Table of Contents

- Introduction 0
- Getting Started 1
  - Installing 1.1
  - Examples 1.2
- Quick Collection 2
  - Extend or Mixin 2.1
  - Usage 2.2
    - fill(...) 2.2.1
    - haul(...) 2.2.2
    - refill(...) 2.2.3
- Sparse Collection 3
  - Extend or Mixin 3.1
  - Configuration 3.2
  - Worker Lifecycle 3.3
  - Usage 3.4
    - filterAsync(...) 3.4.1
    - isPrepared(...) 3.4.2
    - mapAsync(...) 3.4.3
    - prepare(...) 3.4.4
    - reduceAsync(...) 3.4.5
    - resetProjection(...) 3.4.6
    - sortAsync(...) 3.4.7
  - Custom Methods 3.5
  - Advanced Usage 3.6
- Change Log 4

Backbone.Conduit

You Want More Data in Your Backbone

We hear you. Backbone is intentionally a small-scale, event-driven Model-View framework; this is one of its greatest attributes. However there are some use cases where you need larger data sets. Event-driven frameworks need special handling to perform well in these scenarios.

Backbone.Conduit is a Backbone plugin that improves the ability to handle large scale data sets.

Why Use It

As web applications grow more complex, their data requirements tend to increase as well. Scaling to larger data sizes typically falls on the server-side code to handle in the form of paging data as well as complex sorting, filtering, and mapping. However scaling on the server is not always the easier path. There are some use cases where shipping all the data to the client and letting it figure things out is a more efficient use of resources.

`Backbone.Conduit` provides two different Collections you may use:

- QuickCollection provides a Collection that accepts new models ~ 40% faster than the base `Backbone.Collection`. Use it in situations where you expect to have more than a few hundred of items, up to several thousands.
- SparseCollection provides a Collection that manages all data on a different thread. This allows for heavy data processing client-side and can scale to hundreds of thousands of items. It requires more care to load & manage the data, but is very scalable and performant.

Our Goal

The data layer of Backbone -- the `Backbone.Collection` and `Backbone.Model` -- is a simple, well-understood abstraction on top of a REST-ful server interface. `Backbone.Conduit` 's goal is to extend the existing Backbone functionality in ways that allow it to handle more data.

Any scalable solution must be careful to not perform unnecessary work. The core components of *Backbone.Conduit* have made some choices about what work is unnecessary when dealing with large data sets. While Backbone is famously non-opinionated, *Backbone.Conduit* takes an opinion about how to best scale.

Requirements

Backbone.Conduit requires:

- BackboneJS version 1.0 or greater
- All browsers that support Web Workers
- All browsers that support Native Promises. If you must support IE 10/11, you can initialize a Promise.polyfill prior to loading Backbone.Conduit.

Other Links:

- Here's a pdf of this documentation
- Check out the Demo.
- Here's the GitHub Repo
- Please [File An Issue] if you find a problem (https://github.com/pwagener/backbone.conduit/issues).
- Ask Questions either via Gitter or via Twitter

Getting Started with Backbone.Conduit

The two different Collection implementations are effective at different situations.

QuickCollection

This `QuickCollection` is useful for situations where you need better performance in initializing your Collection. It actively avoids doing some of the initialization work in a `Backbone.Model`. It provides several work-alike methods (`haul(...)`, `fill(...)`, and `refill(...)`) that perform better than the comparable `Backbone.Collection` methods. Think of it as a drop-in replacement for a `Backbone.Collection`.

SparseCollection

The `SparseCollection` fetches and stores the raw data in a Collection in a Web Worker. It provides several asynchronous methods (`sortAsync(...)`, `filterAsync(...)`, `mapAsync(...)`, and `reduceAsync(...)`) that enables data processing on large data sets in the client. It also does not create any `Backbone.Model` instances until they are explicitly requested via the `prepare(...)` method. This avoids the extremely expensive Model creation until the developer needs to attach a model to a View.

Installation

Backbone.Conduit is available in both `bower` and `npm`:

```
$ bower install --save backbone.conduit
... or ...
$ npm install --save backbone.conduit
```

Accessing

Backbone.Conduit is packed so that it is accessible in several ways. However you get access to it, this documentation simply refers to `Conduit` as the namespace.

RequireJS Structures

If you're building a webapp that asynchronously loads its dependencies, you're probably using RequireJS:

```
// In your require.config ...
require.config({
    ...
    'backbone.conduit': 'bower_components/backbone.conduit/dist/backbone.conduit.js'
});

// In your separate module files, just pull it in:
define([ 'backbone.conduit' ], function(Conduit) {
    return Conduit.QuickCollection.extend({
        // ... your behavior here
    });
});
```

CommonJS Structures

If you're leveraging it in a server-side environment directly (i.e. Node) or as part of your build process, you're probably using CommonJS. Requiring Conduit works just like any other CommonJS module:

```
var Conduit = require('backbone.conduit');
var MyCollection = Conduit.QuickCollection.extend({
    // ...
});
```

As a Global Object

Finally, if you are not using modular JavaScript, Conduit is accessible globally as `Backbone.Conduit`:

```
var MyCollection = Backbone.Conduit.QuickCollection.Extend({
    // ...
});
```

Examples

Please check out the live demo.

If you are interested in the code behind the demo, it is included in the GitHub repo, with details on running the example locally.

The data-handling pieces of the demo are mostly limited to:

- BasicCollection.js - A basic `Backbone.Collection` instance that is used as a benchmark
- ConduitCollection.js - An example of extending `Conduit.QuickCollection` for faster data handling
- SparseCollection.js - An example of extending `Conduit.SparseCollection` for scalable, asynchronous data handling

Please the the About page of the demo for details of what the code is doing.

QuickCollection: Faster Model Creation

Model creation in `Backbone` is expensive, and scales linearly. At some point above a few hundred items, the simple act of creating the Collection affects the performance of the UI. The `Conduit.QuickCollection` can avoid some of that Model-creation overhead for you. It incorporates best practices for using collections to handle large data, and is optimized to create `Backbone.Model` instances faster.

`QuickCollection` works just like any other `Backbone.Collection`, but it provides three work-alike methods that work slightly differently:

- `refill(...)` works just like `Backbone.Collection.reset(..)`
- `fill(...)` works just like `Backbone.Collection.set(...)`
- `haul(...)` works just like `Backbone.Collection.fetch(...)`

Use these methods when you are handling larger data sets. Depending on the JavaScript engine you use, you'll see some improvements in performance:

Backbone Method	Conduit.QuickCollection	Improvement
`reset(data, options)`	`refill(data, options)`	~ 47%
`set(data, options)`	`fill(data, options)`	~ 54%
`new Collection(data)`	`new Conduit.QuickCollection(data)`	~ 49%

Check out the comparison in this demo.

Some typical use cases

Initializing from Data on the Page

To bootstrap the collection with data that is already available, use the `refill(...)` method:

```
// Data placed on the page by the server
var aLargeArray = <%= @accounts.to_json %>;

var accounts = new Conduit.QuickCollection();
accounts.refill(aLargeArray);
```

Initializing from Data Fetched Asynchronously

To bootstrap the collection with data that from the server, use the `haul(...)` method, listening for the 'sync' event *or* handling the resolved Promise:

```
var AccountsCollection = Conduit.QuickCollection.extend({
    ...
});

var accounts = new AccountsCollection();
accounts.once('sync', function() {
    console.log("You've got data");
});
accounts.haul().then(function() {
    console.log("Another way of knowing you've got data");
});
```

QuickCollection Extension or Mix In

The `QuickCollection` is composed of three different *modules*. To use it, you can either extend `Backbone.Conduit.QuickCollection` *or* mix the modules into your own Collection instance.

Extending

The `QuickCollection` Backbone class includes all the functionality described below. Extend it like you would any other Backbone class:

```
var MyCollection = Backbone.Conduit.QuickCollection.extend({
    initialize: function(models, options) {
        // ...
    },

    // ... and so on
});

var collection = new MyCollection();

// If you have a large amount of data injected onto the page, instead of 'reset(...)'
var aBigArray = [ ... ];
collection.refill(aBigArray);

// Or, if you need to get it asynchronously, instead of 'fetch()' do ...
collection.haul();
```

Mixin

You may already be extending from another `Backbone.Collection` class. Or, you may not want to include all of the functionality of a `QuickCollection`. In that case, you can manually add whatever functionality you need:

```
var MyCollection = Backbone.Collection.extend({
    // ... the usual stuff ...
});

Conduit.haul.mixin(MyCollection);

var collection = new MyCollection();
collection.haul();
```

There are also mixin methods for `Conduit.fill` and `Conduit.refill`. Note that mixing in the `haul` module also will implicitly add `fill` and `refill`, as they are explicit dependencies.

One cavaet with using the mixin capability: it does not alter the *Constructor* of your Collection. Therefore, passing raw model data directly into the collection will use `Backbone.Collection.reset(...)`, which will perform the same as a regular Collection. Instead, instantiate an empty Collection and use `refill` directly:

```
var MyCollection = Backbone.Collection.extend({
    // ... the usual stuff ...
});
Conduit.haul.mixin(MyCollection);

// This won't have any optimizations
var mySlowCollection = new MyCollection(rawModelData);

// ... but this will
var myFastCollection = new MyCollection();
myFastCollection.refill(rawModelData);
```

Using QuickCollection

Apart from loading data, `QuickCollection` should be used exactly the same way as a regular `Backbone.Collection`. Loading data via the `refill(...)`, `fill(...)`, or `haul(...)` methods work very similarly to their Backbone counterparts. However, each method does explicitly do less work than the regular Backbone method. The tradeoffs chosen by Conduit (i.e. no firing of individual "add" events) are easy to work with. See the API of each individual method for details of the differences.

NOTE: You can still use the regular `reset(...)`, `set(...)` and `fetch(...)` methods of a `QuickCollection` instance, but they will not use the optimized Model creation code.

Backbone.Conduit.QuickCollection.fill(...)

The `Backbone.Conduit.QuickCollection` *fill(...)* method provides an alternative to *set(...)*, allowing you to add data to a Collection. It supports the exact same options as specified by Backbone.Collection.set.

Like the refill module, it is provided by default in Conduit.QuickCollection:

```
var collection = new Conduit.QuickCollection();
collection.fill(someLargeArray, options);
```

You can also mix the functionality into your own Collection subclass:

```
var MyCollection = Backbone.Collection.extend({ ... });
Conduit.fill.mixin(MyCollection);
collection.fill(someLargeArray, options);
```

Differences from Backbone.Collection.set(...)

Conduit.fill's behavior differs in some significant ways:

- *fill(...)* does not trigger individual *add*, *remove*, or *change* events. Instead, a single "fill" event will be triggered after all elements have been added.
- No data validation on Model instance creation
- No tracking of previous attributes (i.e. "undefined") within the Model instances

Also, note the following may reduce the effectiveness of *Conduit.fill*'s optimizations:

- If the Model being used by the Collection provides a `defaults` hash, the performance improvements will be reduced
- If you've overwritten the Backbone.Model Constructor, the performance improvements will be greatly reduced (overriding initialize(...) is, of course, just fine)

Backbone.Conduit.QuickCollection.haul(...)

The `Backbone.Conduit.QuickCollection` *haul(...)* method provides an alternative to *fetch(...)* that utilizes either `fill(...)` or `refill(...)` to add the returned data into the Collection. It supports the same options specified by Backbone.Collection.fetch. It returns a *Promise* that resolves when the data has been successfully received from the server and added to the collection.

It is meant as a special-purpose replacement for `fetch(...)` for when you must load a large number of items from the server. `haul(...)` is used exactly like `fetch(...)`:

```
var MyCollection = Conduit.QuickCollection.extend({
    // ... your own Collection behaviors
});
var accounts = new MyCollection();

// If you want to use events, listen to 'sync'
accounts.once('sync', function() {
    // ... do something with the full set of accounts
});

// If you want to use Promises, chain away
accounts.haul().then(function() {
    // ... do something with the full set of accounts
});
```

If you explicitly want to use `refill(...)` (the analogous method to `Backbone.Collection.reset(...)`), then pass in the `reset` option:

```
accounts.haul({ reset: true });
```

In either case, the collection will trigger a "sync" event when it has been synchronized with the server, just like the behavior of `fetch(...)`.

Backbone.Conduit.QuickCollection.refill(...)

The `Backbone.Conduit.QuickCollection` *refill(...)* method provides an alternative to *reset(...)* that is optimized for faster `Backbone.Model` creation. It supports the same options as Backbone.Collection.reset.

Like [the fill module], it is provided by default in `Conduit.QuickCollection`:

```
var MyCollection = Conduit.QuickCollection.extend({
    // ... your own Collection behaviors
});

var accounts = new MyCollection();
accounts.refill(<%= @accounts.to_json %>);
```

Alternatively, you may mix the method into a Collection of your own:

```
var MyCollection = Backbone.Collection.extend({
    // ... your own Collection behaviors
});
Conduit.refill.mixin(MyCollection);

var accounts = new MyCollection();
accounts.refill(<%= @accounts.to_json %>);
```

Conduit's `refill(...)` method fires a "reset" event in the same manner of the `reset(...)` method, so it can be used as a drop-in replacement.

Differences from Backbone.Collection.reset(...)

Performance of `refill(...)` is ~ 45% better than `reset(...)` in most use cases. The behavior does differ in some significant ways:

- No data validation on Model instance creation
- No tracking of previous attributes (i.e. "undefined") within the Model instances

Also, note the following may reduce the effectiveness of *Conduit.refill*'s optimizations:

- If the Model being used by the Collection provides a `defaults` hash, the

performance improvements will be reduced

- If you've overwritten the Backbone.Model Constructor, the performance improvements will be greatly reduced (overriding initialize(...) is, of course, just fine)

SparseCollection

As noted elsewhere, Model creation in `Backbone` is very expensive. For large data sets (tens or hundreds of thousands of items), even relatively simple data structures will cause a Backbone-based web application to hang. Additionally, any non-trivial amount of data organization (joining, filtering, sorting) over a very large number of `Backbone.Model` instances does not scale well, as those operations are synchronous.

The `Conduit.SparseCollection` addresses both of these problems. It fundamentally changes how a `Collection` operates to storage and management data in a dedicated *Web Worker* thread. The data stored directly in the `Collection` is sparse -- only models that have been explicitly requested are created and available there. This leaves the main Javascript thread free to do what it should always be doing: interacting with the user.

You can see how effective this is, even with large data sets, in the demo app.

Please be aware this implementation has some limitations; see the Usage section for details.

SparseCollection Extension or Mixin

The `SparseCollection` can be used directly, extended, or mixed into another `Collection` constructor.

Extending

The `SparseCollection` Backbone class includes all functionality described below. Use it or extend it like you would any other Backbone class:

```
Backbone.Conduit.SparseCollection.extend({
    initialize: function(models, options) {
        // ...
    },

    // ... and so on
});

var collection = new MyCollection();

// If you have a large amount of data injected onto the page, instead of 'reset(...)'
var aBigArray = [ ... ];
collection.refill(aBigArray);

// Or, if you need to get it asynchronously, instead of 'fetch()' do ...
collection.haul();
```

Mixin

If you are already extending from a `Backbone.Collection` class, you may mix in the `sparseData` module's behavior to act like a `SparseCollection`:

```
var MyCollection = Backbone.Collection.extend({ ... });
Conduit.sparseData.mixin(MyCollection);

// Let's get some data
var collection = new MyCollection();
collection.haul().then(function() {
    console.log('We now have ' + collection.length + ' items!');
});
```

The `sparseData` module will also include the `haul`, `fill` and `refill` modules from the `Conduit.QuickCollection`; those methods act as replacements for the corresponding `fetch`, `set`, and `reset` methods of a `Backbone.Collection`.

Note that mixing in functionality to an existing Collection class may be problematic. The internal behavior of a `SparseCollection` is dramatically different than other collections. See the Usage section for more details.

SparseCollection Configuration

Utilizing a web worker for data management requires some configuration. Specifically, you must provide the path to where `Backbone.Conduit` is installed. Use the `Backbone.Conduit.config` object to enable Web Worker support:

```
// Enable the Conduit worker
Conduit.config.enableWorker({
    paths: '/your/path/to/backbone.conduit',

    // Optional arguments:
    debug: true,
    workerDebug: true
}).then(function() {
    console.log('Worker is now enabled!');
});
```

The `paths` argument is required, and can be either a string or an array of strings specifying paths to look for the worker in. Note the path is to the directory, *not* the file.

Other configuration options you can choose to provide:

- debug - If true, JS console logs will include some output about loading & configuring the worker.
- workerDebug - If true, JS console logs will include output for each method executed on the worker.

This call returns a Promise that resolves when the worker has been successfully loaded.

SparseCollection Web Worker Lifecycle

The `SparseCollection` will create a worker thread on demand when necessary. After it is created, the worker will live *until it is manually terminated*. Additionally, any instance of a SparseCollection will create its own worker; there is a significant risk of leaking threads if you plan on using multiple `SparseCollection`s.

Manually Terminating the Web Worker

Unless your `SparseCollection` is going to survive for the life of your web-based application, you should manually terminate it. Since the canonical data is stored on the worker itself, this should only be done after you have prepared all the `Model` instances you need in the UI thread (see prepare(...)). Use the 'collection.stopWorkerNow()' method to terminate the worker, which does so synchronously.

One safe technique would be to register an `onunload` event that does exactly that:

```
var collection = new MySparseCollection();
window.onunload = function() {
    collection.stopWorkerNow();
    // The worker is now stopped.
);
```

jQuery provides a method for registering multiple handlers as well.

Manually Starting the Web Worker

The worker will be created on demand, but doing so takes a bit of time to load the necessary JS from the server and initialize things. If you want to proactively create the worker for your collection, use `collection.createWorkerNow()` method. This returns a `Promise` that resolves when the worker has been create.

```
var collection = new MySparseCollection();
collection.createWorkerNow().then(function() {
    // The worker has now been created
});
```

Using SparseCollection

A `SparseCollection` makes an explicit tradeoff for developers: it uses asynchronous behavior in order to provide better performance and scalability. The canonical copy of the data lives in a fully separate thread, requiring the developer to be much more methodical in the loading, parsing, organizing, and accessing of that data. This also means that most of the synchronous data-related methods in a `Backbone.Collection` (for instance, `sort()`, `find()`, and others) will intentionally throw an error. This class provides asynchronous replacements, however, enabling much more powerful data handling in the client.

The typical flow of code for a `SparseCollection` will look something like this:

```
// Create the Sparse Collection
var collection = new MySparseCollection();

// First fetch the data
collection.haul()
    .then(function() {
        // Next organize the data
        return collection.sortAsync();
    }).then(function() {
        // Next prepare a few models
        return collection.prepare({
            indexes: { min: 0, max: 10 }
        });
    }).then(function(models) {
        // Use the models
        console.log('You've got models!');
    });
```

Loading Data Into the Collection

This module includes the haul module from the `QuickCollection`, and builds on the performance improvements implemented there. To that end, data should be loaded the same way: use the `haul()` method (a replacement for `fetch()`) for loading data via an XHR, or using `fill()` / `refill()` (replacements for `set()` / `reset()` to load data into a Collection directly. Typical usage will look similar to code using `Conduit.QuickCollection`:

```
var MyCollection = Backbone.Collection.extend({ ... });
Conduit.sparseData.mixin(MyCollection);

// Let's get some data
var collection = new MyCollection();
collection.haul().then(function() {
    console.log('We now have ' + collection.length + ' items!');
});
```

Since `haul()` returns a promise, you are guaranteed the data has been stored on the worker when it resolves.

Parsing Loaded Data

Conduit expects the data provided to the worker will be an Array -- not an Object. However, to minimize the size of the JSON file, many API's deliver data packaged inside of another object. For instance, the server may return JSON that looks like:

```
{
    meta: {
        // ... data about the data
    },
    data: [
        // ... the data itself
    ]
}
```

A typical `Backbone.Collection` will override Backbone.Collection.parse(...) to transform this data into the appropriate array. However, that is not feasible or desirable with a very large data set; doing this work on the main UI thread would lead to poor performance.

Instead, you may transform the raw data as a part of the `haul()` operation with the `postFetchTransform` option. You can specify the transformation in two ways: First, if you only want to extract the data from a larger object, specify the property on the object that we should use as the actual data. For instance:

```
var collection = new MyCollection();
collection.haul({
    postFetchTransform: {
        useAsData: 'data'
    }
}).then(function() {
    console.log('The "data" attribute was used as the collection of items');
});
```

If you need to do a more complex transformation, you can provide the name of the method to call that implements the transformation:

```
// Let's get some data ... and transform it
var collection = new MyCollection();
collection.haul({
    postFetchTransform: {
        method: 'extractFromRawData`,
        context: { userName: 'pwagener' }
    }
}).then(function(finalTransformContext) {
    console.log('The raw data has been transformed by my own method');
});
```

When you use a `method` to transform the data, the returned promise will resolve to the final state of the context of the transforming function (named `finalTransformContext` here). This provides a lot of flexibility, including allowing you to extract meta data from the JSON response and keep it on the main UI thread. You can also provide the initial context to the transforming method by providing a `context` key to `postFetchTransform`. The example above provides the `userName`, which can then be used in the transforming method.

Please Note: the implementation of the transforming method (`extractFromRawData` in this example) must be provided separately to the ConduitWorker. See the Custom Methods section for more details on registering Conduit components. For this example, if you wanted to remove a field from the data that will be exposed in the collection, you would do something like:

```
ConduitWorker.registerComponent({
    name: 'sampleComponent',
    methods: [
        {
            name: 'extractFromRawData'
            method: function(rawData) {
                var userName = this.userName;
                return _.map(rawData.data, function(item) {
                    // Add the name from the main thread
                    var result = _.extend({}, { name: userName }, item);

                    // Don't include 'password' in the data
                    return _.omit(item, 'password');
                });
            }
        }
    ]
});
```

That implementation should expect to receive the raw data from the requested URL, and must return an array of javascript objects that will represent the items in the collection. Note it utilizes the context provided that includes the `userName` key from the main UI thread, shown as `this.userName` above.

Data Projections

Since the full copy of the data is managed on the worker thread, most synchronous `Backbone.Collection` method calls on a `sparseData`-enabled collection will throw an error. Instead, `Conduit` provides alternative, asynchronous methods that return promises.

The `sortAsync()`, `filterAsync()`, and `mapAsync()` methods can be thought of as a projection onto the original data. When each projection is applied, the newly projected data becomes available. Projections can build on top of each other, so you can first filter data and then sort it.

When using a `method` to implement the projection, they accept a `context` to execute the method in; the final state of the `context` is provided when the method's returned `Promise` resolves. Since it came from the Worker thread however, you cannot pass functions through via `context`.

The data (projected & otherwise) continues to live on the worker thread. To return to the original, unprojected data, call `resetProjection()`. For instance:

```
collection.filterAsync(
    // Apply some filter
).then(function() {
    return collection.resetProjection();
}).then(function() {
    // The data is now back to its un-filtered state
});
```

Finally, all data projection methods emit their own events (i.e. `sortAsync`, `filterAsync`, `mapAsync`) upon completion to differentiate themselves from the comparable `Backbone.Collection` synchronous methods.

Limitations

This module has some limitations. The most notable limitation is any collection leveraging `sparseData` should be considered **read-only**. The models returned from `prepare(...)` are perfectly functional, so feel free to update those. But bear in mind changes to those models will not automatically propagate to the data on the worker thread.

If needed, you may propagate the data back to the worker yourself via `fill(...)`. Further version of Backbone.Conduit may introduce more functionality related to making them fully writeable and automatically synchronizing the data.

If you have feedback on use cases that are important to you, we'd love to hear it. Please file an issue and help make `Backbone.Conduit` a great way to deal with large data sets.

SparseCollection.filterAsync(...)

Filter the data in a given manner. The method returns a `Promise` that resolves when the filtering in completed.

This can be used as as a "filter by property match", similar to Underscore's _.where(...) method, by providing the `where` option:

```
// Filtering by matching property values:
collection.filterAsync({
    where: {
        name: 'Foo'
    }
}).then(function() {
    console.log('Filtered data has a length of ' + collection.length);
});
```

Alternatively, you can apply a "filter by an evaluation function", similar to Underscore's _.filter(...) method, by providing the `method` option. You will need to provide the Conduit Worker the function implementation separately. See Custom Methods for details.

Once that is done, you can apply it by:

```
// Filter by calling an evaluation function
collection.filterAsync({
    method: 'ageGreaterThan21'
}).then(function(resultingContext) {
    console.log('There are ' + collection.length + ' items older than 21');
});
```

When you are using the `method` option to specify the filtering, the returned `Promise` will resolve to the final context of the filtering function. See the Data Projections Section of SparseCollection Usage for details.

If you prefer to specify the filter directly on the collection, you can declare it on the collection directly, similar to a regular `Backbone.Collection` comparator. For instance:

```
var MyCollection = Conduit.SparseCollection.extend({
    filterSpec: {
        method: 'ageGreaterThan21'
    },
    // ...
};

var collection = new MyCollection();
collection.haul().then(function() {
    return collection.filterAsync();
}).then(function(resultingContext) {
    console.log('The "ageGreaterThan21" filter has now been applied');
});
```

This applies a projection on the underlying data set, which can be removed by calling `resetProjection()`. Note that to filter using an evaluation function, you must provide the function separately. See Custom Methods for details.

When `SparseCollection.filterAsync()` completes, it fires the `filterAsync` event prior to resolving its Promise.

SparseCollection.isPrepared(...)

Determine if a given set of models is available in the main thread. Returns `true` if all the models are available, or `false` otherwise. Use it to determine if the models you need have already been prepared.

The method accepts the same aruments as prepare(...), allowing you to check specific IDs *or* specific indexes

```
// Check the first 100 (note 'max' is exclusive)
var prepared = collection.isPrepared({
    indexes: { min: 0, max: 100 }
});

// Check one specific index
var prepared = collection.isPrepared({
    index: 11
});

// Check specific IDs
var prepared = collection.isPrepared({
    ids: [ 1, 2, 3, 4 ]
});

// Check one specific ID
var prepared = collection.isPrepared({
    id: 5
});
```

If `collection.isPrepared(...)` returns `true`, then using the `collection.get(id)` and `collection.at(index)` methods will work as expected. If it returns `false`, those methods will throw `Exceptions` for the referenced IDs or indexes.

SparseCollection.mapAsync(...)

Map the data on the worker in a given manner. The method returns a promise that resolves when the mapping is complete. This is conceptually the same as Underscore's _.map(...) function. Note that to map the data you must provide the mapping function separately. See Custom Methods for details.

```
// Map the data
collection.mapAsync({
    method: 'translateToGerman'
}).then(function(resultingContext) {
    console.log('The data has now been mapped.');
});
```

This applies a Projection to your data set. Also, note the resulting context of the mapping function will be provided by the resolved `Promise`. See the Data Projections Section of SparseCollection Usage for details.

If you would like, you can provide the mapping function directly on the `SparseCollection` sub-class as `mapSpec`:

```
var MyCollection = Conduit.SparseCollection.extend({
    mapSpec: {
        method: 'translateToGerman'
    }

    // ...
)};

var collection = new MyCollection();
collection.haul().then(function() {
    return collection.mapAsync();
}).then(function(resultingContext) {
    console.log('The data has now been mapped by "translateToGerman" function');
});
```

This applies a projection on the underlying data set, which can be removed by calling `resetProjection()`.

When `SparseCollection.mapAsync()` completes, it fires the `mapAsync` event prior to resolving its Promise.

SparseCollection.prepare(...)

Create specific models in the main thread.

Since model creation is expensive, a `SparseCollection` only holds models that have been explicitly requested. The rest of the data in a collection is stored in raw form in a Web Worker thread. To use models in the main thread, they must first be prepared by calling `collection.prepare(...)`.

`prepare(...)` returns a `Promise` that resolves when the models have been prepared in the main thread. The `Promise` resolves to the model or the set of models that were prepared. Additionally, after the `Promise` resolves, you can use the `collection.get(id)` or `collection.at(index)` to reference the prepared models.

This method allows you to specify the models to prepare by ID or by index, and allows you to prepare them individually or in groups:

```
// Prepare a the first 100 indexes (note 'max' is exclusive)
collection.prepare({
    indexes: { min: 0, max: 100 }
}).then(function(models) { ... });

// Prepare one specific index
collection.prepare({
    index: 11
}).then(function(model) { ... });

// Prepare specific IDs
collection.prepare({
    ids: [ 1, 2, 3, 4 ]
}).then(function(models) { ... });

// Prepare one specific ID
collection.prepare({
    id: 5
}).then(function(model) { ... });
```

Note this method accepts the same arguments as isPrepared(...), which can determine if models have already been prepared or not.

SparseCollection.reduceAsync(...)

Reduce the data in the array on the worker down to a single value. This is conceptually the same operation as Underscore's _.reduce(...) function. Just like the Underscore version, the `method` you provide is passed four values: `memo`, `value`, `index`, and finally a reference to the full `list` of data.

You must provide the Conduit Worker the reduction function separately. See Custom Methods for details. Once that is done, you reduce your data providing the `method` and `memo` arguments:

```
// Reduce the data
collection.reduceAsync({
    method: 'calculateAverage',
    memo: 0
}).then(function(average) {
    console.log('The average is: ' + average);
});
```

Note this method's returned promise resolves to the final reduction value. It is not a projection, so it does not modify the underlying data set stored on the worker.

Finally, you may provide a `context` option to `reduceAsync(...)`. The value provided there will be accessible as `this` in your reduction method.

```
collection.reduceAsync({
    method: 'summarize',
    context: {
        definitions: {
            ...
        }
    }
}).then(function(summary) {
    // ...
});
```

SparseCollection.resetProjection()

Remove any projections on the original data that have been applied from calling `sortAsync()`, `filterAsync()`, or `mapAsync()`, returning the data to its original state.

Any projection is applied on top of previous projections. This method removes all projections from the data, allowing you to reorganize your original data. It returns a `Promise` that resolves when all projections have been removed:

```
collection.sortAsync()
    .then(function() {
        return collection.mapAsync({ ... });
    });

// ... some time later ...

collection.resetProjection()
    .then(function() {
        console.log('Data returned to the unsorted, un-mapped state!');
    });
```

SparseCollection.sortAsync(...)

Sort the data on the worker thread. Method takes a single argument describing the sort operation, which indicates how to sort the data.

You may choose to sort by an individual attribute in your data set by providing the `property` option. When doing so, you can optionally include the `direction` to specify the direction of the sort as `'asc'` (default) or `'desc'`.

```
collection.sortAsync({
   property: 'age`
   direction: 'desc'
}).then(function() {
    // The data on the worker is now sorted by "age"
});
```

Alternatively, you may provide an evaluation function to specify your sorting. You must provide the Conduit Worker the sorting function separately. See Custom Methods for details. Once that is done, you can provide the name of the method as the `method` option:

```
collection.sortAsync({
    method: 'yourSortMethod'
}).then(function(resultingContext) {
    // The data is now sorted using 'yourSortMethod' as the evaluator
});
```

This applies a Projection to your data set. Also, note the resulting context of the sorting function will be provided by the resolved `Promise`. See the Data Projections Section of SparseCollection Usage for details.

When `SparseCollection.sortAsync()` completes, it fires the `sortAsync` event prior to resolving its Promise.

Custom Methods

Using a worker to handle data manipulation scales very well. But using a worker mean that passing a function as a part of the manipulation (i.e. when sorting a `Collection`) is more work. `Backbone.Conduit` allows you to provide extra Javascript files to load when enabling the worker.

Registering Components

If you have functionality that is necessary for all your application's sparse collections, specify it as a part of the core Conduit configuration:

```
Conduit.config.enableWorker({
    paths: '/lib',
    components: [
        '/basicMethods.js'
    ]
});
```

Any components registered as a part of the `Conduit.config.enableWorker(...)` call will be included in all `Backbone.Conduit.SparseCollection` instances.

However, it is much more common for each sparse collection to have its own necessary functionality. In that situation, specify your component files as a part of the Collection. For instance:

```
var MyCollection = Conduit.SparseCollection.extend({
    // ...
    conduit: {
        components: [
            '/sorters.js'
        ],
    },
    //...
});
```

Here, the `sorters.js` file provides methods that can be referenced from the above data manipulation/projection methods. That file can contain anything necessary to provide the sorting functionality.

Implementing Components

ConduitWorker components are specified by naming a method something unique, then providing the method implementation. For instance, suppose your application needed a sorting function that sorted things in a case-insensitive fashion, and it also needed to only include items whose `age` was greater than 21. Your component can define these methods and then register them by calling `ConduitWorker.registerComponent` like:

```
var byNameCaseInsensitive = {
    name: 'byNameCaseInsensitive',
    method: function(item) {
        return item.name.toLowerCase();
    }
};
var ageGreaterThan21 = {
    name: 'ageGreaterThan21',
    method: function(item) {
        return item.age > 21;
    }
}

ConduitWorker.registerComponent({
    name: 'sorters',

    methods: [
        byNameCaseInsensitive,
        ageGreaterThan21
    ]
});
```

This separate `sorters.js` file will be loaded by the ConduitWorker at the appropriate time. To then reference the method when performing a sort, you provide an object as the `comparator` that names the method. I.e:

```
collection.filterAsync({
    method: { method: 'ageGreaterThan21' }
}).then(function() {
    return collection.sortAsync({
        method: { method: 'byNameCaseInsensitive' }
    });
}).then(function() {
    // The data is now filtered via 'ageGreaterThan21' and sorted via the
    // 'byNameCaseInsensitive' worker method
});
```

Advanced Usage

Sharing Workers

As of version 1.1, the *SparseCollection* provides the ability to customize how the worker is managed via the *WrappedWorker* configuration option. The WrappedWorker basically will allow only one message to be sent until a response is received, and will put any other outgoing messages into a queue. The interface is independent of the rest of Conduit's sparseData modules, which means you can extend it more easily and use your own by setting it as value for *WrappedWorker* when you invoke *enableWorker* in the Conduit config.

An included subclass of the *WrappedWorker* is *WrappedWorker.SharedWorker*. This will share the same actual worker "under the hood", so multiple sparse collections can share the same worker. The benefit of this is that you would be able to instantiate multiple *SparseCollection* instances without worrying as much about high memory usage (since each worker has its own JS VM instance).

You can use it like this:

```
Conduit.config.enableWorker({
    paths: '/js/path/to/conduit/dist/',
    WrappedWorker: Conduit.WrappedWorker.SharedWorker,
});
```

Caching Data

Related to sharing workers, version 1.1 and beyond include support for caching GET requests that are executed by the worker. f you combined the *WrappedWorker.SharedWorker* and the *useCache* option of the *restGet* or *haul* method calls, you can have multiple collections use the same raw data from *restGet* call, while each is still able to apply projections on top.

This makes a lot more sense in an example:

```
// Using the WrappedWorker.SharedWorker instance so that multiple
// SparseCollection data collections live on the same Web Worker ...

// Haul data for the first sparse collection
sparseCollectionA.haul({
   url: '/api/foo',
   useCache: true // will cause the response to be cached with the url as key
}).then(function () {
   return sparseCollectionB.haul({
       url: '/api/foo'
       // useCache:true below will cause the haul method to use the existing
       // cached data for the url on the worker.  '/api/foo' will only be
       // fetched from the server once
       useCache: true
   });
});
```

Change Log

If you have problems, please file an issue.

- *1.1.X* - Implemented worker thread pooling & data caching. (see here)
- *1.0.X* - Moved `SparseCollection` out of Experimental stage. Leveraged native Promises.
- *0.6.X* - Added `sparseData` and `SparseCollection` experimental module. Renamed `Conduit.QuickCollection` to `Conduit.QuickCollection` for clarity.
- *0.5.X* - Removed `sortAsync` experimental module; removed `fetchJumbo` module, which was replaced by `haul` ;
- *0.4.X* - Renamed `Conduit.fetchJumbo` to the less-awkward `Conduit.haul` ; provided experimental `Conduit.sortAsync` module.
- *0.3.X* - Provided `Conduit.fetchJumbo` Module as an alternative to `Backbone.Collection.fetch(...)`
- *0.2.X* - Provided `Conduit.fill` Module as an alternative to `Backbone.Collection.set(...)` .
- *0.1.X* - First Release. Provided `Conduit.refill` Module as an alternative to `Backbone.Collection.reset(...)` .

Contributors

- Peter Wagener Original Implementation, Maintainer
- James Ballantine v1.1 Improvements, Feedback