 });
40 tagPostStore.getLocalStore().setNextPk = function(obj) {
41     tagPostStore._pkCounter || (tagPostStore._pkCounter = 0);
42     this.getObjectAccessor().setPk(obj, ++tagPostStore._pkCounter);
43 };
44 registry.register('tagPost', tagPostStore);
45
46 var postStore = new store.Store({
47     pk: ['id', 'lang'],
48     indexes: ['lang', 'slug', 'author'],
49     relations: {
50         manyToMany: {
51             tags: {
52                 relation: 'tagPostSet',
53                 relatedStore: 'tag',
54                 relatedRelation: 'tagPostSet'
55             }
56         }
57     },
58     remoteStore: new store.DummyStore()
59 });
60 registry.register('post', postStore);
61
62 registry.ready();
63
64 var tags = [
65     {id: 1, lang: 'en', name: 'T1'},
66     {id: 1, lang: 'ru', name: 'T1-ru'},
67     {id: 2, lang: 'en', name: 'T1'},
68     {id: 3, lang: 'en', name: 'T3'},
69     {id: 4, lang: 'en', name: 'T4'}
70 ];
71 store.whenIter(tags, function(tag) { return tagStore.getLocalStore().add(tag);
72 ↪ });
73
74 var posts = [
75     {id: 1, lang: 'en', slug: 's11', title: 't11'},
76     {id: 1, lang: 'ru', slug: 's11-ru', title: 't11-ru'},
77     {id: 2, lang: 'en', slug: 's11', title: 't12'}, // slug can be unique_
78 ↪per date
79     {id: 3, lang: 'en', slug: 's13', title: 't11'},

```

(continues on next page)

(continued from previous page)

```

78     {id: 4, lang: 'en', slug: 'sl4', title: 'tl4'}
79   ];
80   store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
81
82   var tagPosts = [
83     {postId: 1, postLang: 'en', tagId: 1, tagLang: 'en'},
84     {postId: 1, postLang: 'ru', tagId: 1, tagLang: 'ru'},
85     {postId: 2, postLang: 'en', tagId: 1, tagLang: 'en'},
86     {postId: 3, postLang: 'en', tagId: 2, tagLang: 'en'},
87     {postId: 4, postLang: 'en', tagId: 4, tagLang: 'en'}
88   ];
89   store.whenIter(tagPosts, function(tagPost) { return tagPostStore.
↪getLocalStore().add(tagPost); });
90
91   var compositePkAccessor = function(o) { return [o.id, o.lang]; };
92   var r;
93
94   r = postStore.find({slug: 'sl1'});
95   assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));
96
97   r = postStore.find({'tags.name': 'T1'});
98   assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));
99
100  r = postStore.find({'tags': {'$rel': {name: 'T1'}}});
101  assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));
102
103  registry.destroy();
104  resolve();
105  }
106  return testManyToMany;
107  });

```

14.5 Compose

```

1  define(['../store', './utils'], function(store, utils) {
2
3    'use strict';
4
5    var assert = utils.assert,
6        expectPks = utils.expectPks;
7
8
9    function testCompose(resolve, reject) {
10     var registry = new store.Registry();
11
12     var authorStore = new store.Store({
13       pk: ['id', 'lang'],
14       indexes: ['firstName', 'lastName'],
15       remoteStore: new store.DummyStore()
16     });
17     registry.register('author', authorStore);
18
19     var tagStore = new store.Store({

```

(continues on next page)

```
20     pk: ['id', 'lang'],
21     indexes: ['slug'],
22     remoteStore: new store.DummyStore()
23   });
24   registry.register('tag', tagStore);
25
26   var tagPostStore = new store.Store({
27     relations: {
28       foreignKey: {
29         post: {
30           field: ['postId', 'postLang'],
31           relatedStore: 'post',
32           relatedField: ['id', 'lang'],
33           relatedName: 'tagPostSet',
34           onDelete: store.cascade
35         },
36         tag: {
37           field: ['tagId', 'tagLang'],
38           relatedStore: 'tag',
39           relatedField: ['id', 'lang'],
40           relatedName: 'tagPostSet',
41           onDelete: store.cascade
42         }
43       }
44     },
45     remoteStore: new store.DummyStore()
46   });
47   tagPostStore.getLocalStore().setNextPk = function(obj) {
48     tagPostStore._pkCounter || (tagPostStore._pkCounter = 0);
49     this.getObjectAccessor().setPk(obj, ++tagPostStore._pkCounter);
50   };
51   registry.register('tagPost', tagPostStore);
52
53   var postStore = new store.Store({
54     pk: ['id', 'lang'],
55     indexes: ['lang', 'slug', 'author'],
56     relations: {
57       foreignKey: {
58         author: {
59           field: ['author', 'lang'],
60           relatedStore: 'author',
61           relatedField: ['id', 'lang'],
62           relatedName: 'posts',
63           onDelete: store.cascade
64         }
65       },
66       manyToMany: {
67         tags: {
68           relation: 'tagPostSet',
69           relatedStore: 'tag',
70           relatedRelation: 'tagPostSet'
71         }
72       }
73     },
74     remoteStore: new store.DummyStore()
75   });
76   registry.register('post', postStore);
```

(continues on next page)

(continued from previous page)

```

77 registry.ready();
78
79
80 var authors = [
81     {id: 1, lang: 'en', firstName: 'Fn1', lastName: 'Ln1'},
82     {id: 1, lang: 'ru', firstName: 'Fn1-ru', lastName: 'Ln1-ru'},
83     {id: 2, lang: 'en', firstName: 'Fn1', lastName: 'Ln2'},
84     {id: 3, lang: 'en', firstName: 'Fn3', lastName: 'Ln1'}
85 ];
86 store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });
87
88 var tags = [
89     {id: 1, lang: 'en', name: 'T1'},
90     {id: 1, lang: 'ru', name: 'T1-ru'},
91     {id: 2, lang: 'en', name: 'T1'},
92     {id: 3, lang: 'en', name: 'T3'},
93     {id: 4, lang: 'en', name: 'T4'}
94 ];
95 store.whenIter(tags, function(tag) { return tagStore.getLocalStore().add(tag);
↪ });
96
97 var posts = [
98     {id: 1, lang: 'en', slug: 's11', title: 't11', author: 1},
99     {id: 1, lang: 'ru', slug: 's11-ru', title: 't11-ru', author: 1},
100    {id: 2, lang: 'en', slug: 's11', title: 't12', author: 1}, // slug can_
↪be unique per date
101    {id: 3, lang: 'en', slug: 's13', title: 't11', author: 2},
102    {id: 4, lang: 'en', slug: 's14', title: 't14', author: 3}
103 ];
104 store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
105
106 var tagPosts = [
107     {postId: 1, postLang: 'en', tagId: 1, tagLang: 'en'},
108     {postId: 1, postLang: 'ru', tagId: 1, tagLang: 'ru'},
109     {postId: 2, postLang: 'en', tagId: 1, tagLang: 'en'},
110     {postId: 3, postLang: 'en', tagId: 2, tagLang: 'en'},
111     {postId: 4, postLang: 'en', tagId: 4, tagLang: 'en'}
112 ];
113 store.whenIter(tagPosts, function(tagPost) { return tagPostStore.
↪getLocalStore().add(tagPost); });
114
115 var author = authorStore.get([1, 'en']);
116
117 store.when(authorStore.compose(author), function(author) {
118     console.debug(author);
119     /*
120      * Similar output of composite object:
121      * {"id":1, "lang":"en", "firstName": "Fn1", "lastName": "Ln1", "posts": [
122      *     {"id":1, "lang": "en", "slug": "s11", "title": "t11", "author":1,
↪"tags": [
123      *         {"id": 1, "lang": "en", "name": "T1"}
124      *     ]},
125      *     {"id": 2, "lang": "en", "slug": "s11", "title": "t12", "author": 1,
↪ "tags":[
126      *         {"id": 1, "lang": "en", "name": "T1"}

```

(continues on next page)

(continued from previous page)

```

127     *     ]}
128     *   ]}"
129     */
130     var compositePkAccessor = function(o) { return [o.id, o.lang]; };
131     assert(expectPks(author.posts, [[1, 'en'], [2, 'en']],
↪compositePkAccessor));
132     assert(expectPks(author.posts[0].tags, [[1, 'en']], compositePkAccessor));
133     assert(expectPks(author.posts[1].tags, [[1, 'en']], compositePkAccessor));
134
135     registry.destroy();
136     resolve();
137   });
138 }
139 return testCompose;
140 });

```

14.6 Decompose

```

1 define(['../store', './utils'], function(store, utils) {
2
3   'use strict';
4
5   var assert = utils.assert,
6       expectPks = utils.expectPks;
7
8
9   function testDecompose(resolve, reject) {
10     var registry = new store.Registry();
11
12     var categoryStore = new store.Store({
13       pk: ['id', 'lang'],
14       remoteStore: new store.DummyStore()
15     });
16     registry.register('category', categoryStore);
17
18     var authorStore = new store.Store({
19       pk: ['id', 'lang'],
20       indexes: ['firstName', 'lastName'],
21       remoteStore: new store.DummyStore()
22     });
23     registry.register('author', authorStore);
24
25     var tagStore = new store.Store({
26       pk: ['id', 'lang'],
27       indexes: ['slug'],
28       remoteStore: new store.DummyStore()
29     });
30     registry.register('tag', tagStore);
31
32     var tagPostStore = new store.Store({
33       relations: {
34         foreignKey: {
35           post: {
36             field: ['postId', 'postLang'],

```

(continues on next page)

(continued from previous page)

```

37         relatedStore: 'post',
38         relatedField: ['id', 'lang'],
39         relatedName: 'tagPostSet',
40         onDelete: store.cascade
41     },
42     tag: {
43         field: ['tagId', 'tagLang'],
44         relatedStore: 'tag',
45         relatedField: ['id', 'lang'],
46         relatedName: 'tagPostSet',
47         onDelete: store.cascade
48     }
49 }
50 },
51 remoteStore: new store.DummyStore()
52 });
53 tagPostStore.getLocalStore().setNextPk = function(obj) {
54     tagPostStore._pkCounter || (tagPostStore._pkCounter = 0);
55     this.getObjectAccessor().setPk(obj, ++tagPostStore._pkCounter);
56 };
57 registry.register('tagPost', tagPostStore);
58
59 var postStore = new store.Store({
60     pk: ['id', 'lang'],
61     indexes: ['lang', 'slug', 'author'],
62     relations: {
63         foreignKey: {
64             author: {
65                 field: ['author', 'lang'],
66                 relatedStore: 'author',
67                 relatedField: ['id', 'lang'],
68                 relatedName: 'posts',
69                 onDelete: store.cascade
70             },
71             category: {
72                 field: ['category_id', 'lang'],
73                 relatedStore: 'category',
74                 relatedField: ['id', 'lang'],
75                 relatedName: 'posts',
76                 onDelete: store.cascade
77             }
78         },
79         manyToMany: {
80             tags: {
81                 relation: 'tagPostSet',
82                 relatedStore: 'tag',
83                 relatedRelation: 'tagPostSet'
84             }
85         }
86     },
87     remoteStore: new store.DummyStore()
88 });
89 registry.register('post', postStore);
90
91 registry.ready();
92
93 var author = {

```

(continues on next page)

```

94     id: 1,
95     lang: 'en',
96     firstName: 'Fn1',
97     lastName: 'Ln1',
98     posts: [
99         {
100             id: 2,
101             lang: 'en',
102             slug: 'sl1',
103             title: 'tl1',
104             category: {id: 8, lang: 'en', name: 'C1'},
105             tags: [
106                 {id: 5, lang: 'en', name: 'T1'},
107                 {id: 6, lang: 'en', name: 'T1'}
108             ]
109         },
110         {
111             id: 3,
112             lang: 'en',
113             slug: 'sl1',
114             title: 'tl2',
115             category: {id: 9, lang: 'en', name: 'C2'},
116             tags: [
117                 {id: 5, lang: 'en', name: 'T1'},
118                 {id: 7, lang: 'en', name: 'T3'}
119             ]
120         }
121     ]
122 };
123
124 store.when(authorStore.decompose(author), function(author) {
125     var compositePkAccessor = function(o) { return [o.id, o.lang]; };
126     var r;
127     r = authorStore.find();
128     assert(expectPks(r, [[1, 'en']], compositePkAccessor));
129     r = postStore.find();
130     assert(expectPks(r, [[2, 'en'], [3, 'en']], compositePkAccessor));
131     for (var i = 0; i < r.length; i++) {
132         assert(r[i].author === 1);
133     }
134     r = tagStore.find();
135     assert(expectPks(r, [[5, 'en'], [6, 'en'], [7, 'en']],
136 ↪ compositePkAccessor));
137     r = tagPostStore.find({postId: 2, postLang: 'en', tagId: 5, tagLang: 'en'}
138 ↪ );
139     assert(r.length === 1);
140     r = tagPostStore.find({postId: 2, postLang: 'en', tagId: 6, tagLang: 'en'}
141 ↪ );
142     assert(r.length === 1);
143     r = tagPostStore.find({postId: 3, postLang: 'en', tagId: 5, tagLang: 'en'}
144 ↪ );
145     assert(r.length === 1);
146     r = tagPostStore.find({postId: 3, postLang: 'en', tagId: 7, tagLang: 'en'}
147 ↪ );
148     assert(r.length === 1);
149     r = categoryStore.find();

```

(continues on next page)

(continued from previous page)

```

146     assert(expectPk(r, [[8, 'en'], [9, 'en']], compositePkAccessor));
147
148     assert(author.posts[0].id === 2);
149     assert(author.posts[0].author === 1);
150     assert(author.posts[1].id === 3);
151     assert(author.posts[1].author === 1);
152     assert(author.posts.length === 2);
153
154     assert(author.posts[0].tags[0].id === 5);
155     assert(author.posts[0].tags[1].id === 6);
156     assert(author.posts[0].tags.length === 2);
157
158     assert(author.posts[1].tags[0].id === 5);
159     assert(author.posts[1].tags[1].id === 7);
160     assert(author.posts[1].tags.length === 2);
161
162     assert(author.posts[0].category_id === 8);
163     assert(author.posts[1].category_id === 9);
164
165     registry.destroy();
166     resolve();
167   });
168 }
169 return testDecompose;
170 });

```

14.7 Observable object

Example of fast real-time aggregation:

```

1  define(['../store', './utils'], function(store, utils) {
2
3    'use strict';
4
5    var assert = utils.assert;
6
7
8    function testObservable(resolve, reject) {
9      // Example of fast real-time aggregation
10     var registry = new store.Registry();
11     registry.observed().attach('register', function(aspect, newStore) {
12       newStore.getLocalStore().observed().attach('add', function(aspect, obj) {
13         ↪store.observe(obj); });
14     });
15
16     var postStore = new store.Store({
17       indexes: ['slug', 'author'],
18       relations: {
19         foreignKey: {
20           author: {
21             field: 'author',
22             relatedStore: 'author',
23             relatedField: 'id',
24             relatedName: 'posts',

```

(continues on next page)

```

24         onDelete: store.cascade
25     }
26 }
27 },
28 remoteStore: new store.DummyStore()
29 });
30 registry.register('post', postStore);
31
32 var authorStore = new store.Store({
33     indexes: ['firstName', 'lastName'],
34     remoteStore: new store.DummyStore()
35 });
36 registry.register('author', authorStore);
37 registry.observed().attach('ready', function() {
38     registry.get('post').getStore().observed().attach('add',
↪function(aspect, post) {
39         registry.get('author').find({id: post.author}).
↪forEach(function(author) {
40             author.observed().set('views_total', author.views_total + post.
↪views_count);
41             post.observed().attach('views_count', function(name, oldValue,
↪newValue) {
42                 author.observed().set('views_total', author.views_total -
↪oldValue + newValue);
43             });
44         });
45     });
46 });
47
48 registry.ready();
49
50 var authors = [
51     {id: 1, firstName: 'Fn1', lastName: 'Ln1', views_total: 0},
52     {id: 2, firstName: 'Fn1', lastName: 'Ln2', views_total: 0},
53     {id: 3, firstName: 'Fn3', lastName: 'Ln1', views_total: 0}
54 ];
55 store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });
56
57 var posts = [
58     {id: 1, slug: 's11', title: 't11', author: 1, views_count: 5},
59     {id: 2, slug: 's11', title: 't12', author: 1, views_count: 6}, // slug
↪can be unique per date
60     {id: 3, slug: 's13', title: 't11', author: 2, views_count: 7},
61     {id: 4, slug: 's14', title: 't14', author: 3, views_count: 8}
62 ];
63 store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
64
65 var author = registry.get('author').get(1);
66 assert(author.views_total === 11);
67 var post = registry.get('post').find({author: author.id})[0];
68 post.observed().set('views_count', post.views_count + 1);
69 assert(author.views_total === 12);
70
71 postStore.getLocalStore().add({id: 5, slug: 's15', title: 't15', author: 1,
↪views_count: 8});

```

(continues on next page)

(continued from previous page)

```

72     assert(author.views_total === 20);
73     resolve();
74
75     }
76     return testObservable;
77 });

```

14.8 StoreObservable

Example of fast real-time aggregation using :

```

1  define(['../store', './utils'], function(store, utils) {
2
3     'use strict';
4
5     var assert = utils.assert;
6
7
8     function testStoreObservable(resolve, reject) {
9         var registry = new store.Registry();
10
11
12         var postStore = new store.Store({
13             indexes: ['slug', 'author'],
14             relations: {
15                 foreignKey: {
16                     author: {
17                         field: 'author',
18                         relatedStore: 'author',
19                         relatedField: 'id',
20                         relatedName: 'posts',
21                         onDelete: store.cascade
22                     }
23                 }
24             },
25             remoteStore: new store.DummyStore()
26         });
27         registry.register('post', postStore);
28
29         registry.get('post').getLocalStorage().observed().attachByAttr('views_count', 0,
30 ↪ function(attr, oldValue, newValue) {
31             var post = this;
32             var author = registry.get('author').get(post.author);
33             author.views_total = (author.views_total || 0) + newValue - oldValue;
34             registry.get('author').getLocalStorage().update(author);
35         });
36
37         var authorStore = new store.Store({
38             indexes: ['firstName', 'lastName'],
39             remoteStore: new store.DummyStore()
40         });
41         registry.register('author', authorStore);
42

```

(continues on next page)

(continued from previous page)

```

43
44     registry.ready();
45
46     var authors = [
47         {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
48         {id: 2, firstName: 'Fn2', lastName: 'Ln2'},
49         {id: 3, firstName: 'Fn3', lastName: 'Ln1'}
50     ];
51     store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });
52
53     var posts = [
54         {id: 1, slug: 's11', title: 't11', author: 1, views_count: 5},
55         {id: 2, slug: 's11', title: 't12', author: 1, views_count: 6}, // slug_
↪can be unique per date
56         {id: 3, slug: 's13', title: 't11', author: 2, views_count: 7},
57         {id: 4, slug: 's13', title: 't11', author: 2, views_count: 8},
58         {id: 5, slug: 's14', title: 't14', author: 3, views_count: 9}
59     ];
60     store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
61
62     var author = registry.get('author').get(1);
63     assert(author.views_total === 11);
64
65     // update
66     var post = registry.get('post').find({author: author.id})[0];
67     post.views_count += 1;
68     registry.get('post').getLocalStore().update(post);
69     assert(author.views_total === 12);
70
71     // add
72     registry.get('post').getLocalStore().add(
73         {id: 6, slug: 's16', title: 't16', author: 1, views_count: 10}
74     );
75     assert(author.views_total === 22);
76
77     // delete
78     registry.get('post').getLocalStore().delete(
79         registry.get('post').get(6)
80     );
81     assert(author.views_total === 12);
82     resolve();
83 }
84 return testStoreObservable;
85 });

```

14.9 Reaction of Result on changes in Store

```

1 define(['../store', './utils'], function(store, utils) {
2
3     'use strict';
4
5     var assert = utils.assert,

```

(continues on next page)

(continued from previous page)

```

6     expectPks = utils.expectPks,
7     expectOrderedPks = utils.expectOrderedPks;
8
9
10    function testResultReaction(resolve, reject) {
11        var registry = new store.Registry();
12
13        var postStore = new store.Store({
14            indexes: ['slug', 'author'],
15            remoteStore: new store.DummyStore()
16        });
17        registry.register('post', postStore);
18
19        registry.ready();
20
21        var posts = [
22            {id: 1, slug: 's11', title: 't11', author: 1},
23            {id: 2, slug: 's11', title: 't12', author: 1}, // slug can be unique per_
↪date
24            {id: 3, slug: 's13', title: 't11', author: 2},
25            {id: 4, slug: 's14', title: 't14', author: 3}
26        ];
27        store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
28
29        var r1 = registry.get('post').find({author: 1});
30        r1.observe();
31        assert(expectPks(r1, [1, 2]));
32        assert(r1.length = 2);
33
34        r1.sort(function(a, b){ return b.id - a.id; });
35        assert(expectOrderedPks(r1, [2, 1]));
36
37        var r2 = r1.slice();
38        assert(expectOrderedPks(r2, [2, 1]));
39        assert(r2.length = 2);
40
41
42        var observer = function(aspect, obj) {
43            observer.args.push([this].concat(Array.prototype.slice.call(arguments)));
44        };
45        observer.args = [];
46        r2.observed().attach(['add', 'update', 'delete'], observer);
47
48        // add
49        postStore.getLocalStore().add({id: 5, slug: 's15', title: 't15', author: 1});
50        assert(expectOrderedPks(r1, [5, 2, 1]));
51        assert(r1.length = 3);
52        assert(expectOrderedPks(r2, [5, 2, 1]));
53        assert(r2.length = 3);
54
55        assert(observer.args.length === 1);
56        assert(observer.args[0][0] === r2);
57        assert(observer.args[0][1] === 'add');
58        assert(observer.args[0][2] === r2[0]);
59
60        observer.args = [];

```

(continues on next page)

(continued from previous page)

```
61     postStore.getLocalStore().add({id: 6, slug: 'sl6', title: 'tl6', author: 2});
62     assert(expectOrderedPks(r1, [5, 2, 1]));
63     assert(r1.length = 3);
64     assert(expectOrderedPks(r2, [5, 2, 1]));
65     assert(r2.length = 3);
66     assert(observer.args.length === 0);
67
68     // update
69     observer.args = [];
70     postStore.getLocalStore().update(postStore.get(5));
71     assert(expectOrderedPks(r1, [5, 2, 1]));
72     assert(r1.length = 3);
73     assert(expectOrderedPks(r2, [5, 2, 1]));
74     assert(r2.length = 3);
75
76     assert(observer.args.length === 1);
77     assert(observer.args[0][0] === r2);
78     assert(observer.args[0][1] === 'update');
79     assert(observer.args[0][2] === r2[0]);
80     assert(observer.args[0][3].id === 5);
81
82     // delete
83     observer.args = [];
84     postStore.getLocalStore().delete(postStore.get(5));
85     assert(expectOrderedPks(r1, [2, 1]));
86     assert(r1.length = 2);
87     assert(expectOrderedPks(r2, [2, 1]));
88     assert(r2.length = 2);
89
90     assert(observer.args.length === 1);
91     assert(observer.args[0][0] === r2);
92     assert(observer.args[0][1] === 'delete');
93     assert(observer.args[0][2].id === 5);
94
95     registry.destroy();
96     resolve();
97 }
98
99
100 function testResultAttachByAttr(resolve, reject) {
101     var registry = new store.Registry();
102
103
104     var postStore = new store.Store({
105         indexes: ['slug', 'author'],
106         relations: {
107             foreignKey: {
108                 author: {
109                     field: 'author',
110                     relatedStore: 'author',
111                     relatedField: 'id',
112                     relatedName: 'posts',
113                     onDelete: store.cascade
114                 }
115             }
116         },
117         remoteStore: new store.DummyStore()
```

(continues on next page)

(continued from previous page)

```

118     });
119     registry.register('post', postStore);
120
121
122     var authorStore = new store.Store({
123         indexes: ['firstName', 'lastName'],
124         remoteStore: new store.DummyStore()
125     });
126     registry.register('author', authorStore);
127
128     registry.get('author').getLocalStore().observed().attach('add', ↵
↵function(aspect, author) {
129         author.views_total = 0;
130         registry.get('post').find({
131             'author.id': author.id
132         }).observe().forEachByAttr('views_count', 0, function(attr, oldValue, ↵
↵newValue) {
133             author.views_total = author.views_total + newValue - oldValue;
134             registry.get('author').getLocalStore().update(author);
135         });
136     });
137
138
139     registry.ready();
140
141     var authors = [
142         {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
143         {id: 2, firstName: 'Fn2', lastName: 'Ln2'},
144         {id: 3, firstName: 'Fn3', lastName: 'Ln1'}
145     ];
146     store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↵add(author); });
147
148     var posts = [
149         {id: 1, slug: 's11', title: 't11', author: 1, views_count: 5},
150         {id: 2, slug: 's11', title: 't12', author: 1, views_count: 6}, // slug ↵
↵can be unique per date
151         {id: 3, slug: 's13', title: 't11', author: 2, views_count: 7},
152         {id: 4, slug: 's13', title: 't11', author: 2, views_count: 8},
153         {id: 5, slug: 's14', title: 't14', author: 3, views_count: 9}
154     ];
155     store.whenIter(posts, function(post) { return postStore.getLocalStore().
↵add(post); });
156
157     var author = registry.get('author').get(1);
158     assert(author.views_total === 11);
159
160     // update
161     var post = registry.get('post').find({author: author.id})[0];
162     post.views_count += 1;
163     registry.get('post').getLocalStore().update(post);
164     assert(author.views_total === 12);
165
166     // add
167     registry.get('post').getLocalStore().add(
168         {id: 6, slug: 's16', title: 't16', author: 1, views_count: 10}
169     );

```

(continues on next page)

```
170     assert(author.views_total === 22);
171
172     // delete
173     registry.get('post').getStore().delete(
174         registry.get('post').get(6)
175     );
176     assert(author.views_total === 12);
177     resolve();
178 }
179
180
181 function testResultRelation(resolve, reject) {
182     var registry = new store.Registry();
183
184
185     var postStore = new store.Store({
186         indexes: ['slug', 'author'],
187         relations: {
188             foreignKey: {
189                 author: {
190                     field: 'author',
191                     relatedStore: 'author',
192                     relatedField: 'id',
193                     relatedName: 'posts',
194                     onDelete: store.cascade
195                 }
196             }
197         },
198         remoteStore: new store.DummyStore()
199     });
200     registry.register('post', postStore);
201
202
203     var authorStore = new store.Store({
204         indexes: ['firstName', 'lastName'],
205         remoteStore: new store.DummyStore()
206     });
207     registry.register('author', authorStore);
208
209
210     registry.ready();
211
212     var authors = [
213         {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
214         {id: 2, firstName: 'Fn2', lastName: 'Ln2'},
215         {id: 3, firstName: 'Fn3', lastName: 'Ln1'},
216         {id: 4, firstName: 'Fn4', lastName: 'Ln4'}
217     ];
218     store.whenIter(authors, function(author) { return authorStore.getStore().
↪add(author); });
219
220     var posts = [
221         {id: 1, slug: 's11', title: 't11', author: 1, views_count: 5},
222         {id: 2, slug: 's11', title: 't12', author: 1, views_count: 6}, // slug_
↪can be unique per date
223         {id: 3, slug: 's13', title: 't11', author: 2, views_count: 7},
224         {id: 4, slug: 's13', title: 't11', author: 2, views_count: 8},
```

(continues on next page)

(continued from previous page)

```

225     {id: 5, slug: 'sl4', title: 'tl4', author: 3, views_count: 9}
226   ];
227   store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });
228
229
230   var r1 = registry.get('author').find({'posts.title': 'tl1'});
231   r1.observe();
232   assert(expectPks(r1, [1, 2]));
233   assert(r1.length = 2);
234
235   r1.sort(function(a, b){ return b.id - a.id; });
236   assert(expectOrderedPks(r1, [2, 1]));
237
238   var r2 = r1.slice();
239   assert(expectOrderedPks(r2, [2, 1]));
240   assert(r2.length = 2);
241
242
243   var observer = function(aspect, obj) {
244     observer.args.push([this].concat(Array.prototype.slice.call(arguments)));
245   };
246   observer.args = [];
247   r2.observed().attach(['add', 'update', 'delete'], observer);
248
249   // add
250   postStore.getLocalStore().add({id: 6, slug: 'sl6', title: 'tl1', author: 3, ↪
↪views_count: 8});
251   assert(expectOrderedPks(r1, [3, 2, 1]));
252   assert(r1.length = 3);
253   assert(expectOrderedPks(r2, [3, 2, 1]));
254   assert(r2.length = 3);
255
256   assert(observer.args.length === 1);
257   assert(observer.args[0][0] === r2);
258   assert(observer.args[0][1] === 'add');
259   assert(observer.args[0][2] === r2[0]);
260   assert(observer.args[0][3] === 0);
261
262   // add 2
263   observer.args = [];
264   postStore.getLocalStore().add({id: 7, slug: 'sl7', title: 'tl7', author: 4, ↪
↪views_count: 8});
265   assert(expectOrderedPks(r1, [3, 2, 1]));
266   assert(r1.length = 3);
267   assert(expectOrderedPks(r2, [3, 2, 1]));
268   assert(r2.length = 3);
269   assert(observer.args.length === 0);
270
271   // update
272   observer.args = [];
273   var post = postStore.get(7);
274   post.title = 'tl1';
275   postStore.getLocalStore().update(post);
276   assert(expectOrderedPks(r1, [4, 3, 2, 1]));
277   assert(r1.length = 4);
278   assert(expectOrderedPks(r2, [4, 3, 2, 1]));

```

(continues on next page)

```
279     assert(r2.length = 4);
280
281     assert(observer.args.length === 1);
282     assert(observer.args[0][0] === r2);
283     assert(observer.args[0][1] === 'add');
284     assert(observer.args[0][2] === r2[0]);
285     assert(observer.args[0][3] === 0);
286
287     // update 2
288     observer.args = [];
289     var post = postStore.get(7);
290     post.slug = 't11';
291     postStore.getLocalStore().update(post);
292     assert(expectOrderedPks(r1, [4, 3, 2, 1]));
293     assert(r1.length = 4);
294     assert(expectOrderedPks(r2, [4, 3, 2, 1]));
295     assert(r2.length = 4);
296     assert(observer.args.length === 0);
297
298     // update 3
299     observer.args = [];
300     var post = postStore.get(7);
301     post.title = 't17';
302     postStore.getLocalStore().update(post);
303     assert(expectOrderedPks(r1, [3, 2, 1]));
304     assert(r1.length = 3);
305     assert(expectOrderedPks(r2, [3, 2, 1]));
306     assert(r2.length = 3);
307
308     assert(observer.args.length === 1);
309     assert(observer.args[0][0] === r2);
310     assert(observer.args[0][1] === 'delete');
311     assert(observer.args[0][2] === authorStore.get(4));
312     assert(observer.args[0][3] === 0);
313
314     // delete
315     observer.args = [];
316     var post = postStore.get(7);
317     postStore.getLocalStore().delete(post);
318     assert(expectOrderedPks(r1, [3, 2, 1]));
319     assert(r1.length = 3);
320     assert(expectOrderedPks(r2, [3, 2, 1]));
321     assert(r2.length = 3);
322     assert(observer.args.length === 0);
323
324     // delete
325     observer.args = [];
326     var post = postStore.get(6);
327     postStore.getLocalStore().delete(post);
328     assert(expectOrderedPks(r1, [2, 1]));
329     assert(r1.length = 2);
330     assert(expectOrderedPks(r2, [2, 1]));
331     assert(r2.length = 2);
332     assert(observer.args.length === 1);
333     assert(observer.args[0][0] === r2);
334     assert(observer.args[0][1] === 'delete');
335     assert(observer.args[0][2] === authorStore.get(3));
```

(continues on next page)

(continued from previous page)

```
336     assert(observer.args[0][3] === 0);
337
338     registry.destroy();
339     resolve();
340   }
341
342
343   function testResult(resolve, reject) {
344     store.when(store.whenIter([testResultReaction, testResultAttachByAttr, ↵
↵testResultRelation], function(suite) {
345       return new Promise(suite);
346     }), function() {
347       resolve();
348     });
349   }
350
351
352   return testResult;
353 });
```


CHAPTER 15

Contributing

Please, use Dojo Style Guide and Dojo contributing workflow.

CHAPTER 16

Indices and tables

- `genindex`
- `modindex`
- `search`

I

IRegistryObserver() (class), 26
 IRegistryObserver.update() (IRegistryObserver method), 27
 IStoreObserver() (class), 21
 IStoreObserver.update() (IStoreObserver method), 21

O

Observable() (class), 29
 Observable.Observable.prototype.attach() (Observable.Observable.prototype method), 29
 Observable.Observable.prototype.detach() (Observable.Observable.prototype method), 29
 Observable.Observable.prototype.get() (Observable.Observable.prototype method), 29
 Observable.Observable.prototype.isObservable() (Observable.Observable.prototype method), 30
 Observable.Observable.prototype.notify() (Observable.Observable.prototype method), 30
 Observable.Observable.prototype.set() (Observable.Observable.prototype method), 29

R

Registry() (class), 25
 Registry.register() (Registry method), 25
 Registry.Registry.prototype.begin() (Registry.Registry.prototype method), 26
 Registry.Registry.prototype.clean() (Registry.Registry.prototype method), 26
 Registry.Registry.prototype.commit() (Registry.Registry.prototype method), 26
 Registry.Registry.prototype.destroy() (Registry.Registry.prototype method), 26
 Registry.Registry.prototype.get() (Registry.Registry.prototype method), 25
 Registry.Registry.prototype.getStores() (Registry.Registry.prototype method), 25
 Registry.Registry.prototype.has() (Registry.Registry.prototype method), 25

Registry.Registry.prototype.keys() (Registry.Registry.prototype method), 25
 Registry.Registry.prototype.observed() (Registry.Registry.prototype method), 26
 Registry.Registry.prototype.ready() (Registry.Registry.prototype method), 26
 Registry.Registry.prototype.rollback() (Registry.Registry.prototype method), 26
 registryObserver() (built-in function), 26
 Result() (class), 23
 Result.Result.prototype.addRelatedSubject() (Result.Result.prototype method), 23
 Result.Result.prototype.observe() (Result.Result.prototype method), 23
 Result.Result.prototype.observed() (Result.Result.prototype method), 23
 ResultObservable() (class), 33
 ResultObservable.ResultObservable.prototype (ResultObservable.ResultObservable attribute), 33
 ResultObservable.ResultObservable.prototype.attachByAttr() (ResultObservable.ResultObservable.prototype method), 33

S

Store() (class), 17
 Store.Store.prototype.add() (Store.Store.prototype method), 19
 Store.Store.prototype.clean() (Store.Store.prototype method), 20
 Store.Store.prototype.compose() (Store.Store.prototype method), 19
 Store.Store.prototype.decompose() (Store.Store.prototype method), 20
 Store.Store.prototype.delete() (Store.Store.prototype method), 19
 Store.Store.prototype.destroy() (Store.Store.prototype method), 20
 Store.Store.prototype.find() (Store.Store.prototype method), 19

Store.Store.prototype.get() (Store.Store.prototype method), 19

Store.Store.prototype.observed() (Store.Store.prototype method), 20

Store.Store.prototype.pull() (Store.Store.prototype method), 19

Store.Store.prototype.register() (Store.Store.prototype method), 20

Store.Store.prototype.save() (Store.Store.prototype method), 19

Store.Store.prototype.update() (Store.Store.prototype method), 19

StoreObservable() (class), 31

StoreObservable.StoreObservable.prototype (StoreObservable.StoreObservable attribute), 31

StoreObservable.StoreObservable.prototype.attachByAttr() (StoreObservable.StoreObservable.prototype method), 31

storeObserver() (built-in function), 20

SubResult() (class), 24