# PaulUithol / Backbone-relational



Latest commit to the **master** branch

Support backbone's new syntax for `Backbone.Model.set`, using separat… …

```
…e key/value parameters instead of a single `attributes` object.
```

commit 7efecad0cf
PaulUithol authored 2 days ago
**Backbone-relational** / README.md



100644 322 lines (243 sloc) 13.974 kb



# Backbone-relational

Backbone-relational provides one-to-one, one-to-many and many-to-one relations between models for Backbone. To use relations, extend `Backbone.RelationalModel` (instead of the regular `Backbone.Model`) and define a property `relations`, containing an array of option objects.

Each relation must define (as a minimum) the `type`, `key` and `relatedModel`. Available relation types are `Backbone.HasOne` and `Backbone.HasMany`. Backbone-relational features:

- Bidirectional relations, which notify related models of changes through events.
- Control how relations are serialized using the `includeInJSON` option.
- Automatically convert nested objects in a model's attributes into Model instances using the `createModels` option.
- Retrieve (a set of) related models through the `fetchRelated(key<string>, [options<object>])` method.
- Determine the type of `HasMany` collections with `collectionType`.
- Bind new events to a `Backbone.RelationalModel` for:
  - addition to a `HasMany` relation (bind to `add:<key>`; arguments: `(addedModel, relatedCollection)`),
  - removal from a `HasMany` relation (bind to `remove:<key>`; arguments: `(removedModel, relatedCollection)`),
  - changes to the key itself on `HasMany` and `HasOne` relations (bind to `update:<key>`; arguments=`(model, relatedModel/relatedCollection)`).

# Contents



# Installation

Backbone-relational depends on backbone (and thus on underscore). Include Backbone-relational right after Backbone and Underscore:

```
<script type="text/javascript" src="./js/underscore.js"></script>
<script type="text/javascript" src="./js/backbone.js"></script>
<script type="text/javascript" src="./js/backbone-relational.js"></script>
```

Backbone-relational has been tested with Backbone 0.5.3 (or newer) and Underscore 1.2.1 (or newer).

# Backbone.Relation options

Each `Backbone.RelationalModel` can contain an array of `relations`. Each relation supports a number of options, of which `relatedModel`, `key` and `type` are mandatory. A relation could look like the following:

```
Zoo = Backbone.RelationalModel.extend({
    relations: [{
            type: Backbone.HasMany,
            key: 'animals',
            relatedModel: 'Animal',
            collectionType: 'AnimalCollection',
            reverseRelation: {
                key: 'livesIn',
                includeInJSON: 'id'
                // 'relatedModel' is automatically set to 'Zoo'; the 'relationType' to 'HasOne'.
            }
        }]
});

Animal = Backbone.RelationalModel.extend({
    urlRoot: '/animal/'
});

AnimalCollection = Backbone.Collection.extend({
    model: Animal,

    url: function( models ) {
        return '/animal/' + ( models ? 'set/' + _.pluck( models, 'id' ).join(';') + '/' : '' );
    }
});
```

### relatedModel

Value: a string (which can be resolved to an object type on the global scope), or a reference to a `Backbone.RelationalModel` type.

### key

Value: a string. References an attribute name on `relatedModel`.

### type

Value: a string, or a reference to a `Backbone.Relation` type

Example: `Backbone.HasOne` or `'HasMany'`.

#### HasOne relations (`Backbone.HasOne`)

The key for a `HasOne` relation consists of a single `Backbone.RelationalModel`. The default `reverseRelation.type` for a HasOne relation is HasMany. This can be set to `HasOne` instead, to create a one-to-one relation.

#### HasMany relations (`Backbone.HasMany`)

The key for a `HasMany` relation consists of a `Backbone.Collection`, containing zero or more `Backbone.RelationalModels`. The default `reverseRelation.type` for a HasMany relation is HasOne; this is the only option here, since many-to-many is not supported directly.

#### Many-to-many relations

A many-to-many relation can be modeled using two `Backbone.HasMany` relations, with a link model in between:

```
Person = Backbone.RelationalModel.extend({
    relations: [
        {
            type: 'HasMany',
            key: 'jobs',
            relatedModel: 'Job',
            reverseRelation: {
                key: 'person'
            }
        }
    ]
});

// A link object between 'Person' and 'Company', to achieve many-to-many relations.
Job = Backbone.RelationalModel.extend({
    defaults: {
        'startDate': null,
        'endDate': null
    }
})

Company = Backbone.RelationalModel.extend({
    relations: [
        {
            type: 'HasMany',
            key: 'employees',
            relatedModel: 'Job',
            reverseRelation: {
                key: 'company'
            }
        }
    ]
});

niceCompany = new Company( { name: 'niceCompany' } );
niceCompany.bind( 'add:employees', function( model, coll ) {
        // Will see a Job with attributes { person: paul, company: niceCompany } being added here
    });

paul.get('jobs').add( { company: niceCompany } );
```

### collectionType

Value: a string (which can be resolved to an object type on the global scope), or a reference to a `Backbone.Collection` type.

Determine the type of collections used for a `HasMany` relation. Defining a `url(models<Backbone.Model[]>)` function on this Collection that's able to build a url for either the whole collection, or a set of models enables `fetchRelated` to fetch all missing models in one request, instead of firing a separate request for each. See Backbone-tastypie for an example.

### collectionKey

Value: a string or a boolean

By default, the relation's `key` attribute will be used to create a reference to the RelationalModel instance from the generated collection. If you set `collectionKey` to a string, it will use that string as the reference to the RelationalModel, rather than the relation's `key` attribute. If you don't want this behavior at all, just set `collectionKey` to false (or any falsy value) and this reference will not be created.

### includeInJSON

Value: a boolean, or a string referencing one of the model's attributes. Default: `true`.

Determines how a relation will be serialized following a call to the `toJSON` method. A value of `true` serializes the full set of attributes on the

related model(s), in which case the relations of this object are serialized as well. Set to `false` to exclude the relation completely. You can also choose to include a single attribute from the related model by using a string. For example, `'name'`, or `Backbone.Model.prototype.idAttribute` to include ids.

### createModels

Value: a boolean. Default: `true`.

Should models be created from nested objects, or not?

### reverseRelation

If the relation should be bidirectional, specify the details for the reverse relation here. It's only mandatory to supply a `key`; `relatedModel` is automatically set. The default `type` for a `reverseRelation` is `HasMany` for a `HasOne` relation (which can be overridden to `HasOne` in order to create a one-to-one relation), and `HasOne` for a `HasMany` relation. In this case, you cannot create a reverseRelation with type `HasMany` as well; please see Many-to-many relations on how to model these type of relations.

**Please note**: if you define a relation (plus a `reverseRelation`) on a model, but never actually create an instance of that model, the model's `constructor` will never run, which means it's `initializeRelations` will never get called, and the reverseRelation will not be initialized either. In that case, you could either define the relation on the opposite model, or define two single relations. See issue 20 for a discussion.

## Backbone.RelationalModel

`Backbone.RelationalModel` introduces a couple of new methods and events.

### Methods

#### getRelations `relationalModel.getRelations()`

Returns the set of initialized relations on the model.

#### fetchRelated `relationalModel.fetchRelated(key<string>, [options<object>])`

Fetch models from the server that were referenced in the model's attributes, but have not been found/created yet. This can be used specifically for lazy-loading scenarios.

By default, a separate request will be fired for each additional model that is to be fetched from the server. However, if your server/API supports it, you can fetch the set of models in one request by specifying a `collectionType` for the relation you call `fetchRelated` on. The `collectionType` should have an overridden `url(models<Backbone.Model[]>)` method that allows it to construct a url for an array of models. See the example at the top of Backbone.Relation options or Backbone-tastypie for an example.

### Events

- `add`: triggered on addition to a `HasMany` relation.
  Bind to `add:<key>`; arguments: `(addedModel<Backbone.Model>, related<Backbone.Collection>)`.
- `remove`: triggered on removal from a `HasMany` relation.
  Bind to `remove:<key>`; arguments: `(removedModel<Backbone.Model>, related<Backbone.Collection>)`.
- `update`: triggered on changes to the key itself on `HasMany` and `HasOne` relations.
  Bind to `update:<key>`; arguments: `(model<Backbone.Model>, related<Backbone.Model|Backbone.Collection>)`.

## Example

```
paul = new Person({
    id: 'person-1',
    name: 'Paul',
    user: { id: 'user-1', login: 'dude', email: 'me@gmail.com' }
});

// A User object is automatically created from the JSON; so 'login' returns 'dude'.
paul.get('user').get('login');

ourHouse = new House({
    id: 'house-1',
    location: 'in the middle of the street',
    occupants: ['person-1', 'person-2', 'person-5']
});

// 'ourHouse.occupants' is turned into a Backbone.Collection of Persons.
// The first person in 'ourHouse.occupants' will point to 'paul'.
ourHouse.get('occupants').at(0); // === paul

// If a collection is created from a HasMany relation, it contains a reference
// back to the originator of the relation
```

```
ourHouse.get('occupants').livesIn; // === ourHouse

// the relation from 'House.occupants' to 'Person' has been defined as a bi-directional HasMany relation,
// with a reverse relation to 'Person.livesIn'. So, 'paul.livesIn' will automatically point back to 'ourHouse'.
paul.get('livesIn'); // === ourHouse

// You can control which relations get serialized to JSON (when saving), using the 'includeInJSON'
// property on a Relation. Also, each object will only get serialized once to prevent loops.
paul.get('user').toJSON();
    /* result:
        {
            email: "me@gmail.com",
            id: "user-1",
            login: "dude",
            person: {
                id: "person-1",
                name: "Paul",
                livesIn: {
                    id: "house-1",
                    location: "in the middle of the street",
                    occupants: ["person-1"] // just the id, since 'includeInJSON' references the 'idAttribute'
                },
                user: "user-1" // not serialized because it is already in the JSON, so we won't create a loop
            }
        }
    */

// Load occupants 'person-2' and 'person-5', which don't exist yet, from the server
ourHouse.fetchRelated( 'occupants' );

// Use the 'add' and 'remove' events to listen for additions/removals on HasMany relations (like 'House.occupants').
ourHouse.bind( 'add:occupants', function( model, coll ) {
        // create a View?
        console.debug( 'add %o', model );
    });
ourHouse.bind( 'remove:occupants', function( model, coll ) {
        // destroy a View?
        console.debug( 'remove %o', model );
    });

// Use the 'update' event to listen for changes on a HasOne relation (like 'Person.livesIn').
paul.bind( 'update:livesIn', function( model, attr ) {
        console.debug( 'update to %o', attr );
    });


// Modifying either side of a bi-directional relation updates the other side automatically.
// Make paul homeless; triggers 'remove:occupants' on ourHouse, and 'update:livesIn' on paul
ourHouse.get('occupants').remove( paul.id );

paul.get('livesIn'); // yup; nothing.

// Move back in; triggers 'add:occupants' on ourHouse, and 'update:livesIn' on paul
paul.set( { 'livesIn': 'house-1' } );
```

This is achieved using the following relations and models:

```
House = Backbone.RelationalModel.extend({
    // The 'relations' property, on the House's prototype. Initialized separately for each instance of House.
    // Each relation must define (as a minimum) the 'type', 'key' and 'relatedModel'. Options are
    // 'includeInJSON', 'createModels' and 'reverseRelation', which takes the same options as the relation itself.
    relations: [
        {
            type: Backbone.HasMany, // Use the type, or the string 'HasOne' or 'HasMany'.
            key: 'occupants',
            relatedModel: 'Person',
            includeInJSON: Backbone.Model.prototype.idAttribute,
            collectionType: 'PersonCollection',
            reverseRelation: {
                key: 'livesIn'
            }
        }
    ]
});

Person = Backbone.RelationalModel.extend({
    relations: [
        { // Create a (recursive) one-to-one relationship
            type: Backbone.HasOne,
            key: 'user',
            relatedModel: 'User',
            reverseRelation: {
                type: Backbone.HasOne,
                key: 'person'
            }
        }
    ],
```

```
    initialize: function() {
        // do whatever you want :)
    }
});

PersonCollection = Backbone.Collection.extend({
    url: function( models ) {
        // Logic to create a url for the whole collection, or a set of models.
        // See the tests, or Backbone-tastypie, for an example.
        return '/person/' + ( models ? 'set/' + _.pluck( models, 'id' ).join(';') + '/' : '' );
    }
});

User = Backbone.RelationalModel.extend();
```

# Under the hood

Each `Backbone.RelationalModel` registers itself with `Backbone.Store` upon creation (and is removed from the `Store` when destroyed). When creating or updating an attribute that is a key in a relation, removed related objects are notified of their removal, and new related objects are looked up in the `Store`.



# Markdown Cheat Sheet

## Format Text

Headers

```
# This is an <h1> tag
```

```
## This is an <h2> tag
###### This is an <h6> tag
```

Text styles

```
*This text will be italic*
_This will also be italic_
**This text will be bold**
__This will also be bold__

*You **can** combine them*
```

### Lists

Unordered

```
* Item 1
* Item 2
  * Item 2a
  * Item 2b
```

Ordered

```
1. Item 1
2. Item 2
3. Item 3
   * Item 3a
   * Item 3b
```

### Miscellaneous

Images

```
![GitHub Logo](/images/logo.png)
Format: ![Alt Text](url)
```

Links

```
http://github.com - automatic!
[GitHub](http://github.com)
```

Blockquotes

```
As Kanye West said:

> We're living the future so
> the present is our past.
```

### Code Examples in Markdown

Syntax highlighting with GFM

```
```javascript
function fancyAlert(arg) {
  if(arg) {
    $.facebox({div:'#foo'})
  }
}
```
```

Or, indent your code 4 spaces

```
Here is a Python code example
without syntax highlighting:

    def foo:
      if not bar:
        return true
```

Inline code for comments

```
I think you should use an
`<addr>` element here instead.
```

Something went wrong with that request. Please try again. Dismiss