bookshelf-schema Documentation

Release 0.3.1

Alexander Bogdanov

Nov 13, 2017

Contents

The Bookshelf plugin that adds fields, relations, scopes and more to bookshelf models.

CHAPTER 1

Related plugins

- bookshelf-fields - the ancestor of this plugin
- bookshelf-scopes - the source of inspiration for scopes helpers

CHAPTER 2

Contents:

2.1 Basic usage

2.1.1 CoffeeScript

Enable plugin:

```
Schema = require 'bookshelf-schema'
knex = require('knex')({...})
db = require('bookshelf')(knex)
db.plugin Schema({...})
```

Define model:

```
{StringField, EmailField} = require 'bookshelf-schema/lib/fields'
{HasMany} = require 'bookshelf-schema/lib/relations'
Photo = require './photo'

class User extends db.Model
  tableName: 'users'
  @schema [
    StringField 'username'
    EmailField 'email'
    HasMany Photo
  ]
```

2.1.2 JavaScript

Enable plugin:

```
var Schema = require('bookshelf-schema');
var knex = require('knex')({...});
```

```
var db = require('bookshelf')(knex);
db.plugin(Schema({...}));
```

Define model:

```
var Fields = require('bookshelf-schema/lib/fields'),
    StringField = Fields.StringField,
    EmailField = Fields.EmailField;

var Relations = require('bookshelf-schema/lib/relations'),
    HasMany = Relations.HasMany;

var Photo = require('./photo');

var User = db.Model.extend({ tableName: 'users' }, {
  schema: [
    StringField('username'),
    EmailField('email'),
    HasMany(Photo)
  ]
});
```

2.1.3 Schema definition

Schema passed to `db.Model.schema` method or to a "schema" static field is an array of "schema entities". Each of that entity class defines special methods used in process of augementing and initiaizing model.

The *bookshelf-schema* comes with several predefined classes adding fields, relations, scopes etc. You may see some of them in examples above: StringField, EmailField, HasMany.

You may define your own schema entities with custom behaviour.

2.1.4 Plugin options

Schema (*options = {}*)

Options:

createProperties: Boolean, default true should fields and relations create accessors or not

validation: Boolean enable model validation

relationsAccessorPrefix: String prefix for relations accessors

language, labels, messages are passed to Checkit

2.2 Fields

Note: Exported from bookshelf-schema/lib/fields

Fields enhances models in several ways:

- each field adds an accessor property so instead of calling `model.get('fieldName')` you may use `model.fieldName` directly

- each field may convert data when model is parsed or formatted
- model may use field-specific validation before save or explicitly. Validation uses the Checkit module.

2.2.1 Examples

CoffeeScript

```
{StringField, EncryptedStringField} = require 'bookshelf-schema/lib/fields'

class User extends db.Model
  tableName: 'users'
  @schema [
    StringField 'username', required: true
    EncryptedStringField 'password', minLength: 8
  ]

User.forge(username: 'alice', password: 'secret-password').save()  # [1]
.then (alice) ->
  User.forge(id: alice.id).fetch()
.then (alice) ->
  alice.username.should.equal 'alice'                              # [2]
  alice.password.verify('secret-password').should.become.true      # [3]
```

JavaScript

```
var Fields = require('bookshelf-schema/lib/fields');
var StringField = Fields.StringField;
var EncryptedStringField = Fields.EncryptedStringField;

var User = db.Model.extend( { tableName: 'users' }, {
  schema: [
    StringField('username', {required: true}),
    EncryptedStringField('password', {minLength: 8})
  ]
});

User.forge({username: 'alice', password: 'secret-password'}).save()  // [1]
.then( function(alice) {
  return User.forge({id: alice.id}).fetch()
}).then( function(aclice) {
  alice.username.should.equal('alice');                              // [2]
  alice.password.verify('secret-password').should.become.true;       // [3]
});
```

- **[1]**: model is validated before save
- **[2]**: alice.get('username') is called internally
- **[3]**: password field is converted to special object when fetched from database.

2.2.2 Validation

```
Model.prototype.validate()
```

Returns Promise[Checkit.Error]

Model method validate is called automatically before saving or may be called explicitly. It takes validation rules added to model by fields and passes them to Checkit.

You may override this method in your model to add custom validation logic.

2.2.3 Base class

All fields are a subclass of Field class.

class `Field` (*name*, *options = {}*)

Arguments

- **`name`** (*`String`*) – the name of the field
- **`options`** (*`Object`*) – field options

Options:

column: String use passed string as a database column name instead of field name

createProperty: Boolean, default true create accessor for this field

validation: Boolean, default true enable validation of this field value

message: String used as a default error message

label: String used as a field label when formatting error messages

validations: Array array of validation rules that Checkit can understand

2.2.4 Field classes

StringField

class `StringField` (*name*, *options = {}*)

Options:

minLength | min_length: Integer validate field value length is not lesser than minLength value

maxLength | max_length: Integer validate field value length is not greater than maxLength value

EmailField

class `EmailField` (*name*, *options = {}*)

Like a StringField with simple check that value looks like a email address.

UUIDField

class `UUIDField` (*name*, *options = {}*)

Like as StringField that should be formatted as a UUID.

EncryptedStringField

class `EncryptedStringField`(*name*, *options = {}*)

Options:

algorithm: String | Function Function: function that will take string, salt, iteration count and key length as an arguments and return Promise with encrypted value

String: algorithm name passed to crypto.pbkdf2

iterations: Integer iterations count passed to encryption function

keylen: Integer key length passed to encryption function

saltLength: Integer, default 16 salt length in bytes

saltAlgorithm: Function function used to generate salt. Should take salt length as a parameter and return a Promise with salt value

minLength | min_length: Integer validate that unencrypted field value length is not lesser than minLength value checked only when unencrypted value available

maxLength | max_length: Integer validate that unencrypted field value length is not greater than maxLength value checked only when unencrypted value available

class `EncryptedString`()
Internal class used to handle encrypted value.

EncryptedStringField value became EncryptedString when saved. It looses it's plain value. You should use method `verify(value) : Promise` to verify value against saved string.

NumberField

class `NumberField`(*name*, *options = {}*)

Options:

greaterThan | greater_than | gt: Number validates that field value is greater than option value

greaterThanEqualTo | greater_than_equal_to | gte | min: Number validates that field value is not lesser than option value

lessThan | less_than | lt: Number validates that field value is lesser than option value

lessThanEqualTo | less_than_equal_to | lte | max: Number validates that field value is not greater than option value

IntField

class `IntField`(*name*, *options = {}*)

NumberField checked to be an Integer.

Options (in addition to options from NumberField):

naturalNonZero | positive: Boolean validates that field value is positive

natural: Boolean validates that field value is positive or zero

FloatField

class `FloatField` (*name*, *options = {}*)

NumberField checked to be Float

BooleanField

class `BooleanField` (*name*, *options = {}*)

Converts value to Boolean

DateTimeField

class `DateTimeField` (*name*, *options = {}*)

Validates that value is a Date or a string than can be parsed as Date. Converts value to Date.

DateField

class `DateField` (*name*, *options = {}*)

DateTimeField with stripped Time part.

JSONField

class `JSONField` (*name*, *options = {}*)

Validates that value is object or a valid JSON string. Parses string from JSON when loaded and stringifies to JSON when formatted.

2.2.5 Advanced validation

- you may assign object instead of value to validation options:

```
minLength: {value: 10, message: '{{label}} is too short to be valid!'}
```

- you may add complete Checkit validation rules to field with validations option:

```
StringField 'username', validations: [{rule: 'minLength:5'}]
```

2.3 Relations

Note: Exported from bookshelf-schema/lib/relations

Relations are used to declare relations between models.

When applied to model it will

- create function that returns appropriate model or collection, like you normally does when define relations for bookshelf models

- create accessor prefixed by '$' symbol (may be configured)
- may prevent destroying of parent model or react by cascade destroying of related models or detaching them

2.3.1 Examples

CoffeeScript

```coffeescript
{HasMany} = require 'bookshelf-schema/lib/relations'

class Photo extends db.Model
  tableName: 'photos'

class User extends db.Model
  tableName: 'users'
  @schema [
    StringField 'username'
    HasMany Photo, onDestroy: 'cascade'                # [1]
  ]

User.forge(username: 'alice').fetch()
.then (alice) ->
  alice.load('photos')                                 # [2]
.then (alice) ->
  alice.$photos.at(0).should.be.an.instanceof Photo    # [3]
```

JavaScript

```javascript
var Relations = require('bookshelf-schema/lib/relations');
var HasMany = Relations.HasMany;

var Photo = db.Model.extend({ tableName: 'photos' });
var User = db.Model.extend({ tableName: 'users' }, {
  schema: [
    StringField('username'),
    HasMany(Photo, {onDestroy: 'cascade'})             // [1]
  ]
});

User.forge({username: 'alice'}).fetch()
.then( function(alice) {
  return alice.load('photos');                         // [2]
}).then ( function(alice) {
  alice.$photos.at(0).should.be.an.instanceof(Photo); // [3]
});
```

- **[1]** HasMany will infer relation name from the name of related model and set it to 'photos'

  When relation name is generated from model name it uses model name with lower first letter and pluralize it for multiple relations.

- **[1]** when used with *registry* plugin you may use model name instead of class. It will be resolved in a lazy manner.
- **[2]** load will work like in vanilla bookshelf thanks to auto-generated method 'photos'

- **[3]** $photos internally calls `alice.related('photos')` and returns fetched collection

2.3.2 Relation name

Actual relation name (the name of generated function) is generated from one of the following, sequently:

- name passed as an option to relation constructor
- string, passed as a relation if used with registry
- relatedModel.name or relatedModel.displayName
- camelized and singularized related table name

Additionally if name isn't passed as an option relation name is pluralized for the multiple relations and its first letter is converted to lower case.

2.3.3 Accessor helper methods

In addition to common collection or model methods accessors provides several helpers:

assign (*list*, *options = {}*)

Arguments

- **list** (*Array*) – list of related models, ids, or plain objects
- **options** (*Object*) – options passed to save methods

```
alice.$photos.assign([ ... ])
```

Assigns passed objects to relation. All related models that doesn't included to passed list will be detached. It will fetch passed ids and tries to creates new models for passed plain objects.

For singular relations such as HasOne or BelongsTo it accepts one object instead of list.

attach (*list*, *options = {}*)

```
alice.$photos.attach([ ... ])
```

Similar to assign but only attaches objects.

detach(list, options = {})()

```
alice.$photos.detach([ ... ])
```

Similar to assign but only detaches objects. Obviously it can't detach plain objects.

Note: assign, attach and detach are wrapped with transaction

2.3.4 Count

`Collection.prototype.`**count**()

Bookshelf Collection.prototype.count method is replaced and now *(finally!)* usable with relations and scoped collections. So you can do something like `alice.$photos.count().then (photosCount) -> ...`

And it still passes all the count-related tests provided by Bookshelf.

2.3.5 Base class

All relations are a subclass of Relation class.

class `Relation` (*model*, *options = {}*)

Arguments

- **`model`** (*(`Class`|`String`)*) – related model class. Could be a string if used with registry plugin.
- **`options`** (*`Object`*) – relation options

Options:

createProperty: Boolean, default true create accessors for this relation

accessorPrefix: String, default "$" used to generate name of accessor property

onDestroy: String, one of "ignore", "cascade", "reject", "detach", default "ignore" determines what to do when parend model gets destroyed

- ignore - do nothing
- cascade - destroy related models
- reject - prevent parent model destruction if there is related models
- detach - detach related models first

Note: Model.destroy is patched so it will wrap callbacks and actual model destroy with transaction

---

through: (Class|String) generate "through" relation

foreignKey, otherKey, foreignKeyTarget, otherKeyTarget, throughForeignKey, throughForeignKeyTarget: String has the same meaning as in appropriate Bookshelf relations

2.3.6 Relation classes

HasOne

class `HasOne` (*model*, *options = {}*)

BelongsTo

class `BelongsTo` (*model*, *options = {}*)

Adds IntField <name>_id to model schema

Note: if custom **foreignKey** used it may be necessary to explicitly add corresponding field to avoid validation errors

---

HasMany

class `HasMany` (*model*, *options = {}*)

MorphOne

class **MorphOne** (*model*, *polymorphicName*, *options = {}*)

Arguments

- **polymorphicName** (*String*) –

Options:

columnNames: [String, String] First is a database column for related id, second - for related type

morphValue: String, defaults to target model tablename The string value associated with this relation.

MorphMany

class **MorphMany** (*model*, *polymorphicName*, *options = {}*)

Arguments

- **polymorphicName** (*String*) –

Options:

columnNames: [String, String] First is a database column for related id, second - for related type

morphValue: String, defaults to target model tablename The string value associated with this relation.

MorphTo

class **MorphTo** (*polymorphicName*, *targets*, *options = {}*)

Arguments

- **polymorphicName** (*String*) –
- **targets** (*Array*) – list of target models

Options:

columnNames: [String, String] First is a database column for related id, second - for related type

Adds IntField <name>_id or columnNames[0] to model schema

Adds StringField <name>_type of columnNames[1] to model schema

2.4 Scopes

Note: Exported from bookshelf-schema/lib/scopes

Adds rails-like scopes to model.

2.4.1 Examples

CoffeeScript

```coffeescript
Scope = require 'bookshelf-schema/lib/scopes'

class User extends db.Model
  tableName: 'users'
  @schema [
    StringField 'username'
    BooleanField 'flag'
    Scope 'flagged', -> @where flag: true                    # [1]
    Scope 'nameStartsWith', (prefix) ->                      # [2]
      @where 'username', 'like', "#{prefix}%"
  ]

class Group extends db.Model
  tableName: 'groups'
  @schema [
    BelongsToMany User
  ]

User.flagged().fetchAll()
.then (flaggedUsers) ->
  flaggedUsers.all('flag').should.be.true

User.flagger().nameStartsWith('a').fetchAll()                # [3]
.then (users) ->
  users.all('flag').should.be.true
  users.all( (u) -> u.username[0] is 'a' ).should.be.true

Group.forge(name: 'users').fetch()
.then (group) ->
  group.$users.flagged().fetch()                             # [4]
.then (flaggedUsers) ->
  flaggedUsers.all('flag').should.be.true
```

JavaScript

```javascript
var Scope = require('bookshelf-schema/lib/scopes');

var User = db.Model.extend( { tableName: 'users' }, {
  schema: [
    StringField('username'),
    BooleanField('flag'),
    Scope('flagged', function(){                             // [1]
      this.where({ flag: true });
    }),
    Scope('nameStartsWith', function(prefix) {               // [2]
      this.where('username', 'like', prefix + '%')
    })
  ]
});

var Group = db.Model.extend( {  tableName: 'groups' }, {
  schema: [ BelongsToMany(User) ]
```

```
});

User.flagged().fetchAll()
.then( function(flaggedUsers) {
  flaggedUsers.all('flag').should.be.true;
});

User.flagged().nameStartsWith('a').fetchAll()              // [3]
.then( function(users) {
  users.all('flag').should.be.true;
  users.all(function(u){
    return u.username[0] == 'a';
  }).should.be.true;
});

Group.forge({ name: 'users' }).fetch()
.then( function(group) {
  return group.$users.flagged().fetch()                    // [4]
}).then( function(flaggedUsers) {
  flaggedUsers.all('flag').should.be.true;
});
```

- **[1]**: scope invoked in context of query builder, not model
- **[2]**: scopes are just a functions and may use an arguments
- **[3]**: scopes may be chained
- **[4]**: scopes from target model are automatically lifted to relation

2.4.2 Base class

class `Scope` (*name*, *builder*)

Arguments

- **`name`** (*`String`*) – scope name
- **`builder`** (*`Function`*) – scope function

2.4.3 Default scope

Scope with name "default" is automatically applied when model is fetched from database.

2.4.4 Unscoped

`Model.unscoped()`

`Model.prototype.unscoped()`

`Collection.unscoped()`

`Collection.prototype.unscoped()`

Model and Collection gets method *unscoped* that removes all applied scopes.

2.5 Listen

Note: Exported from bookshelf-schema/lib/listen

Declare event listener.

2.5.1 Examples

CoffeeScript

```coffeescript
Listen = require 'bookshelf-schema/lib/listen'

class User extends db.Model
  tableName: 'users'

  @schema [
    Listen 'saved', ( -> console.log "#{@username} saved" )
    Listen 'fetched', 'onFetched'
  ]

  onFetched: -> console.log "#{@username} fetched"
```

JavaScript

```javascript
var Listen = require('bookshelf-schema/lib/listen');

var User = db.Model.extend( {
    tableName: 'users',
    onFetched: function() {
      console.log this.username + ' fetched';
    }
  }, {
    schema: [
      Listen('saved', function(){ console.log( this.username + ' saved'); }),
      Listen('fetched', 'onFetched')
    ]
});
```

Callbacks are called in context of model instance. If callback is a string it should be a model method name.

2.5.2 Base class

class **Listen** (*event*, *callbacks...*)

Arguments

- **event** (*String*) – Bookshelf event
- **callback** (*(Function|String)*) – callback function or method name

2.6 Options

Note: Exported from bookshelf-schema/lib/options

Sets plugin options for specific model

2.6.1 Examples

CoffeeScript

```coffeescript
Options = require 'bookshelf-schema/lib/options'

class User extends db.Model
  tableName: 'users'
  @schema [
    Options validation: false                    # [1]
  ]
```

JavaScript

```javascript
var Options = require('bookshelf-schema/lib/options')

var User = db.Model.extend({ tableName: 'users' }, {
  schema: [ Options({ validation: false }) ]      // [1]
});
```

- **[1]** disable validation for model User

2.6.2 Class Options

class `Options` (*options*)

Arguments

- **`options`** (*Object*) – merged with plugin options and stored in model class

CHAPTER 3

Indices and tables

- genindex
- search

Index