Chapter 12. Changes Feed

12.1. Introduction

The Changes Feed pattern provides a lightweight way of keeping your data in sync with the server by applying differences as opposed to reloading data.

12.2. The Problem

The user of a web application is not always the only person (or thing) modifying the underlying data. For example, in our calendar application, multiple users could be modifying appointments on the calendar at the same time. An obvious solution to this issue would be periodically running `fetch` on our collections to make sure they are up-to-date. However, this solution has a few problems because a full fetch may:

1. Be a slow operation on the server
2. Re-render a large number of views
3. Incur a large amount of client-side processing
4. Disrupt the browser with a large number of changes

A much better solution would be a changes feed that will only send what has changed since the last time we checked.

12.3. Changes feed on a Collection

Since Collections are responsible for adding, changing, and destroying Models, they are an ideal place for a changes feed. Let’s look at how we would implement a changes feed for our calendar:

```
Collections.AppointmentChanges = Backbone.Collection.extend({
  model: Models.Appointment,
  url: "/appointments",
  initialize: function(models, options) {
    _.bindAll(this, 'changes', 'processChanges', 'processChange', 'since');
    this.collection = options.collection;
    this.bind('reset', this.processChanges);
    setInterval(this.changes, 15*1000);
  },

  since: function() {
    return this.collection.max(function(appointment) {
      return appointment.get('updated_at');
    });
  },

  changes: function() {
    this.fetch({ data: { since: this.since() } });
  },

  processChanges: function() {
    this.each(this.processChange);
  },

  processChange: function(appointment) {
    var existing = this.collection.get(appointment.id);
    if (existing) {
      if (appointment.get('deleted')) {
        this.collection.remove(existing);
      } else {
        existing.set(appointment.attributes);
      }
    } else {
      this.collection.add(appointment)
    }
  }
});
```

First of all, we're setting the URL to the same as the url for `Calendar.Appointments`, but when we fetch changes we're passing a since parameter:

```
Collections.AppointmentChanges = Backbone.Collection.extend({
  model: Models.Appointment,
  url: "/appointments",
  changes: function() {
    this.fetch({ data: { since: this.since() } });
  },
  // ...
})
```

The URL in this collection is just a normal `index` call on the `/appointments` route. We're passing the since parameter to fetch so that the server can send change objects instead of the full index. Our implementation here depends on a change object looking exactly like a normal object in the case of an addition or update, and with `deleted: true` in case of a deletion.

The `since` method simply gets the maximum `updated_at` timestamp from all of our appointments and sends that to the server.

```
  since: function() {
    return this.collection.max(function(appointment) {
      return appointment.get('updated_at');
    });
  }
```

If we have no appointments, `since` will be `''`. In that case, the server will treat it like a normal index call.

We also setup a 15 second interval to call the changes method. Thus, we are continuously polling the server for changes:

```
  initialize: function(models, options) {
    _.bindAll(this, 'changes', 'processChanges', 'processChange', 'since');
    this.collection = options.collection;
    this.bind('reset', this.processChanges);
    setInterval(this.changes, 15*1000);
  },
```

If the polling call emits a `reset` event, we run the `processChanges` method.

At this point, our changes collection is populated with a bunch of `Appointment` objects. These appointments represent `Appointments` that have changed since the timestamp. The `processChanges` method is simply an iterator that calls `processChange` on each `Appointment`.

```
  processChange: function(appointment) {
    var existing = this.collection.get(appointment.id);
    if (existing) {
      if (appointment.get('deleted')) {
        this.collection.remove(existing);
      } else {
        existing.set(appointment.attributes);
      }
    } else {
      this.collection.add(appointment)
    }
  }
```

`processChange` performs the following actions:

1. Search the main collection to see if we already have the `Appointment` in our system
2. If we have the `Appointment`, check to see if it has been deleted (we would implement this server side with a `deleted` boolean), and if it has, remove if from our main collection
3. If it has not been deleted, `set` its attributes to the new ones from the server, because some attribute change has occurred
4. If we do not have an existing `Appointment`, it means it is new and needs to be added to our collection

The great thing about this changes feed is that it is so simple. All we have to do is propagate the change to a corresponding `add`, `update`, or `destroy` call on the model or collection. None of our code needs to know that a changes feed even exists! Also, `AppointmentChanges` only needs to know about the collection it's instantiated with—nothing about views, the router or anything else!

12.4. Conclusion

As in Chapter 11, *Non-REST Models*, sometimes we need to step outside of Backbone's RESTful roots to extend our application. Case in point is the Changes Feed, in which a Backbone collection is not representing data directly. Rather it is a kind of metadata: a collection of change objects. Collections and models do not have to correspond directly with a database table on the server. On the contrary, some of the most powerful and intriguing uses of collections and models are driven by metadata and interact with first-class data objects as a result.