Meteor Tracker

Source code of released version | Source code of development version ***

Meteor Tracker is an incredibly tiny (~1k) but incredibly powerful library for **transparent reactive programming** in JavaScript.

Overview

Tracker gives you much of the power of a full-blown Functional Reactive Programming (FRP) system without requiring you to rewrite your program as a FRP data flow graph. Combined with Tracker-aware libraries, this lets you build complex event-driven programs without writing a lot of boilerplate event-handling code.

Tracker is essentially a simple *convention*, or interface, that lets reactive data sources (like your database) talk to reactive data consumers (such as a live-updating HTML templating library) without the application code in between having to be involved. Since the convention is very simple, it is quick and easy for library authors to make their libraries Tracker-aware, so that they can participate in Tracker reactivity.

This README has a short introduction to Tracker. For a complete guide to Tracker, consult the thorough and informative Tracker Manual, which is five times longer than the Tracker source code itself. You can also browse the API reference on the main Meteor docs page.

Example

Take this ordinary JavaScript function:

```
var currentTemperatureFahrenheit = function () {
  return currentTemperatureCelsius() * 9/5 + 32;
};
```

We can call it for its value (assuming there's a `currentTemperatureCelsius` function):

```
> currentTemperatureFahrenheit()
71.8
```

But, if the `currentTemperatureCelsius` function is Tracker-aware (or even if it's not, but as long it reads the current temperature ultimately from some Tracker-aware data source), then we can also call `currentTemperatureFahrenheit` *reactively*:

```
> var handle = Tracker.autorun(function () {
  console.log("The current temperature is", currentTemperatureFahrenheit(),
              "F");
  });
The current temperature is 71.8 F       (printed immediately)
The current temperature is 71.9 F       (printed a few minutes later)
> handle.stop();                        (stop temperature changes from printing)
```

The function passed to `Tracker.autorun` is called once immediately, and then it's called again whenever there are any changes to any of the *reactive data sources* that it referenced. To make this work, `currentTemperatureCelsius` just needs to register with Tracker as a reactive data source when it's called, which takes only a few lines of code.

Or, instead of calling `Tracker.autorun` ourselves, we might use `currentTemperatureFahrenheit` in a Blaze template:

```
<!-- In demo.html -->
<template name="demo">
  The current temperature is {{currentTemp}} degrees Fahrenheit.
  {{#if belowFreezing}}
    That's below freezing!
  {{/if}}
</templates>
```

```
// In demo.js
Template.demo.helpers({
  currentTemp: function () {
    return currentTemperatureFahrenheit();
  },
  belowFreezing: function () {
    return currentTemperatureFahrenheit() < 32.0;
  }
});
```

When this template is shown, the temperature shown on the screen will update live as the weather changes. The "below freezing" message will appear when the temperature dips below freezing, and disappear when it rises above freezing, all without having to write any additiona logic. Blaze makes this work simply by calling `Tracker.autorun` around its calls out to our helper functions.

What does it look like on the other side, for package authors that are creating new reactive data sources? Here's what the implementation of `currentTemperatureCelsius` might look like (supposing you had an object `Thermometer`, with methods `read` and `onChange`):

```
var temperatureDep = new Tracker.Dependency;

var currentTemperatureCelsius = function () {
  temperatureDep.depend();
  return Thermometer.read();
};

Thermometer.onChange(function () {
  temperatureDep.changed();
});
```

As you can see, it only takes a few lines of code to make `Thermometer` Tracker-aware so that it can participate in transparent reactivity.

It's easy for a library to detect if Tracker is available, and cooperate with it if so (by making the appropriate `depend` and `changed` calls, which have virtually no overhead except when called in a reactive context), and function normally if not. So adding Tracker support doesn't break compatibility with existing users of a library, and the support doesn't have any performance impact if it's not used. (The impact when it *is* used is small as well.)

Tracker-aware libraries from the Meteor Project

The Meteor project provides a variety of Tracker-aware libraries:

- Blaze is a reactive templating/DOM update library designed to work well with Tracker.
- Minimongo is a Tracker-aware reimplementation of the MongoDB API in JavaScript, very useful for storing and querying client-side data. We also have a "full stack database driver" for Mongo that replicates data from a real server-side MongoDB database into Minimongo.
- reactive-dict, and reactive-var provide some fundamental reactive data types. session provides a `ReactiveDict` whose contents persist across hot updates to an app's code.
- Other Meteor core packages are generally Tracker-aware wherever it makes sense. For example, the current connection status is reactive in Meteor's DDP implementation, and the currently logged in user is reactive in Meteor Accounts system.

Future directions

Ideas for future work include:

- Unifying reactive-dict and reactive-var, so that instead of making a `ReactiveDict` you can simply put an object into a `ReactiveVar`
- Providing a contract for *settable* reactive values, to make it easier to build bidirectionally bound forms. There is some discussion of this here.
- Making it possible to control invalidation order, for example by specifying that if autorun B was created inside autorun A, then if both A and B are invalidated and eligible to be rerun, then A will rerun before B. It is easy to construct theoretical situations where this would be valuable, but situations where this makes a difference in real apps seem to be surprisingly rare.

Also as described on the project pages for Mini databases and Full stack database drivers, the Meteor project intends to eventually sponsor development of Tracker-aware libraries that emulate other server-side databases besides Mongo.