Chapter 10. Changes Feed

10.1. 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.

10.2. 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:

```
Calendar.AppointmentChanges = new (Calendar.Appointments.extend({
  initialize: function(options) {
    _.bindAll(this, 'changes', 'processChanges', 'processChange');
    this.bind('refresh', 'processChanges');
    setInterval(this.changes, 15*1000);
  },

  since: function() {
    return Calendar.Appointments.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 = Calendar.Appointments.get(appointment.id);
    if (existing) {
      if (appointment.get('deleted')) {
        Calendar.Appointments.remove(existing);
      } else {
        existing.set(appointment.attributes);
      }
    } else {
      Calendar.Appointments.add(appointment)
    }
  }
}))
```

First of all, we're extending `Calendar.Appointments`, which is a singleton Collection containing our appointments.

```
Calendar.AppointmentChanges = new (Calendar.Appointments.extend({
  // ...
}));
```

This means that we inherit the URL from `Calendar.Appointments`. The URL in that collection is just a normal `index` call on the `/appointments` route. In our singleton, however, we use the `since` method to call `fetch` with a `since=` parameter. This instructs the server to only return appointments since that point in time.

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

```
  since: function() {
    return Calendar.Appointments.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:

```
Calendar.AppointmentChanges = new (Calendar.Appointments.extend({
  initialize: function(options) {
    _.bindAll(this, 'changes', 'processChanges', 'processChange');
    this.bind('refresh', 'processChanges');
    setInterval(this.changes, 15*1000);
  },
  // ...
}));
```

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

At this point, out changes collection is populated with a bunch of `Appointment` objects (because we inherited from `Calendar.Appointments`). 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 = Calendar.Appointments.get(appointment.id);
    if (existing) {
      if (appointment.get('deleted')) {
        Calendar.Appointments.remove(existing);
      } else {
        existing.set(appointment.attributes);
      }
    } else {
      Calendar.Appointments.add(appointment)
    }
  }
```

`processChange` performs the following actions:

1. Search our `Appointments` 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 \`Calendar.Appointments\`—nothing about views, the router or anything else!