Store.js Documentation

JOOR Inc.

Oct 27, 2018

Contents:

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/

CHAPTER 3

Articles

- Article (in English) "Implementation of the pattern Repository for browser's JavaScript"
- Article (in Russian): " Repository JavaScript"

Contents

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).

CHAPTER 5

Implemented Patterns

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

CHAPTER 6

Used programming paradigms

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

CHAPTER 7

Store

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: {
        fistManyToManyName: {
            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.`**`observed`**`()`
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 `Store.prototype.add()`
"update"	on object is updated in store, triggered by `Store.prototype.update()`
"delete"	on object is deleted from store, triggered by `Store.prototype.delete()`
"restoreObject"	on object is restored, triggered by `Store.prototype.delete()`
"destroy"	immediately before store is destroyed, triggered by `Store.prototype.destroy()` 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 `Store.prototype.add()`
"preUpdate"	before object is updated in store, triggered by `Store.prototype.update()`
"preDelete"	before object is deleted from store, triggered by `Store.prototype.delete()`

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”.

CHAPTER 8

Result

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 it's 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.**observed**()

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

CHAPTER 9

Registry

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. Usualy 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 uncommited 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.`**`observed`**`()`
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.

CHAPTER 10

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 is class of current instance is not DummyObservable.

Return type Boolean

CHAPTER 11

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

CHAPTER 12

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**

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

CHAPTER 13

Query Object

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}
```

CHAPTER 14

Examples

14.1 Query

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert,
        expectPks = utils.expectPks;


    function testQuery(resolve, reject) {
        var registry = new store.Registry();

        function Post(attrs) {
            store.clone(attrs, this);
        }
        Post.prototype = {
            constructor: Post,
            getSlug: function() {
                return this.slug;
            }
        };

        var postStore = new store.Store({
            indexes: ['slug', 'author'],
            remoteStore: new store.DummyStore(),
            model: Post
        });
        registry.register('post', postStore);

        registry.ready();


```

(continues on next page)

(continued from previous page)

```
        var posts = [
            new Post({id: 1, slug: 'sl1', title: 'tl1', author: 1}),
            new Post({id: 2, slug: 'sl1', title: 'tl2', author: 1}),  // slug can be unique per date
            new Post({id: 3, slug: 'sl3', title: 'tl1', author: 2}),
            new Post({id: 4, slug: 'sl4', title: 'tl4', author: 3})
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().add(post); });

        var r;

        r = registry.get('post').find({slug: 'sl1'});
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({getSlug: 'sl1'});
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({slug: 'sl1', author: 1});
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({author: {'$ne': 1}});
        assert(expectPks(r, [3, 4]));

        r = registry.get('post').find({'$callable': function(post) { return post.author === 1; }});
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({author: function(author_id) { return author_id === 1; }});
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({'$and': [{slug: 'sl1'}, {author: 1}]});
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({'$or': [{slug: 'sl1'}, {author: 2}]});
        assert(expectPks(r, [1, 2, 3]));

        r = registry.get('post').find({'$or': [{slug: 'sl1'}, {title: 'tl1'}]}); // No index
        assert(expectPks(r, [1, 2, 3]));

        r = registry.get('post').find({
            '$and': [
                {
                    '$or': [
                        {slug: 'sl1'},
                        {slug: 'sl2'}
                    ]
                },
                {author: 1}
            ]
        });
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({slug: {'$in': ['sl1', 'sl3']}});
        assert(expectPks(r, [1, 2, 3]));
```

(continues on next page)

(continued from previous page)

```

        registry.destroy();
        resolve();
    }
    return testQuery;
});
```

14.2 Simple relations

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert,
        expectPks = utils.expectPks;


    function testSimpleRelations(resolve, reject) {
        var registry = new store.Registry();

        var postStore = new store.Store({
            indexes: ['slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: 'author',
                        relatedStore: 'author',
                        relatedField: 'id',
                        relatedName: 'posts',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);

        var authorStore = new store.Store({
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);

        registry.ready();

        var authors = [
            {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
            {id: 2, firstName: 'Fn1', lastName: 'Ln2'},
            {id: 3, firstName: 'Fn3', lastName: 'Ln1'}
        ];
        store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });

        var posts = [
```

(continues on next page)

(continued from previous page)

```
            {id: 1, slug: 'sl1', title: 'tl1', author: 1},
            {id: 2, slug: 'sl1', title: 'tl2', author: 1},  // slug can be unique per date
            {id: 3, slug: 'sl3', title: 'tl1', author: 2},
            {id: 4, slug: 'sl4', title: 'tl4', author: 3}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().add(post); });

        var r = registry.get('post').find({slug: 'sl1'});
        assert(expectPks(r, [1, 2]));

        var author = registry.get('author').get(1);
        r = registry.get('post').find({'author': author});
        assert(expectPks(r, [1, 2]));

        r = registry.get('post').find({'author.firstName': 'Fn1'});
        assert(expectPks(r, [1, 2, 3]));

        r = registry.get('post').find({author: {'$rel': {firstName: 'Fn1'}}});
        assert(expectPks(r, [1, 2, 3]));

        r = registry.get('author').find({'posts.slug': {'$in': ['sl1', 'sl3']}});
        assert(expectPks(r, [1, 2]));

        r = registry.get('author').find({posts: {'$rel': {slug: {'$in': ['sl1', 'sl3']}}}});
        assert(expectPks(r, [1, 2]));


        // Add
        var post = {id: 5, slug: 'sl5', title: 'tl5', author: 3};
        return registry.get('post').add(post).then(function(post) {
            assert(5 in registry.get('post').getLocalStore().pkIndex);
            assert(registry.get('post').getLocalStore().indexes['slug']['sl5'].indexOf(post) !== -1);


            // Update
            post = registry.get('post').get(5);
            post.slug = 'sl5.2';
            return registry.get('post').update(post).then(function(post) {
                assert(5 in registry.get('post').getLocalStore().pkIndex);
                assert(registry.get('post').getLocalStore().indexes['slug']['sl5.2'].indexOf(post) !== -1);
                assert(registry.get('post').getLocalStore().indexes['slug']['sl5'].indexOf(post) === -1);


                // Delete
                var author = registry.get('author').get(1);
                post = registry.get('post').find({author: 1})[0];
                assert(registry.get('post').getLocalStore().indexes['slug']['sl1'].indexOf(post) !== -1);
                assert(1 in registry.get('post').getLocalStore().pkIndex);
                return registry.get('author').delete(author).then(function() {
                    assert(registry.get('post').getLocalStore().indexes['slug']['sl1'].indexOf(post) === -1);
```

(continues on next page)

(continued from previous page)

```
                    assert(!(1 in registry.get('post').getLocalStore().pkIndex));
                    var r = registry.get('author').find();
                    assert(expectPks(r, [2, 3]));
                    r = registry.get('post').find();
                    assert(expectPks(r, [3, 4, 5]));

                    registry.destroy();
                    // resolve();
                });
            });
        });
    }
    return testSimpleRelations;
});
```

14.3 Composite relations

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert,
        expectPks = utils.expectPks;


    function testCompositeRelations(resolve, reject) {
        var registry = new store.Registry();

        // Use reverse order of store creation.
        var authorStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);

        var postStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['lang', 'slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: ['author', 'lang'],
                        relatedStore: 'author',
                        relatedField: ['id', 'lang'],
                        relatedName: 'posts',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);

```

(continues on next page)

(continued from previous page)

```
        registry.ready();

        var authors = [
            {id: 1, lang: 'en', firstName: 'Fn1', lastName: 'Ln1'},
            {id: 1, lang: 'ru', firstName: 'Fn1-ru', lastName: 'Ln1-ru'},
            {id: 2, lang: 'en', firstName: 'Fn1', lastName: 'Ln2'},
            {id: 3, lang: 'en', firstName: 'Fn3', lastName: 'Ln1'}
        ];
        store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });

        var posts = [
            {id: 1, lang: 'en', slug: 'sl1', title: 'tl1', author: 1},
            {id: 1, lang: 'ru', slug: 'sl1-ru', title: 'tl1-ru', author: 1},
            {id: 2, lang: 'en', slug: 'sl1', title: 'tl2', author: 1},  // slug can␣
↪be unique per date
            {id: 3, lang: 'en', slug: 'sl3', title: 'tl1', author: 2},
            {id: 4, lang: 'en', slug: 'sl4', title: 'tl4', author: 3}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });

        var compositePkAccessor = function(o) { return [o.id, o.lang]; };

        var r = postStore.find({slug: 'sl1'});
        assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));

        var author = registry.get('author').get([1, 'en']);
        r = registry.get('post').find({'author': author});
        assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));

        r = postStore.find({'author.firstName': 'Fn1'});
        assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));

        r = postStore.find({author: {'$rel': {firstName: 'Fn1'}}});
        assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));

        r = authorStore.find({'posts.slug': {'$in': ['sl1', 'sl3']}});
        assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));

        r = authorStore.find({posts: {'$rel': {slug: {'$in': ['sl1', 'sl3']}}}});
        assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));

        // Add
        var post = {id: 5, lang: 'en', slug: 'sl5', title: 'tl5', author: 3};
        postStore.add(post).then(function(post) {
            assert([5, 'en'] in postStore.getLocalStore().pkIndex);
            assert(postStore.getLocalStore().indexes['slug']['sl5'].indexOf(post) !==␣
↪-1);


            // Update
            var post = postStore.get([5, 'en']);
            post.slug = 'sl5.2';
            postStore.update(post).then(function(post) {
                assert([5, 'en'] in postStore.getLocalStore().pkIndex);
                assert(postStore.getLocalStore().indexes['slug']['sl5.2'].
↪indexOf(post) !== -1);
```

(continues on next page)

(continued from previous page)

```
                assert(postStore.getLocalStore().indexes['slug']['sl5'].indexOf(post)
↪=== -1);


                // Delete
                var author = authorStore.get([1, 'en']);
                post = postStore.find({author: 1, lang: 'en'})[0];
                assert(postStore.getLocalStore().indexes['slug']['sl1'].indexOf(post)
↪!== -1);
                assert([1, 'en'] in postStore.getLocalStore().pkIndex);
                authorStore.delete(author).then(function(post) {
                    assert(postStore.getLocalStore().indexes['slug']['sl1'].
↪indexOf(post) === -1);
                    assert(!([1, 'en'] in postStore.getLocalStore().pkIndex));
                    var r = authorStore.find();
                    assert(expectPks(r, [[1, 'ru'], [2, 'en'], [3, 'en']],
↪compositePkAccessor));
                    r = postStore.find();
                    assert(expectPks(r, [[1, 'ru'], [3, 'en'], [4, 'en'], [5, 'en']],
↪compositePkAccessor));

                    registry.destroy();
                    resolve();
                });
            });
        });
    }
    return testCompositeRelations;
});
```

14.4 Many to many relations

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert,
        expectPks = utils.expectPks;


    function testManyToMany(resolve, reject) {
        var registry = new store.Registry();

        var tagStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['slug'],
            remoteStore: new store.DummyStore()
        });
        registry.register('tag', tagStore);

        var tagPostStore = new store.Store({
            relations: {
                foreignKey: {
                    post: {
```

(continues on next page)

(continued from previous page)

```
                        field: ['postId', 'postLang'],
                        relatedStore: 'post',
                        relatedField: ['id', 'lang'],
                        relatedName: 'tagPostSet',
                        onDelete: store.cascade
                    },
                    tag: {
                        field: ['tagId', 'tagLang'],
                        relatedStore: 'tag',
                        relatedField: ['id', 'lang'],
                        relatedName: 'tagPostSet',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        tagPostStore.getLocalStore().setNextPk = function(obj) {
            tagPostStore._pkCounter || (tagPostStore._pkCounter = 0);
            this.getObjectAccessor().setPk(obj, ++tagPostStore._pkCounter);
        };
        registry.register('tagPost', tagPostStore);

        var postStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['lang', 'slug', 'author'],
            relations: {
                manyToMany: {
                    tags: {
                        relation: 'tagPostSet',
                        relatedStore: 'tag',
                        relatedRelation: 'tagPostSet'
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);

        registry.ready();

        var tags = [
            {id: 1, lang: 'en', name: 'T1'},
            {id: 1, lang: 'ru', name: 'T1-ru'},
            {id: 2, lang: 'en', name: 'T1'},
            {id: 3, lang: 'en', name: 'T3'},
            {id: 4, lang: 'en', name: 'T4'}
        ];
        store.whenIter(tags, function(tag) { return tagStore.getLocalStore().add(tag);
↪ });

        var posts = [
            {id: 1, lang: 'en', slug: 'sl1', title: 'tl1'},
            {id: 1, lang: 'ru', slug: 'sl1-ru', title: 'tl1-ru'},
            {id: 2, lang: 'en', slug: 'sl1', title: 'tl2'},  // slug can be unique␣
↪per date
            {id: 3, lang: 'en', slug: 'sl3', title: 'tl1'},
```

(continues on next page)

(continued from previous page)

```
            {id: 4, lang: 'en', slug: 'sl4', title: 'tl4'}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });

        var tagPosts = [
            {postId: 1, postLang: 'en', tagId: 1, tagLang: 'en'},
            {postId: 1, postLang: 'ru', tagId: 1, tagLang: 'ru'},
            {postId: 2, postLang: 'en', tagId: 1, tagLang: 'en'},
            {postId: 3, postLang: 'en', tagId: 2, tagLang: 'en'},
            {postId: 4, postLang: 'en', tagId: 4, tagLang: 'en'}
        ];
        store.whenIter(tagPosts, function(tagPost) { return tagPostStore.
↪getLocalStore().add(tagPost); });

        var compositePkAccessor = function(o) { return [o.id, o.lang]; };
        var r;

        r = postStore.find({slug: 'sl1'});
        assert(expectPks(r, [[1, 'en'], [2, 'en']], compositePkAccessor));

        r = postStore.find({'tags.name': 'T1'});
        assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));

        r = postStore.find({tags: {'$rel': {name: 'T1'}}});
        assert(expectPks(r, [[1, 'en'], [2, 'en'], [3, 'en']], compositePkAccessor));

        registry.destroy();
        resolve();
    }
    return testManyToMany;
});
```

14.5 Compose

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert,
        expectPks = utils.expectPks;


    function testCompose(resolve, reject) {
        var registry = new store.Registry();

        var authorStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);

        var tagStore = new store.Store({
```

(continues on next page)

(continued from previous page)

```
            pk: ['id', 'lang'],
            indexes: ['slug'],
            remoteStore: new store.DummyStore()
        });
        registry.register('tag', tagStore);

        var tagPostStore = new store.Store({
            relations: {
                foreignKey: {
                    post: {
                        field: ['postId', 'postLang'],
                        relatedStore: 'post',
                        relatedField: ['id', 'lang'],
                        relatedName: 'tagPostSet',
                        onDelete: store.cascade
                    },
                    tag: {
                        field: ['tagId', 'tagLang'],
                        relatedStore: 'tag',
                        relatedField: ['id', 'lang'],
                        relatedName: 'tagPostSet',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        tagPostStore.getLocalStore().setNextPk = function(obj) {
            tagPostStore._pkCounter || (tagPostStore._pkCounter = 0);
            this.getObjectAccessor().setPk(obj, ++tagPostStore._pkCounter);
        };
        registry.register('tagPost', tagPostStore);

        var postStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['lang', 'slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: ['author', 'lang'],
                        relatedStore: 'author',
                        relatedField: ['id', 'lang'],
                        relatedName: 'posts',
                        onDelete: store.cascade
                    }
                },
                manyToMany: {
                    tags: {
                        relation: 'tagPostSet',
                        relatedStore: 'tag',
                        relatedRelation: 'tagPostSet'
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);
```

(continues on next page)

(continued from previous page)

```
77
78          registry.ready();
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: 'sl1', title: 'tl1', author: 1},
99              {id: 1, lang: 'ru', slug: 'sl1-ru', title: 'tl1-ru', author: 1},
100             {id: 2, lang: 'en', slug: 'sl1', title: 'tl2', author: 1},  // slug can␣
    ↪be unique per date
101             {id: 3, lang: 'en', slug: 'sl3', title: 'tl1', author: 2},
102             {id: 4, lang: 'en', slug: 'sl4', title: 'tl4', 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": "sl1", "title": "tl1", "author":1,
    ↪"tags": [
123              *         {"id": 1, "lang": "en", "name": "T1"}
124              *     ]},
125              *     {"id": 2, "lang": "en", "slug": "sl1", "title": "tl2", "author": 1,
    ↪ "tags":[
126              *         {"id": 1, "lang": "en", "name": "T1"}
```

(continues on next page)

(continued from previous page)

```
             *     ]}
             * ]}"
             */
            var compositePkAccessor = function(o) { return [o.id, o.lang]; };
            assert(expectPks(author.posts, [[1, 'en'], [2, 'en']],␣
↪compositePkAccessor));
            assert(expectPks(author.posts[0].tags, [[1, 'en']], compositePkAccessor));
            assert(expectPks(author.posts[1].tags, [[1, 'en']], compositePkAccessor));

            registry.destroy();
            resolve();
        });
    }
    return testCompose;
});
```

14.6 Decompose

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert,
        expectPks = utils.expectPks;


    function testDecompose(resolve, reject) {
        var registry = new store.Registry();

        var categoryStore = new store.Store({
            pk: ['id', 'lang'],
            remoteStore: new store.DummyStore()
        });
        registry.register('category', categoryStore);

        var authorStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);

        var tagStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['slug'],
            remoteStore: new store.DummyStore()
        });
        registry.register('tag', tagStore);

        var tagPostStore = new store.Store({
            relations: {
                foreignKey: {
                    post: {
                        field: ['postId', 'postLang'],
```

(continues on next page)

(continued from previous page)

```
                        relatedStore: 'post',
                        relatedField: ['id', 'lang'],
                        relatedName: 'tagPostSet',
                        onDelete: store.cascade
                    },
                    tag: {
                        field: ['tagId', 'tagLang'],
                        relatedStore: 'tag',
                        relatedField: ['id', 'lang'],
                        relatedName: 'tagPostSet',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        tagPostStore.getLocalStore().setNextPk = function(obj) {
            tagPostStore._pkCounter || (tagPostStore._pkCounter = 0);
            this.getObjectAccessor().setPk(obj, ++tagPostStore._pkCounter);
        };
        registry.register('tagPost', tagPostStore);

        var postStore = new store.Store({
            pk: ['id', 'lang'],
            indexes: ['lang', 'slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: ['author', 'lang'],
                        relatedStore: 'author',
                        relatedField: ['id', 'lang'],
                        relatedName: 'posts',
                        onDelete: store.cascade
                    },
                    category: {
                        field: ['category_id', 'lang'],
                        relatedStore: 'category',
                        relatedField: ['id', 'lang'],
                        relatedName: 'posts',
                        onDelete: store.cascade
                    }
                },
                manyToMany: {
                    tags: {
                        relation: 'tagPostSet',
                        relatedStore: 'tag',
                        relatedRelation: 'tagPostSet'
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);

        registry.ready();

        var author = {
```

(continues on next page)

(continued from previous page)

```
            id: 1,
            lang: 'en',
            firstName: 'Fn1',
            lastName: 'Ln1',
            posts: [
                {
                    id: 2,
                    lang: 'en',
                    slug: 'sl1',
                    title: 'tl1',
                    category: {id: 8, lang: 'en', name: 'C1'},
                    tags: [
                        {id: 5, lang: 'en', name: 'T1'},
                        {id: 6, lang: 'en', name: 'T1'}
                    ]
                },
                {
                    id: 3,
                    lang: 'en',
                    slug: 'sl1',
                    title: 'tl2',
                    category: {id: 9, lang: 'en', name: 'C2'},
                    tags: [
                        {id: 5, lang: 'en', name: 'T1'},
                        {id: 7, lang: 'en', name: 'T3'}
                    ]
                }
            ]
        };

        store.when(authorStore.decompose(author), function(author) {
            var compositePkAccessor = function(o) { return [o.id, o.lang]; };
            var r;
            r = authorStore.find();
            assert(expectPks(r, [[1, 'en']], compositePkAccessor));
            r = postStore.find();
            assert(expectPks(r, [[2, 'en'], [3, 'en']], compositePkAccessor));
            for (var i = 0; i < r.length; i++) {
                assert(r[i].author === 1);
            }
            r = tagStore.find();
            assert(expectPks(r, [[5, 'en'], [6, 'en'], [7, 'en']],␣
↪compositePkAccessor));
            r = tagPostStore.find({postId: 2, postLang: 'en', tagId: 5, tagLang: 'en'}
↪);
            assert(r.length === 1);
            r = tagPostStore.find({postId: 2, postLang: 'en', tagId: 6, tagLang: 'en'}
↪);
            assert(r.length === 1);
            r = tagPostStore.find({postId: 3, postLang: 'en', tagId: 5, tagLang: 'en'}
↪);
            assert(r.length === 1);
            r = tagPostStore.find({postId: 3, postLang: 'en', tagId: 7, tagLang: 'en'}
↪);
            assert(r.length === 1);

            r = categoryStore.find();
```

(continues on next page)

(continued from previous page)

```
            assert(expectPks(r, [[8, 'en'], [9, 'en']], compositePkAccessor));

            assert(author.posts[0].id === 2);
            assert(author.posts[0].author === 1);
            assert(author.posts[1].id === 3);
            assert(author.posts[1].author === 1);
            assert(author.posts.length === 2);

            assert(author.posts[0].tags[0].id === 5);
            assert(author.posts[0].tags[1].id === 6);
            assert(author.posts[0].tags.length === 2);

            assert(author.posts[1].tags[0].id === 5);
            assert(author.posts[1].tags[1].id === 7);
            assert(author.posts[1].tags.length === 2);

            assert(author.posts[0].category_id === 8);
            assert(author.posts[1].category_id === 9);

            registry.destroy();
            resolve();
        });
    }
    return testDecompose;
});
```

14.7 Observable object

Example of fast real-time aggregation:

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert;


    function testObservable(resolve, reject) {
        // Example of fast real-time aggregation
        var registry = new store.Registry();
        registry.observed().attach('register', function(aspect, newStore) {
            newStore.getLocalStore().observed().attach('add', function(aspect, obj) {␣
↪store.observe(obj); });
        });

        var postStore = new store.Store({
            indexes: ['slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: 'author',
                        relatedStore: 'author',
                        relatedField: 'id',
                        relatedName: 'posts',
```

(continues on next page)

(continued from previous page)

```
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);

        var authorStore = new store.Store({
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);
        registry.observed().attach('ready', function() {
            registry.get('post').getLocalStore().observed().attach('add',␣
↪function(aspect, post) {
                registry.get('author').find({id: post.author}).
↪forEach(function(author) {
                    author.observed().set('views_total', author.views_total + post.
↪views_count);
                    post.observed().attach('views_count', function(name, oldValue,␣
↪newValue) {
                        author.observed().set('views_total', author.views_total -␣
↪oldValue + newValue);
                    });
                });
            });
        });

        registry.ready();

        var authors = [
            {id: 1, firstName: 'Fn1', lastName: 'Ln1', views_total: 0},
            {id: 2, firstName: 'Fn1', lastName: 'Ln2', views_total: 0},
            {id: 3, firstName: 'Fn3', lastName: 'Ln1', views_total: 0}
        ];
        store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });

        var posts = [
            {id: 1, slug: 'sl1', title: 'tl1', author: 1, views_count: 5},
            {id: 2, slug: 'sl1', title: 'tl2', author: 1, views_count: 6},  // slug␣
↪can be unique per date
            {id: 3, slug: 'sl3', title: 'tl1', author: 2, views_count: 7},
            {id: 4, slug: 'sl4', title: 'tl4', author: 3, views_count: 8}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });

        var author = registry.get('author').get(1);
        assert(author.views_total === 11);
        var post = registry.get('post').find({author: author.id})[0];
        post.observed().set('views_count', post.views_count + 1);
        assert(author.views_total === 12);

        postStore.getLocalStore().add({id: 5, slug: 'sl5', title: 'tl5', author: 1,␣
↪views_count: 8});
```

(continues on next page)

(continued from previous page)

```
        assert(author.views_total === 20);
        resolve();

    }
    return testObservable;
});
```

14.8 StoreObservable

Example of fast real-time aggregation using :

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert;


    function testStoreObservable(resolve, reject) {
        var registry = new store.Registry();


        var postStore = new store.Store({
            indexes: ['slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: 'author',
                        relatedStore: 'author',
                        relatedField: 'id',
                        relatedName: 'posts',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);

        registry.get('post').getLocalStore().observed().attachByAttr('views_count', 0,
↪ function(attr, oldValue, newValue) {
            var post = this;
            var author = registry.get('author').get(post.author);
            author.views_total = (author.views_total || 0) + newValue - oldValue;
            registry.get('author').getLocalStore().update(author);
        });


        var authorStore = new store.Store({
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);

```

(continues on next page)

(continued from previous page)

```
        registry.ready();

        var authors = [
            {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
            {id: 2, firstName: 'Fn2', lastName: 'Ln2'},
            {id: 3, firstName: 'Fn3', lastName: 'Ln1'}
        ];
        store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });

        var posts = [
            {id: 1, slug: 'sl1', title: 'tl1', author: 1, views_count: 5},
            {id: 2, slug: 'sl1', title: 'tl2', author: 1, views_count: 6},  // slug␣
↪can be unique per date
            {id: 3, slug: 'sl3', title: 'tl1', author: 2, views_count: 7},
            {id: 4, slug: 'sl3', title: 'tl1', author: 2, views_count: 8},
            {id: 5, slug: 'sl4', title: 'tl4', author: 3, views_count: 9}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });

        var author = registry.get('author').get(1);
        assert(author.views_total === 11);

        // update
        var post = registry.get('post').find({author: author.id})[0];
        post.views_count += 1;
        registry.get('post').getLocalStore().update(post);
        assert(author.views_total === 12);

        // add
        registry.get('post').getLocalStore().add(
            {id: 6, slug: 'sl6', title: 'tl6', author: 1, views_count: 10}
        );
        assert(author.views_total === 22);

        // delete
        registry.get('post').getLocalStore().delete(
            registry.get('post').get(6)
        );
        assert(author.views_total === 12);
        resolve();
    }
    return testStoreObservable;
});
```

14.9 Reaction of Result on changes in Store

```
define(['../store', './utils'], function(store, utils) {

    'use strict';

    var assert = utils.assert,
```

(continues on next page)

(continued from previous page)

```
        expectPks = utils.expectPks,
        expectOrderedPks = utils.expectOrderedPks;


    function testResultReaction(resolve, reject) {
        var registry = new store.Registry();

        var postStore = new store.Store({
            indexes: ['slug', 'author'],
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);

        registry.ready();

        var posts = [
            {id: 1, slug: 'sl1', title: 'tl1', author: 1},
            {id: 2, slug: 'sl1', title: 'tl2', author: 1},  // slug can be unique per date
            {id: 3, slug: 'sl3', title: 'tl1', author: 2},
            {id: 4, slug: 'sl4', title: 'tl4', author: 3}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().add(post); });

        var r1 = registry.get('post').find({author: 1});
        r1.observe();
        assert(expectPks(r1, [1, 2]));
        assert(r1.length = 2);

        r1.sort(function(a, b){ return b.id - a.id; });
        assert(expectOrderedPks(r1, [2, 1]));

        var r2 = r1.slice();
        assert(expectOrderedPks(r2, [2, 1]));
        assert(r2.length = 2);


        var observer = function(aspect, obj) {
            observer.args.push([this].concat(Array.prototype.slice.call(arguments)));
        };
        observer.args = [];
        r2.observed().attach(['add', 'update', 'delete'], observer);

        // add
        postStore.getLocalStore().add({id: 5, slug: 'sl5', title: 'tl5', author: 1});
        assert(expectOrderedPks(r1, [5, 2, 1]));
        assert(r1.length = 3);
        assert(expectOrderedPks(r2, [5, 2, 1]));
        assert(r2.length = 3);

        assert(observer.args.length === 1);
        assert(observer.args[0][0] === r2);
        assert(observer.args[0][1] === 'add');
        assert(observer.args[0][2] === r2[0]);

        observer.args = [];
```

(continues on next page)

(continued from previous page)

```
        postStore.getLocalStore().add({id: 6, slug: 'sl6', title: 'tl6', author: 2});
        assert(expectOrderedPks(r1, [5, 2, 1]));
        assert(r1.length = 3);
        assert(expectOrderedPks(r2, [5, 2, 1]));
        assert(r2.length = 3);
        assert(observer.args.length === 0);

        // update
        observer.args = [];
        postStore.getLocalStore().update(postStore.get(5));
        assert(expectOrderedPks(r1, [5, 2, 1]));
        assert(r1.length = 3);
        assert(expectOrderedPks(r2, [5, 2, 1]));
        assert(r2.length = 3);

        assert(observer.args.length === 1);
        assert(observer.args[0][0] === r2);
        assert(observer.args[0][1] === 'update');
        assert(observer.args[0][2] === r2[0]);
        assert(observer.args[0][3].id === 5);

        // delete
        observer.args = [];
        postStore.getLocalStore().delete(postStore.get(5));
        assert(expectOrderedPks(r1, [2, 1]));
        assert(r1.length = 2);
        assert(expectOrderedPks(r2, [2, 1]));
        assert(r2.length = 2);

        assert(observer.args.length === 1);
        assert(observer.args[0][0] === r2);
        assert(observer.args[0][1] === 'delete');
        assert(observer.args[0][2].id === 5);

        registry.destroy();
        resolve();
    }


    function testResultAttachByAttr(resolve, reject) {
        var registry = new store.Registry();


        var postStore = new store.Store({
            indexes: ['slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: 'author',
                        relatedStore: 'author',
                        relatedField: 'id',
                        relatedName: 'posts',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
```

(continues on next page)

(continued from previous page)

```
        });
        registry.register('post', postStore);


        var authorStore = new store.Store({
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);

        registry.get('author').getLocalStore().observed().attach('add', function(aspect, author) {
            author.views_total = 0;
            registry.get('post').find({
                'author.id': author.id
            }).observe().forEachByAttr('views_count', 0, function(attr, oldValue, newValue) {
                author.views_total = author.views_total + newValue - oldValue;
                registry.get('author').getLocalStore().update(author);
            });
        });


        registry.ready();

        var authors = [
            {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
            {id: 2, firstName: 'Fn2', lastName: 'Ln2'},
            {id: 3, firstName: 'Fn3', lastName: 'Ln1'}
        ];
        store.whenIter(authors, function(author) { return authorStore.getLocalStore().add(author); });

        var posts = [
            {id: 1, slug: 'sl1', title: 'tl1', author: 1, views_count: 5},
            {id: 2, slug: 'sl1', title: 'tl2', author: 1, views_count: 6},  // slug can be unique per date
            {id: 3, slug: 'sl3', title: 'tl1', author: 2, views_count: 7},
            {id: 4, slug: 'sl3', title: 'tl1', author: 2, views_count: 8},
            {id: 5, slug: 'sl4', title: 'tl4', author: 3, views_count: 9}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().add(post); });

        var author = registry.get('author').get(1);
        assert(author.views_total === 11);

        // update
        var post = registry.get('post').find({author: author.id})[0];
        post.views_count += 1;
        registry.get('post').getLocalStore().update(post);
        assert(author.views_total === 12);

        // add
        registry.get('post').getLocalStore().add(
            {id: 6, slug: 'sl6', title: 'tl6', author: 1, views_count: 10}
        );
```

(continues on next page)

(continued from previous page)

```
        assert(author.views_total === 22);

        // delete
        registry.get('post').getLocalStore().delete(
            registry.get('post').get(6)
        );
        assert(author.views_total === 12);
        resolve();
    }


    function testResultRelation(resolve, reject) {
        var registry = new store.Registry();


        var postStore = new store.Store({
            indexes: ['slug', 'author'],
            relations: {
                foreignKey: {
                    author: {
                        field: 'author',
                        relatedStore: 'author',
                        relatedField: 'id',
                        relatedName: 'posts',
                        onDelete: store.cascade
                    }
                }
            },
            remoteStore: new store.DummyStore()
        });
        registry.register('post', postStore);


        var authorStore = new store.Store({
            indexes: ['firstName', 'lastName'],
            remoteStore: new store.DummyStore()
        });
        registry.register('author', authorStore);


        registry.ready();

        var authors = [
            {id: 1, firstName: 'Fn1', lastName: 'Ln1'},
            {id: 2, firstName: 'Fn2', lastName: 'Ln2'},
            {id: 3, firstName: 'Fn3', lastName: 'Ln1'},
            {id: 4, firstName: 'Fn4', lastName: 'Ln4'}
        ];
        store.whenIter(authors, function(author) { return authorStore.getLocalStore().
↪add(author); });

        var posts = [
            {id: 1, slug: 'sl1', title: 'tl1', author: 1, views_count: 5},
            {id: 2, slug: 'sl1', title: 'tl2', author: 1, views_count: 6},  // slug␣
↪can be unique per date
            {id: 3, slug: 'sl3', title: 'tl1', author: 2, views_count: 7},
            {id: 4, slug: 'sl3', title: 'tl1', author: 2, views_count: 8},
```

(continues on next page)

(continued from previous page)

```
            {id: 5, slug: 'sl4', title: 'tl4', author: 3, views_count: 9}
        ];
        store.whenIter(posts, function(post) { return postStore.getLocalStore().
↪add(post); });


        var r1 = registry.get('author').find({'posts.title': 'tl1'});
        r1.observe();
        assert(expectPks(r1, [1, 2]));
        assert(r1.length = 2);

        r1.sort(function(a, b){ return b.id - a.id; });
        assert(expectOrderedPks(r1, [2, 1]));

        var r2 = r1.slice();
        assert(expectOrderedPks(r2, [2, 1]));
        assert(r2.length = 2);


        var observer = function(aspect, obj) {
            observer.args.push([this].concat(Array.prototype.slice.call(arguments)));
        };
        observer.args = [];
        r2.observed().attach(['add', 'update', 'delete'], observer);

        // add
        postStore.getLocalStore().add({id: 6, slug: 'sl6', title: 'tl1', author: 3,␣
↪views_count: 8});
        assert(expectOrderedPks(r1, [3, 2, 1]));
        assert(r1.length = 3);
        assert(expectOrderedPks(r2, [3, 2, 1]));
        assert(r2.length = 3);

        assert(observer.args.length === 1);
        assert(observer.args[0][0] === r2);
        assert(observer.args[0][1] === 'add');
        assert(observer.args[0][2] === r2[0]);
        assert(observer.args[0][3] === 0);

        // add 2
        observer.args = [];
        postStore.getLocalStore().add({id: 7, slug: 'sl7', title: 'tl7', author: 4,␣
↪views_count: 8});
        assert(expectOrderedPks(r1, [3, 2, 1]));
        assert(r1.length = 3);
        assert(expectOrderedPks(r2, [3, 2, 1]));
        assert(r2.length = 3);
        assert(observer.args.length === 0);

        // update
        observer.args = [];
        var post = postStore.get(7);
        post.title = 'tl1';
        postStore.getLocalStore().update(post);
        assert(expectOrderedPks(r1, [4, 3, 2, 1]));
        assert(r1.length = 4);
        assert(expectOrderedPks(r2, [4, 3, 2, 1]));
```

(continues on next page)

(continued from previous page)

```
        assert(r2.length = 4);

        assert(observer.args.length === 1);
        assert(observer.args[0][0] === r2);
        assert(observer.args[0][1] === 'add');
        assert(observer.args[0][2] === r2[0]);
        assert(observer.args[0][3] === 0);

        // update 2
        observer.args = [];
        var post = postStore.get(7);
        post.slug = 'tl1';
        postStore.getLocalStore().update(post);
        assert(expectOrderedPks(r1, [4, 3, 2, 1]));
        assert(r1.length = 4);
        assert(expectOrderedPks(r2, [4, 3, 2, 1]));
        assert(r2.length = 4);
        assert(observer.args.length === 0);

        // update 3
        observer.args = [];
        var post = postStore.get(7);
        post.title = 'tl7';
        postStore.getLocalStore().update(post);
        assert(expectOrderedPks(r1, [3, 2, 1]));
        assert(r1.length = 3);
        assert(expectOrderedPks(r2, [3, 2, 1]));
        assert(r2.length = 3);

        assert(observer.args.length === 1);
        assert(observer.args[0][0] === r2);
        assert(observer.args[0][1] === 'delete');
        assert(observer.args[0][2] === authorStore.get(4));
        assert(observer.args[0][3] === 0);

        // delete
        observer.args = [];
        var post = postStore.get(7);
        postStore.getLocalStore().delete(post);
        assert(expectOrderedPks(r1, [3, 2, 1]));
        assert(r1.length = 3);
        assert(expectOrderedPks(r2, [3, 2, 1]));
        assert(r2.length = 3);
        assert(observer.args.length === 0);

        // delete
        observer.args = [];
        var post = postStore.get(6);
        postStore.getLocalStore().delete(post);
        assert(expectOrderedPks(r1, [2, 1]));
        assert(r1.length = 2);
        assert(expectOrderedPks(r2, [2, 1]));
        assert(r2.length = 2);
        assert(observer.args.length === 1);
        assert(observer.args[0][0] === r2);
        assert(observer.args[0][1] === 'delete');
        assert(observer.args[0][2] === authorStore.get(3));
```

(continues on next page)

(continued from previous page)

```
        assert(observer.args[0][3] === 0);

        registry.destroy();
        resolve();
    }


    function testResult(resolve, reject) {
        store.when(store.whenIter([testResultReaction, testResultAttachByAttr,␣
↪testResultRelation], function(suite) {
            return new Promise(suite);
        }), function() {
            resolve();
        });
    }


    return testResult;
});
```

CHAPTER 15

Contributing

Please, use Dojo Style Guide and Dojo contributing workflow.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Index