Meteor Tracker

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

Transparent Reactive Programming

A basic problem when writing software, especially application software, is monitoring some value – like a record in a database, the currently selected item in a table, the width of a window, or the current time – and updating something whenever that value changes.

There are several common patterns for expressing this idea in code, including:

- **Poll and diff.** Periodically (every second, for example), fetch the current value of the thing, see if it's changed, and if so perform the update.
- **Events.** The thing that can change emits an event when it changes. Another part of the program (often called a controller) arranges to listen for this event, and get the current value and perform the update when the event fires.
- **Bindings.** Values are represented by objects that implement some interface, like BindableValue. Then a 'bind' method is used to tie two BindableValues together so that when one value changes, the other is updated automatically. Sometimes as part of setting up the binding, a transformation function can be specified. For example, Foo could be bound to Bar with the transformation function of toUpperCase.

Another pattern, not yet as commonly used but very well suited for complex modern apps, is **reactive programming**. A reactive programming system works like a spreadsheet. The programmer says, "the contents of cell A3 should be equal to the sum of cells B1-B6, multiplied by the value of cell C4", and the spreadsheet is responsible for automatically modeling the flow of data between the cells, so that when cell B2 changes the value of cell A3 will be automatically updated.

So reactive programming is a **declarative** style of programming, in which the programmer says what should happen (A3 should be kept equal to the result of a certain computation over the other cells), not how it should happen, as in imperative programming.

Reactive programming is perfect for building user interfaces, because instead of attempting to model all interactions in a single piece of cohesive code the programmer can express what should happen upon specific changes. The paradigm of responding to a change is simpler to understand than modeling which changes affect the state of the program explicitly. For example, suppose that we are writing an HTML5 app with a table of items, and the user can click on an item to select it, or ctrl-click to select multiple items. We might have a `<h1>` tag and

want the contents of the tag to be equal to the name of the currently selected item, capitalized, or else "Multiple selection" if multiple items are selected. And we might have a set of `<tr>` tags and want the CSS class on each `<tr>` to be 'selected' if the items corresponding to that row is in the set of selected items, or the empty string otherwise.

Reactive programming is a perfect fit for a situation like this:

- If we use **poll and diff**, the UI will be unacceptably laggy. After the user clicks, the screen won't actually update until the next polling cycle. Also, we have to store the old selection set and diff it against the new selection set, which is a bit of a hassle.
- If we use **events**, we have to write some fairly tangled controller code to manually map changes to the selection, or to the name of the selected item, onto updates to the UI. For example, when the selection changes we have to remember to update both the `<h1>` and (typically) two affected `<tr>`s. What's more, when the selection changes we have to automatically register an event handler on the newly select item so that we can remember to update the `<h1>`. This controller code is hard to structure cleanly and to maintain, especially as the UI is extended and redesigned.
- If we use **bindings**, we will have to use a complex DSL (domain specific language) to express the complex relationships between the variables. The DSL will have to include indirection (bind the contents of the `<h1>` not to the name of any fixed item, but to the item indicated by the current selection), transformation (capitalize the name), and conditionals (if more than one item is selected, show a placeholder string).

Reactive programming promises to let us express this problem exactly as we think about it, giving us concise, readable, maintainable code. The only problem? Historically, reactive programming systems have been too complicated and slow for most people to use in production. They required reactive expressions to be modeled as a tree of nodes, and had to run an expensive dataflow analysis every time a change happened in order to know what to update.

Meteor Tracker is a new reactive programming system that is incredibly simple and lightweight. It is **small** (around 1kb, gzipped and minifed), **simple** (you just write normal JavaScript and it's automatically reactive!), and **fast** (no expensive dataflow analysis – a reactive update is about as expensive as firing an event).

When writing modern HTML5 applications, Tracker is your new best friend!

A Simple Example

```
var favoriteFood = "apples";
var favoriteFoodDep = new Tracker.Dependency;

var getFavoriteFood = function () {
```

```
  favoriteFoodDep.depend();
  return favoriteFood;
};

var setFavoriteFood = function (newValue) {
  favoriteFood = newValue;
  favoriteFoodDep.changed();
};

getFavoriteFood();
// "apples"
```

This code should look familiar. It keeps track of a single value, `favoriteFood`, which is the user's favorite food as a string. And it provides a function `getFavoriteFood` to get the current favorite food and a function `setFavoriteFood` to set it.

But it also contains three additional lines – the creation of a `Dependency` object, a call to `depend` on that object when the current value of `favoriteFood` is read, and a call to `changed` on that object when `favoriteFood` changes. That's all it takes to make `favoriteFood` a fully-fledged **reactive value**. You can learn more about how to create and use reactive values in the section *"Creating Reactive Values"* below.

So what's a "reactive value" and how do we use it?

```
var handle = Tracker.autorun(function () {
  console.log("Your favorite food is " + getFavoriteFood());
});
// "Your favorite food is apples"

setFavoriteFood("mangoes");
// "Your favorite food is mangoes"
```

`Tracker.autorun` immediately runs the function passed to it, printing "Your favorite food is apples".

But wait, there's more! Whenever the user's favorite food changes, the autorun runs its function again. So when `setFavoriteFood("mangoes")` is called, the autorun is retriggered and "Your favorite food is mangoes" is printed. This happens because the user's favorite food is a reactive value (that is, because it calls `changed` from its setter and `depend` from its getter).

Notice that this happens without registering for any events or setting up any bindings! The autorun simply logs the reactive values accessed while running its function, and reruns the function whenever any of those values change.

```
setFavoriteFood("peaches");
// "Your favorite food is peaches"
setFavoriteFood("bananas");
```

```
// "Your favorite food is bananas"
handle.stop();
setFavoriteFood("cake");
// (nothing printed)
```

The autorun remains active, printing a line every time the favorite food changes.

Like `setTimeout` or `setInterval`, when `Tracker.autorun` is called, it returns a handle that can be used to stop the autorun later. Stopping the autorun cleans it up and causes it to no longer run, so nothing is printed after setting the favorite food to "cake".

What if the autorun doesn't call `getFavoriteFood` directly? What if there is some other function – `getReversedFood` – in between `getFavoriteFood` and the autorun?

```
var getReversedFood = function () {
  return getFavoriteFood().split("").reverse().join("");
};
getReversedFood();
// "ekac"

var handle = Tracker.autorun(function () {
  console.log("Your favorite food is " + getReversedFood() + " when reversed");
});
// Your favorite food is ekac when reversed
setFavoriteFood("pizza");
// Your favorite food is azzip when reversed
```

No problem at all. **Autoruns can make arbitrary blocks of JavaScript code reactive.** The code can call other functions, do arbitrary computations on the reactive values, or use the values of some reactive values to decide what other reactive values to look at. It's just plain JavaScript!

This is a very powerful idea because for many users, it enables **transparent reactive programming** – reactive programming with no code changes. The only code that needs to know about Tracker are **providers** of reactive values (such as a database library) and **consumers** of reactive values (such as a frontend library like Blaze, React, or famo.us). The rest of the code – the application code itself, that does database queries to produce values to be used in the UI – can be written without even knowing that the reactive programming system exists. This makes applications fun and super fast to write, and incredibly easy to maintain, compared to other systems like polling, manually wired up events, or bindings.

The next section shows how transparent reactive programming can be used in a real application to make a live-updating user interface.

Building a User Interface with Transparent Reactive Programming

In the previous section we created a reactive value from scratch (the `getFavoriteFood` and `setFavoriteFood` functions). In practice, when you're writing a real application, you'll probably never need to do this. Instead you'll probably use libraries that already have Tracker support, so that when you do a database query, or get the current time or the size of the browser window, the library handles the reactive value's registration for you. (It's easy to add Tracker support to a library – often little more than a call to `depend` and a call to`changed` in the right places.)

One library you can use is ReactiveDict. It provides a simple dictionary object whose `get(key)` and `set(key, value)` functions are fully reactive.

```
var forecasts = new ReactiveDict;
forecasts.set("san-francisco", "cloudy");
forecasts.get("san-francisco");
// "cloudy"
```

Now let's put the current forecast for San Francisco on the screen, and make it reactively update whenever the forecast changes. The code to do this is shockingly short.

```
$('body').html("The weather here is <span class='forecast'></span>!");
Tracker.autorun(function () {
  $('.forecast').text(forecasts.get('san-francisco'));
});
// Page now says "The weather here is cloudy!"

forecasts.set("san-francisco", "foggy");
// Page updates to say "The weather here is foggy!"
```

In a real application you'd probably never need to write even that much code. You'd use a library like Meteor Blaze, which lets you instead use a templating syntax such as:

```
<template name="weather">
  The weather here is {{forecast}}!
</template>

// In app.js
Template.weather.forecast = function () {
  return forecasts.get("san-francisco");
};
```

Blaze parses the template and sets up autoruns to update all of the elements as necessary. But fundamentally, what Blaze is doing is very simple – it's simply letting you express your app's interface in HTML templates instead of in JavaScript code.

The power of Tracker comes from the fact that you can use arbitrary JavaScript to compute what will appear on the screen, and it will be automatically reactive. Here's a more complex example:

```
var forecasts = new ReactiveDict;
forecasts.set("Chicago", "cloudy");
forecasts.set("Tokyo", "sunny");

var settings = new ReactiveDict;
settings.set("city", "Chicago");

$('body').html("The weather in <span class='city'></span> is <span class='weather'></span>.'
Tracker.autorun(function () {
  console.log("Updating");
  var currentCity = settings.get('city');
  $('.city').text(currentCity);
  $('.weather').text(forecasts.get(currentCity).toUpperCase());
});
// Prints "Updating"
// Page now says "The weather in Chicago is CLOUDY."
```

In this example, one reactive variable ("city" in `settings`) is used to determine what other reactive variable to look at (the appropriate `forecast` for that city), and also the forecast is changed to upper case before it's used.

When we change any of the reactive values that actually went into rendering the page, the page automatically updates:

```
settings.set("city", "Tokyo");
// Prints "Updating"
// Page updates to "The weather in Tokyo is SUNNY."

forecasts.set("Tokyo", "wet");
// Prints "Updating"
// Page updates to "The weather in Tokyo is WET."
```

But, when we change a reactive value that wasn't needed to render the page, nothing happens, because Tracker knows that that value wasn't actually used.

```
forecasts.set("Chicago", "warm");
// Does *not* print "Updating"
// No work is done
```

Like any app written with Tracker, this code naturally breaks down into three parts:

- The **reactive data source**, in this case the ReactiveDict library. A real app will use a variety of libraries that provide reactive values, like a database library that lets you do arbitrary reactive database queries (such as Meteor's Minimongo, which provides a reactive implementation of the

entire MongoDB query API), or simple libraries that provide the current time or window size as a reactive value.
- The **reactive data consumer**, in this case a simple autorun that uses jQuery to update a DOM element, but in a real app more likely a reactive templating library like Meteor Blaze.
- The **application logic**, in this case the actual code that fetches the appropriate forecast and capitalizes it.

The key idea of Tracker is that the application code doesn't have to be aware of reactivity at all! This is **transparent reactive programming**. The application code just uses the public APIs of the reactive data sources (`currentCity.get()`, or a normal MongoDB query using Minimongo, for example). The application code never has to call any of the Tracker APIs. Only creators of new reactive data sources (like implementors of new databases), or of new reactive data consumers (like new templating systems), need to call the Tracker API or even be aware that Tracker exists.

Tracker works behind the scenes to set up the reactivity. How does this work?

How Tracker Works

The one-sentence summary is that `Tracker.autorun` logs all of the reactive values that are accessed while running the autorun's function, and when any of those reactive values change, they fire an event that causes the autorun to rerun itself.

More precisely:

Each call to `Tracker.autorun` creates an object called a `Computation`. Whenever an autorun is executing code, the global variable `Tracker.currentComputation` is set to the `Computation` that goes with that autorun. (When no autorun is running, `Tracker.currentComputation` is null.)

A reactive data source (for example the `get()` method of `ReactiveDict`) checks to see if there is a `currentComputation`, that is, if it's being called from inside an autorun. If so, it holds on to a reference to that `Computation` and arranges to call a method on it when the relevant data (for example, the referenced key on the `ReactiveDict`) changes in the future. This tells the `Computation` created by the autorun to rerun itself.

In other words, `Tracker` simply provides a global variable that is a rendezvous point between data that could change, and code that needs to know when the data changes. The global variable lets a reactive data source figure out which autoruns to notify when a reactive value changes.

The following sections describe this system in complete detail.

Monitoring Reactive Values

In `Tracker`, the usual way to monitor a reactive value is with `Tracker.autorun`. `Tracker.autorun` takes one argument, a function that might (or might not) end up referencing one or more reactive values when it's executed. `Tracker.autorun` runs the function immediately, before returning a result, and arranges to run the function again whenever the referenced reactive values change.

Each call to `Tracker.autorun` creates a new `Computation` object, which represents the autorun. Calling the `stop()` method on the `Computation` cleans up the autorun and stops it from ever rerunning again.

Autoruns Can Be Nested Inside Autorun

```
var weather = new ReactiveDict;

weather.set("sky", "sunny");
weather.set("temperature", "cool");

var weatherPrinter = Tracker.autorun(function () {
  console.log("The sky is " + weather.get("sky"));
  var temperaturePrinter = Tracker.autorun(function () {
    console.log("The temperature is " + weather.get("temperature"));
  });
});
// "The sky is sunny"
// "The temperature is cool"
```

In the example above, the `temperaturePrinter` autorun is nested inside the `weatherPrinter` autorun. The first time through, both autoruns run the functions passed to them, printing out "The sky is sunny" and "The temperature is cool".

```
weather.set("temperature", "hot");
// "The temperature is hot"
```

When there are nested autoruns, dependencies accrue to the innermost enclosing autorun. So changing `temperature` causes `temperaturePrinter` to rerun, but not `weatherPrinter`, because `temperaturePrinter` is the innermost autorun around the call to `weather.get("temperature")`.

```
weather.set("sky", "stormy");
// The sky is stormy
// The temperature is hot
```

On the other hand, changing `sky` causes both messages to print. That's because the change to `sky` triggers a rerun of the outer `weatherPrinter` autorun, which tears down and recreates the inner `temperaturePrinter` autorun in the course of rerunning itself.

Even though a new `temperaturePrinter` autorun is created every time the `weatherPrinter` autorun is rerun, it's not necessary to call `stop()` on the old `temperaturePrinter` autoruns. This is because of the automatic cleanup convention, discussed in the next section.

When is it advantageous to nest autoruns? Most commonly in libraries such as Blaze, in cases where control is desired over the *granularity of reactivity*. Putting the two `console.log` lines in two different autoruns creates *finer-grained reactivity*, because if only the temperature changes, only the second `console.log` needs to be rerun. If both `console.log` lines had been put in the same autorun, any change to either variable would cause both messages to be printed. This would be *coarse-grained reactivity*. By using autoruns as *reactivity boundaries* in this way, it's possible for libraries like Blaze to tightly optimize the amount of work that is done when a given reactive value changes, though this is not typically necessary in application code.

The Automatic Cleanup Convention

For a library to be fully compatible with Tracker, it should obey this convention: if an object is created inside an autorun that needs to be stopped or cleaned up later, it should be automatically stopped whenever that autorun reruns itself. Why is this the correct convention? Because presumably, the autorun will recreate the object when it reruns itself, at least if the object is still wanted.

Here's an example of how this convention might be used inside a Meteor application:

```
Tracker.autorun(function () {
  if (Session.get("subscribeToNewsFeed")) {
    Meteor.subscribe("newsFeed", Meteor.userId());
  }
});
```

This code opens a DDP subscription to the current user's news feed, but only if the "subscribeToNewsFeed" flag is set. If the "subscribeToNewsFeed" flag is set to false, the subscription, if any, is destroyed (because the autorun will rerun itself and will fail to recreate the subscription). Likewise, if the user logs in as a different user, the subscription is torn down (because of the automatic cleanup convention) and replaced with a new subscription that points to the newly logged in user (as a result of the autorun rerunning and calling `Meteor.subscribe` with the new value of `Meteor.userId()`).

When you nest autoruns, autorun itself follows the automatic cleanup convention. Inner autoruns will be cleaned up when outer autoruns are rerun.

```
weatherPrinter.stop();
weather.set("temperature", "chilly");
// Nothing happens
```

We stopped `weatherPrinter` but not `temperaturePrinter`. Nevertheless, when we change the temperature nothing is printed, because stopping `weatherPrinter` automatically stopped `temperaturePrinter` as the latter was created from inside the former.

When Autoruns Throw Exceptions

If an autorun's function throws an exception the first time it is run, the exception propagates out of `Tracker.autorun` to the caller. No autorun is actually started, any `Tracker.Computation` created is destroyed, and the function is never rerun. This is usually the most convenient behavior. (It's useful to propagate the exception, and since an exception is being raised no `Tracker.Computation` can be returned, meaning that if an autorun were created it would be difficult for the caller to stop it.)

If an autorun's already been run once without throwing an exception, it continues to run regardless of whether it throws exceptions on subsequent reruns.

Accessing the Current Computation Inside an Autorun

The autorun function is called with one argument: the `Computation` object that represents this particular autorun.

```
var partyInfo = new ReactiveDict;
partyInfo.set("status", "going-strong");

Tracker.autorun(function (comp) {
  if (partyInfo.equals("status", "over")) {
    console.log("Mom's home! Get out! The party's over!");
    comp.stop();
    return;
  }

  runDiscoLights();
  crankTheMusic();
  eatPixieSticks();
});
```

This autorun will self-destruct (stop itself) if the "status" flag of the party is ever set to "over".

The following code will **not work**:

```
// WRONG
var partyRunner = Tracker.autorun(function () {
  if (partyInfo.equals("status", "over")) {
    partyRunner.stop(); // WRONG
    return;
  }
```

```
  ...
});
```

This code is almost right, but will fail to stop the party if the "status" flag has already been set to "over" on the first time the autorun runs. That's because that at that point, `Tracker.autorun` has not returned yet, so the value of `partyRunner` is still `undefined`. This is exactly the reason why the autorun function receives the current Computation as an argument.

Detecting the First Run

Often, an autorun has logic that's specific to the first run. The `firstRun` flag on `Computation` reveals whether the autorun is being run for the first time.

```
var book = new ReactiveDict;
book.set("title", "A Game of Thrones");
Tracker.autorun(function (c) {
  if (c.firstRun) {
    console.log("Curling up on the couch to read.");
  }
  console.log("Now reading: " + book.get("title"));
});
// "Curling up on the couch to read."
// "Now reading: A Game of Thrones"
```

On the first run, the "Curling up on the couch to read" message is printed, because `firstRun` is true.

```
book.set("title", "A Clash of Kings");
// "Now reading: A Clash of Kings"
```

When we change `title`, the autorun runs again, but the "Curling up on the couch to read" message is not printed because it is not the first time through the function.

Ignoring Changes to Certain Reactive Values

Occasionally it is useful for an autorun to ignore changes to a particular reactive value, even though that value was used in the course of running the autorun. This can be done by wrapping the code that accesses that value inside `Tracker.nonreactive`.

```
var game = new ReactiveDict;
game.set("score", 42);
game.set("umpire", "Giraffe");

Tracker.autorun(function () {
  Tracker.nonreactive(function () {
    console.log("DEBUG: current game umpire is " + game.get("umpire"));
```

```
  });
  console.log("The game score is now " + game.get("score") + "!");
});
// "DEBUG: current game umpire is Giraffe"
// "The game score is now 42!"
```

When "score" changes, the autorun is updated and new messages are printed. Changes to the umpire are considered less interesting, though, so even though we record the current umpire whenever we print the score, an umpire change is not sufficient to trigger an update.

```
  game.set("umpire", "Hippo");
  // (nothing printed)

  game.set("score", 137);
  // "DEBUG: current game umpire is Hippo"
  // "The game score is now 137!"
```

Fine-Grained Reactivity

When using reactive APIs, it is best to ask only for the data that your app actually needs. For example, if your app stores `Person` objects, and needs only the given and family name of a particular `Person`, it should retrieve only those fields and not other fields, like mailing address or date of birth, that aren't needed.

This improves performance for two reasons:

- First of all, in most any system, retrieving less data will be faster and will use less memory.
- Second, in the Tracker system of transparent reactivity, retrieving data is the same as subscribing to change notifications for that data.

By retrieving a piece of data like a `Person` object's date of birth, you're indicating that you want your code to be rerun whenever that `Person`'s date of birth changes. So to limit the amount of recomputation that happens, you should only retrieve the date of birth if it's actually needed for the computation that you're doing. (Of course, this only matters for code that is actually run in a reactive context, like an autorun, a reactive HTML template, and so forth.)

Well-designed reactive APIs support *fine-grained* queries. They let you ask for just the data that you need, like the family name of a `Person` object without the other fields. This improves efficiency because the app will be notified only when the family name of the `Person` changes, not when the other fields like date of birth change. The opposite of *fine-grained* is *coarse-grained*. A query retrieving all of the fields on a `Person` object might be called *coarse-grained* if most of the fields were unneeded, especially if the unneeded fields changed very frequently and this created a performance problem.

Fine-grained reactivity can improve performance, but it's also important to not go overboard in creating incredibly fine-grained reactivity when it's not needed for performance. For example, there probably isn't any point in an ultra-fine-grained API that lets you retrieve just the first character of the name of a `Person`.

Here are some examples of fine-grained reactivity in common Meteor APIs.

ReactiveDict:`get` versus `equals`

What's the difference between these two ways of writing an autorun?

```
var data = new ReactiveDict;

// VERSION 1 (INEFFICIENT)
Tracker.autorun(function () {
  if (data.get("favoriteFood") === "pizza")
    console.log("Inefficient code says: Time to get some pizza!");
  else
    console.log("Inefficient code say: No pizza for you!");
});

// VERSION 2 (MORE EFFICIENT)
Tracker.autorun(function () {
  if (data.equals("favoriteFood", "pizza")) // CHANGED LINE
    console.log("Efficient code says: Time to get some pizza!");
  else
    console.log("Efficient code says: No pizza for you!");
});
```

The first version ("Inefficient") prints a message whenever "favoriteFood" changes. The second ("More efficient") version is finer-grained: it prints a message only when "favoriteFood" changes to or from "pizza".

```
data.set("favoriteFood", "apples");
// Inefficient code says: No pizza for you!
// Efficient code says: No pizza for you!
data.set("favoriteFood", "pears");
// Inefficient code says: No pizza for you!
data.set("favoriteFood", "oranges");
// Inefficient code says: No pizza for you!
data.set("favoriteFood", "pizza");
// Inefficient code says: Time to get some pizza!
// Efficient code says: Time to get some pizza!
data.set("favoriteFood", "pancakes");
// Inefficient code says: No pizza for you!
// Efficient code says: No pizza for you!
```

The coarse-grained version calls `get("favoriteFood")` on the `ReactiveDict`, so it depends on the value of "favoriteFood", and whenever "favoriteFood" changes the autorun has to rerun. This results in a lot of wasted work, because the autorun doesn't actually care about the value of "favoriteFood". It only cares if "favoriteFood" is "pizza".

On the other hand, the fine-grained version calls `equals("favoriteFood", "pizza")` on the `ReactiveDict`, so it does not depend on the value of "favoriteFood", but rather *whether "favoriteFood" has a specific value, "pizza"*.

You should use `equals` in preference to `get` whenever possible, because it will often be radically faster. For example, suppose there is a table with 1000 rows, that there is a ReactiveDict with a key named "selection" that indicates the currently selected row, and that each row should receive the "selectedRow" CSS class if it is selected. If `equals` is used to determine if a row should get the CSS class, then when "selection" changes, only two rows have to be checked for CSS class updates, the previously selected row and the newly selected row. If `get` is used, then all 1000 rows will have to be checked for CSS class updates.

Collection.find() versus Collection.cursor.fetch()

In Meteor, a Collection is used for storing multiple records or "documents" that are all of a similar type. For example, a User Collection might contain a record for every user of an application.

`Collection.find()` returns a cursor, which represents a query to a Collection in the abstract — it doesn't return any actual documents. Calling `cursor.fetch()`, on the other hand, returns the documents matched by the query. In addition, when `cursor.fetch()` is called from inside an autorun, it creates dependencies for all the documents matched by the call to `find()`. Inside an autorun, just using `Collection.find()` doesn't set up any reactive dependencies.

For example, calling `Posts.find()` will return a cursor that *represents* a query for all the posts in an application, while calling `Posts.find({tag: "kittens"})` will give you a cursor representing a query for all posts about kittens. To get all of the posts about kittens, you can call `Posts.find({tag: "kittens"}).fetch()`.

In order to enable *fine-grained reactivity*, you can pass *selectors* into `find()` to narrow the scope of the query. There are two main types of selectors, *property selectors* and *field selectors*. *Property selectors* (in the format `{property: constraint}` are used to fetch documents whose `property` property satisfies some constraint. For example, the `{tag: "meteor"}` selector, when passed into `Posts.find`, returns only the Posts that have been tagged with "meteor". Alternatively, a selector `{tag: {$in: ["meteor", "node", "javascript"]}}` would return posts tagged with "meteor", "node", or "javascript".

```
// VERSION 1 (INEFFICIENT)
Tracker.autorun(function(){
  var allPosts = Posts.find().fetch();
```

```
  for (var post in allPosts) {
    if (post.tag === "kittens") console.log(post.title);
  }
});

// VERSION 2 (MORE EFFICIENT)
Tracker.autorun(function(){
  var postsAboutKittens = Posts.find({tag:"kittens"}).fetch();
  postsAboutKittens.forEach(function (post) {
    console.log(post.title);
  });
});
```

The code above shows the advantage of using a *property selector*. Both autoruns print out the titles of all posts tagged "kittens". However, the "inefficient" version reruns any time a Post (of any sort) is added or deleted, while the "efficient" version only reruns when a Post tagged with "kittens" is edited, added, or deleted. This is because the "inefficient" version calls`Posts.find().fetch()`, which makes the autorun depend on the values of all Posts, but the "efficient" version calls `Posts.find({tag:"kittens}).fetch()`, which only sets up dependencies on Posts tagged "kittens".

In addition to *property selectors*, you can use a *field selector* to reduce the number of fields in the documents returned by a query. Inside an autorun, using a field selector reduces the number of fields that can cause the autorun to rerun.

```
Tracker.autorun(function(){
  var postAuthors = Posts.find({}, {fields: {tag: 0, author: 1, title: 0}}).fetch();
  postAuthors.forEach(function (post) {
    console.log(post.author);
  });
});
```

Calling `Posts.find({}, {fields: {author: 1,title: 0}}).fetch()` (where the fields selector is `{author: 1,title: 0}`) returns a list of all the authors of Posts. The autorun will rerun whenever the author of any Post changes, but not if the author decides to change the title of the Post.

Collection.cursor.count()

`cursor.count()` returns the *number* of documents matched by a cursor, as opposed to `fetch()`, `forEach()`, or `map()`, which return the documents themselves. Likewise, when called from inside an autorun, `cursor.count()` sets up a dependency specifically on the number of documents matched by the cursor. This means that if `cursor.count()` is called from within an autorun, the autorun only reruns when the number of documents matched by the cursor changes.

```
// VERSION 1: INEFFICIENT
Tracker.autorun(function(){
  console.log(Posts.find({tags:"kittens"}).fetch().length);
});

// VERSION 2: EFFICIENT
Tracker.autorun(function(){
  console.log(Posts.find({tags:"kittens"}).count());
});
```

Both of the autoruns above print out the number of posts about kittens. However, the "inefficient" autorun will rerun any time a post about kittens is added, removed, *or edited*. The second autorun only reruns when the number of posts about kittens changes, which means that it won't rerun if a post is merely edited, but does rerun if a post is added or removed.

Guidelines for Fine-Grained and Coarse-Grained Reactivity

Here are some general guidelines for using fine-grained reactivity.

- When fine-grained reactivity is as easy as coarse-grained reactivity, prefer fine-grained. In other words, don't go out of your way to fetch data that you don't need.
- If your app is slow, profile it. If you discover that an expensive function is being called far too often because it depends on data it doesn't actually use, switch to finer-grained functions.
- When designing an API for a library, try to make fine-grained reactivity the default, but don't compromise ease of use to accomplish this. If you can't make fine-grained reactivity the default, consider providing a fine-grained API for users that need more performance.
- When writing a library, a good approach is to match the level of reactivity to what your library can efficiently compute. For example, the `equals` method on `ReactiveDict` is valuable precisely because `ReactiveDict` can compute it more efficiently than the application could, by separately tracking the autoruns that depend on each possible value of each variable. It only takes `ReactiveDict` O(1) time to determine which `equals` callers should be notified when the value of a key changes, but if you tried to implement `equals` yourself on top of the `get` function on `ReactiveDict`, it would take O(n) time.

Creating Reactive Values

It the heart of `Tracker` is the notion of a **reactive value** – a value that changes from time to time, that can be read by calling a getter function, and that will trigger any affected autoruns to be rerun whenever it changes.

A reactive value could represent:

- The currently selected item in a table in a user interface
- The result of a database query (changing whenever the underlying information in the database changes)
- In a web browser, the current size of the browser window, or (in a browser that supports the HTML5 pushState API) the current URL
- A string such as "5 minutes ago" or "last Tuesday" that gives the difference between a particular time and the current time

`Tracker` makes it easy to create new types of reactive values. This work is usually done by library authors that want to built reactivity support into their libraries, or by application authors that need a particular, custom kind of reactive value that isn't provided by any existing library.

This section shows two easy ways to create a reactive value: by using the `ReactiveDict` library to do the work for you, and by doing the work yourself using the `Tracker.Dependency` object. As an advanced topic, it also shows a not-so-easy way to create a reactive value, by bypassing the `Tracker.Dependency` object and using the underlying `Tracker.Computation` API, and it shows how `Tracker.Dependency` is just a convenient wrapper around that lower-level API.

Finally, it covers a couple of topics mostly of interest to library implementors: examples to illustrate the many-to-many relationship between `Tracker.Dependency` and `Tracker.Computation`; and an example that shows how one might use Tracker to take a "wait and see" approach to recurring behaviors such as subscriptions.

Creating a Reactive Value Using the ReactiveDict Library

An incredibly simple way to create a reactive value is to use the `ReactiveDict` library. Since `ReactiveDict` implements a reactive dictionary, making a reactive value is as simple as creating a `ReactiveDict` and using its `get()` and `set()` methods.

```
var data = new ReactiveDict;

var getFavoriteFood = function () {
  return data.get("favoriteFood");
};

var setFavoriteFood = function (newValue) {
  data.set("favoriteFood", newValue);
};
```

This simple approach is often all that's needed to create a quick reactive value for an application. One limitation is that at present, values in a `ReactiveDict` may not be objects or arrays, only strings, numbers, booleans, Dates, null, and undefined.

`ReactiveDict` is not included in the main Tracker library. Rather, it's a small

(~600 byte) utility library that you can use alongside Tracker. The command `meteor add reactive-dict` will add it to a Meteor app.

Some Examples of Custom Reactive Values Using ReactiveDict

Here's a reactive value that gives the current time, with updates triggered every second:

```
var data = new ReactiveDict;

data.set("now", new Date);
setTimeout(function () {
  data.set("now", new Date);
}, 1000);

var currentTime = function () {
  data.get("now");
};
```

Here's a reactive value that gives the current size of the browser window:

```
var data = new ReactiveDict;

var updateWindowSize = function () {
  data.set("width", window.innerWidth);
  data.set("height", window.innerHeight);
};
updateWindowSize();
window.addEventListener("resize", updateWindowSize);

var windowSize = function () {
  return {
    width: data.get("width"),
    height: data.get("height")
  };
};
```

Creating a Reactive Value Using Tracker.Dependency

The other common way to create reactive values is to use the `Tracker.Dependency` helper object, which is included in the main Tracker library.

To see how `Tracker.Dependency` is used, first suppose that we had a variable `favoriteFood` and getter and setter functions `getFavoriteFood` and `setFavoriteFood` that we could use to access it:

```
var favoriteFood = "apples";

var getFavoriteFood = function () {
```

```
  return favoriteFood;
};

var setFavoriteFood = function (newValue) {
  favoriteFood = newValue;
};

`favoriteFood` can be made into a reactive variable by adding just four lines of code.

var favoriteFood = "apples";
var favoriteFoodDep = new Tracker.Dependency;  // FIRST ADDED LINE

var getFavoriteFood = function () {
  favoriteFoodDep.depend();                    // SECOND ADDED LINE
  return favoriteFood;
};

var setFavoriteFood = function (newValue) {
  if (newValue !== favoriteFood)               // THIRD ADDED LINE
    favoriteFoodDep.changed();                 // FOURTH ADDED LINE
  favoriteFood = newValue;
};
```

The first added line creates a new `Tracker.Dependency` object called `favoriteFoodDep` that will store the list of autoruns that care about the current value of `favoriteFood`.

In the second added line, whenever the value of `favoriteFood` is read, the current autorun (if any) is added to `favoriteFoodDep`. This is done by calling the `depend()` function on `favoriteFoodDep`. This will do nothing if `getFavoriteFood()` is not being called from inside an autorun.

Finally, in the third and fourth added lines, whenever the value of `favoriteFood` changes, all of the autoruns that were recorded in `favoriteFoodDep` are told to rerun themselves by calling `changed()` on `favoriteFoodDep`. For efficiency, but also to avoid change loops, `changed()` is called only if the new value is different from the old value.

(It's important to use `changed()` only if the value of `favoriteFood` has actually changed — not if it only could potentially have changed! See "Avoiding Change Loops" in the section *"The Flush Cycle"* below.)

This is all that you need to know to use `Tracker.Dependency`: create a new `Tracker.Dependency` for each distinct reactive value that you are maintaining, call `depend()` whenever you read that reactive value, and call `changed()` whenever that reactive value changes.

The `Tracker.Computation` Object

The `Tracker.Dependency` is actually just a convenient helper object, provided for your convenience. It is built on top of the fundamental Tracker object, `Tracker.Computation`.

A `Tracker.Computation` object is created every time `Tracker.autorun` is called. It represents that particular instance of an autorun, and it has a method `stop()` that terminates and cleans up the autorun.

Computations also have two other methods:

- `invalidate()`: Cause this autorun to rerun itself.
- `onInvalidate(callback)`: Cause `callback` to be called the next time this autorun reruns itself, or when it is stopped.

They also have three attributes, `stopped`, `invalidated`, and `firstRun`, whose meanings are easy to guess (and are fully specified in the documentation).

Code can look at the global variable `Tracker.currentComputation` to see if it is running from inside an autorun. Outside an autorun, `Tracker.currentComputation` is null; inside an autorun, it is the `Tracker.Computation` object that represents the autorun. When code uses`Tracker.Computation` inside nested autoruns, that `Tracker.Computation` object represents the code's "parent" autorun, or the smallest autorun that completely contains the code.

A `Tracker.Dependency` is simply a container for storing a set of `Tracker.Computation` objects. `depend()` adds `Tracker.currentComputation`, if any, to the set. And `changed()` calls `invalidate()` on every computation in the set.

A `Tracker.Dependency` also sets up an `onInvalidate` callback on every computation that it tracks. Whenever a computation is invalidated, the callback removes it from the set of computations being tracked by the `Tracker.Dependency`. Computations are invalidated by a call to `changed()` on a `Tracker.Dependency`, by a call to `changed()` on some *other* `Tracker.Dependency` that is used by the same computation, or by a call to `stop()` on an autorun.

Sound simple? It is! `Tracker.Dependency` is something that you could have implemented yourself on top of the `Tracker.Computation` API. Here's how you might do it:

```
Dependency = function () {
  this._nextId = 0;
  this._dependents = {};
};

Dependency.prototype.depend = function () {
  var self = this;
```

```
  if (Tracker.currentComputation) {
    var id = self._nextId++;
    self._dependents[id] = Tracker.currentComputation;
    Tracker.currentComputation.onInvalidate(function () {
      delete self._dependents[id];
    });
  }
};

Dependency.prototype.changed = function () {
  for (var id in this._dependents) {
    this._dependents[id].invalidate();
  }
};
```

A `Dependency` is just a set of `Tracker.Computation` objects, stored in `_dependents`. For easy removal of computations from the set, `_dependentsis` structured as an object, with unique integers as keys and computations as values.

`depend()` does nothing if there is no `currentComputation`. If there is a `currentComputation`, `depend()` assigns a unique `id` (`_nextId++`) to it, adds it to the set of computations in `_dependentsById` , and sets up an `onInvalidate` callback to remove it prior to rerunning the autorun. (If the autorun still depends on this reactive variable when it is rerun, in the course of rerunning it will end up calling `depend()` to re-add itself.) The `onInvalidate` callback also removes the computation from `_dependentsById` if the autorun is stopped.

`changed()` simply calls `invalidate()` on all of the `_dependentsById`. It doesn't need to remove the `_dependentsById` from the set because this will happen through the `onInvalidate` callbacks that we've already set up.

The code above is not exactly the same as the official implementation of `Tracker.Dependency`. The real implementation is a few lines longer. These extra lines make sure that a given computation is stored only once in `_dependentsById`, no matter how many times `depend()` is called, and allow a computation other that `currentComputation` to be passed to `depend()`. They also provide a function `hasDependents()` that tests to see if `_dependentsById` is empty.

Creating a Reactive Value the Hard Way

In rare cases it's useful to bypass the convenience of `Tracker.Dependency` and use the `Tracker.Computation` object directly. Here's an example. This function returns a string like "17 minutes ago" giving the difference between a time in the past and the current time.

```
var minutesAgo = function (timeInPast) {
var now = new Date;
```

```
var millisecondsAgo = now.getTime() - timeInPast.getTime();
var minutesAgo = Math.floor(millisecondsAgo / 1000 / 60);

  if (Tracker.active) {
    // Save the computation to invalidate when the time changes.
    // (If we just called Tracker.currentComputation inside the setTimeout callback,
    // we would get the wrong result! We would get the *then-current* computation,
    // if any, not the computation that called minutesAgo.)
    var computation = Tracker.currentComputation;

    var timer = setTimeout(function () {
      computation.invalidate();
    }, 60 * 1000 /* one minute */);
    computation.onInvalidate(function () {
      clearTimeout(timer);
    });
  }

  if (minutesAgo < 1) {
    return "just now";
  } else {
    return minutesAgo + " minutes ago";
  }
};
```

If this function is called from inside an autorun, it arranges for the autorun to be rerun in exactly one minute. If the autorun is stopped, the `onInvalidate` handler cleans up the timer so as to avoid wasting memory.

To further understand how the timer is managed with `onInvalidate`, suppose we have an autorun that depends on both a call to `minutesAgo` and some other reactive value.

```
var startTime = new Date;
var data = new ReactiveDict;
data.set("favoriteFood", "peaches");

var logger = Tracker.autorun(function () {
  console.log("The app started " + minutesAgo(startTime) + " and my favorite " +
    "food is " + data.get("favoriteFood") + ".");
});
// Prints "The app started just now and my favorite food is peaches."
// 60 seconds later: "The app started 1 minutes ago and my favorite food is peaches."
```

When the value of “favoriteFood” changes, the `minutesAgo` timer will be canceled and reset to one minute. The change to “favoriteFood” will cause the autorun to be invalidated (canceling the existing timer) and rerun (creating a new one).

The Many-to-Many Relationship Between Computations and Dependencies

To fully understand Tracker, it is critical to understand why the following code is wrong:

```
// WRONG
Dependency = function () {
  this._nextId = 0;
  this._dependents = {};
};

Dependency.prototype.depend = function () {
  if (Tracker.currentComputation) {
    var id = self._nextId++;
    this._dependents[id] = Tracker.currentComputation;
  }
};

Dependency.prototype.changed = function () {
  for (var id in this._dependents) {
    this._dependents[id].invalidate();
  }
  this._dependents = {};
};
// WRONG
```

Compare this code to the implementation of `Dependency` in the earlier section, *The `Tracker.Computation` Object*. Instead of using the computation's `onInvalidate` callback to clean up the `_dependents` list, it's done from `changed`. This is wrong.

An easy way to see that it's wrong is that if an autorun is stopped, its `Tracker.Computation` object will not be removed from the `_dependents` list. This results in wasted memory. For a reactive value that never happens to change (that is, for which `changed()` is never called), this is a permanent memory leak.

But something more fundamental is wrong with this code. There is a many-to-many relationship between Computations and Dependencies. Just as one Dependency can be depended on by several Computations (which the code above gets right), one Computation can reference several Dependencies (which the code above neglects). A computation should be removed from the `_dependents` list *whenever that computation reruns*, no matter whether it was this *particular* Dependency that changed and caused it to rerun, or *some other* Dependency that the computation also used. Thus we have to clean up `_dependents` from an `onInvalidate` callback rather than from `changed`.

Remember, when a Computation reruns, it will re-establish all of its Dependency

relationships. Often this will mean that it tears down its dependencies and recreates exactly the same set every time, as in this code:

```
var comp = Tracker.autorun(function () {
  console.log(data.get("favoriteFood"));
});
```

Each time the autorun is invalidated, `comp` is removed from the dependent list for "favoriteFood", only to be immediately re-added as the autorun reruns.

But that's only because the this autorun is very simple. In general, the dependencies could be different each time the autorun reruns:

```
var comp = Tracker.autorun(function () {
  if (data.get("favoriteFood") === "banana")
    console.log(minutesAgo(someTime));
  else
    console.log("you should really give bananas a chance");
});
```

In this code, setting "favoriteFood" will ultimately cause `minutesAgo` to start or stop a timer depending on the favorite food chosen, and it's all intermediated through the `onInvalidate` callback.

Taking a "Wait and See" Approach

This section shows a library that uses Tracker can improve performance by taking a "wait and see" approach with regard to recurring behaviors. Consider the example below, which automatically subscribes a user to a particular magazine based on their current interests. Users interested in either fashion or fitness are subscribed to *Cosmo*, while users that prefer gossip or celebrities are subscribed to *People*.

```
var data = ReactiveDict;
data.set("currentInterest", "fashion");

Tracker.autorun(function () {
  var currentInterest = data.get("currentInterest");
  if (currentInterest === "fashion" || currentInterest === "fitness")
    subscribeToMagazine("Cosmo");
  if (currentInterest === "celebrities" || currentInterest === "gossip")
    subscribeToMagazine("People");
});
// "Putting a check in the mail to subscribe to Cosmo"

data.set("currentInterest", "gossip");
// "Putting a check in the mail to subscribe to People"
// "Canceling our subscription to Cosmo"
```

```
data.set("currentInterest", "celebrities");
// Nothing printed (People subscription is maintained, not canceled and restarted)
```

If a user that's interested in fashion decides they like celebrities instead, they are unsubscribed from *Cosmo* and auto-subscribed to *People*. But when that user's interest changes again from "fashion" to "gossip", they stay subscribed to *People*. The *People* subscription is maintained *without having to tear down and re-create the original subscription.*

This is what we mean by a "wait and see" approach. When a user's `currentInterest` changes, their subscription isn't deleted immediately. The subscription is maintained until the autorun is rerun; if the user still needs to be subscribed to the same magazine, then the old subscription is reused. In other words, `subscribeToMagazine` *waits to see* whether the user's subscription has truly changed. If it has, a new subscription is created and the old one is deleted; otherwise, the old subscription is maintained.

Below is an implementation of the `subscribeToMagazine` function that takes a "wait and see" approach to subscriptions. `subscribeToMagazine` uses the global variable `subscriptions` to track subscriptions; each key-value pair in `subscriptions` represents a different magazine subscription. Each key is a magazine's name, and each value contains object with a `stop()` handle for stopping the subscription.

```
var subscriptions = {}; // keys are names of magazines
var subscribeToMagazine = function (magazineName) {
  if (! _.has(subscriptions, magazineName)) {
    // We're not currently subscribed to this magazine. Go start a subscription.
    console.log("Putting a check in the mail to subscribe to", magazineName);
    subscriptions[magazineName] = {
      references: 1,
      stop: function () {
        // Wait for all autoruns to finish rerunning before processing the unsubscription
        // request. It's possible that stop() was called because an autorun is rerunning,
        // so before doing anything drastic, we should wait to see if the autorun
        // recreates the subscription when it is rerun.
        this.references--;
        Tracker.afterFlush(function () {
          if (this.references === 0) {
            // Nope, either the autorun did not recreate the subscription, or there
            // were multiple calls to subscribeToMagazine() in the first place and not all
            // of them have been stopped. Go ahead and cancel.
            console.log("Canceling our subscription to", magazineName);
            delete subscriptions[magazineName];
          }
        });
      }
    };
```

```
  }) else {
    // We already have a subscription to this magazine running. Increment the reference
    // count to stop it from going away.
    subscriptions[magazineName].references++;
  }

  if (Tracker.active) {
    Tracker.onInvalidate(function (c) {
      // subscribeToMagazine was called from inside an autorun, and the autorun is
      // about to rerun itself. Tentatively plan to cancel the subscription. If the
      // autorun resubscribes to that same magazine when it is rerun, the logic in
      // in stop() is smart enough to leave the subscription alone rather than
      // canceling and immediately recreating it.
      //
      // (Tracker.onInvalidate is a shortcut for Tracker.currentComputation.onInvalidate.)
      if (_.has(subscriptions, magazineName)) {
        subscriptions[magazineName].stop();
      }
    });
  }

  return subscriptions[magazineName];
};
```

Suppose that the user's `currentInterest` is "fashion", meaning that they should be automatically subscribed to *Cosmo*. The autorun runs, calling `subscribeToMagazine()` with a `magazineName` of "Cosmo". `subscribeToMagazine()` checks the list of subscriptions to see if there is already a new subscription for *Cosmo*. There isn't, so it adds a new subscription to `subscriptions`.

In addition, `subscribeToMagazine()` sets up an `onInvalidate` callback that will run whenever a reactive value (such as `currentInterest`) inside the enclosing autorun changes, but before the autorun is scheduled to be rerun. Changing `currentInterest` from "fashion" to "fitness" triggers the `onInvalidate`callback, which calls the`stop()` method on all current subscriptions. Using `stop()` doesn't automatically delete the subscriptions, though, because `stop()` *waits to see* whether the user will be re-subscribed to the same magazines after the autorun is rerun.

`stop()` uses the `reference` property on a subscription to figure out whether or not the subscription should actually be stopped. Whenever a new subscription to a magazine is created, the `reference` property is set to 1. When `stop()` runs, it decrements `references` but doesn't delete the subscription just yet. Then, when the autorun gets rerun, `subscribeToMagazine` can reuse the previously created subscription instead of having to build one from scratch. If the user's interests change to "fitness" from "fashion" (meaning they should stay subscribed to

Cosmo), `subscribeToMagazine` finds the previously created *Cosmo* subscription and reinstates it by incrementing `references` back to 1.

But how do we actually stop a subscription? We can do so by setting up an `afterFlush()` callback in `stop()` to clean up any unnecessary subscriptions. If the user's `currentInterest` changes to "celebrities", the`reference` property of the *Cosmo* subscription is set to 0 by the `stop()` method. Then, the autorun reruns, calling `subscribeToMagazine`, which creates a subscription to *People* with a `reference` property of 1. Once the flush cycle is over (after all autoruns are rerun), the afterFlush() callback goes ahead and deletes any subscriptions that have a `reference` property of 0. After the autorun is rerun, a new *People* subscription is in the list but the *Cosmo* subscription is gone.

The Flush Cycle

When – precisely – do autoruns rerun? In other words, when do changes to reactive values actually take effect? In this code:

```
var data = new ReactiveDict;
data.set("favoriteFood", "chicken");

Tracker.autorun(function () {
  console.log(data.get("favoriteFood"));
});

console.log("start update");
data.set("favoriteFood", "waffles");
data.set("favoriteFood", "pie");
console.log("finish update");
```

will "waffles" and "pie" be printed before the word "finish", or after the word "finish"?

The answer is that, by default, all changes to reactive values are batched up and put into effect all at once, when all JavaScript code has finished executing and the system is idle. (For example, in a web browser, this might be after the browser has finished handling the current event.) This process of doing a batch of reactive updates is called *flushing*, and it's also possible to manually trigger it by calling `Tracker.flush()`.

So the example code above will print:

```
chicken
start update
finish update
waffles
pie
```

But if the example is changed by adding a call to `Tracker.flush()`:

```
var data = new ReactiveDict;
data.set("favoriteFood", "chicken");

Tracker.autorun(function () {
  console.log(data.get("favoriteFood"));
});

console.log("start update");
data.set("favoriteFood", "waffles");
data.set("favoriteFood", "pie");
Tracker.flush(); // ADDED LINE
console.log("finish update");
```

Then it will print:

```
chicken
start update
waffles
pie
finish update
```

The Reason for the Flush Cycle

Why have flushing? Why not do updates immediately, whenever any data changes?

Predictability The most important reason is *predictability*. Updates don't happen while your code is running, so you don't have to worry about data structures or the DOM changing out from under you. On the other hand, if you do want to see the results of your changes, you can call `Tracker.flush()` to trigger an update and it won't return until everything is fully updated.

Here's an example. Let's say that in response to a click event, an application reads data out of a form in a popup window, and uses that data to make a write to the database. Then it reads a second value from the form, and makes a second write to the database. Because reactive updates are deferred until after your code has finished executing, you can be certain that the first write to the database won't cause a reactive update that closes the popup. No updates happen until flush time, so the DOM is completely locked down and stable while your code is running.

On the other hand, suppose that in response to a click event, an application sets a reactive value that causes a popup window to open, and then from the same event handler it wants to use jQuery to find a particular DOM element in the popup window and focus it. In this case, after setting the reactive value that opens the popup, the application can simply call `Tracker.flush()`. When flush returns, all updates will have completed, meaning that the popup window will

have been rendered and put into the DOM, guaranteeing that it can be found and manipulated with jQuery.

So *not flushing* guarantees that nothing will change out from under you, while *flushing* guarantees that all updates have completed. No matter which kind of predictabliity you want, Tracker can provide it.

Consistency A second reason is *consistency*. Flushing moves the system from one consistent state to another. If several variables are being changed, the update won't happen until all of the variables have their new values.

Take this example, which tracks the bank balances of two users, Alice and Bob.

```
var balances = new ReactiveDict;
balances.set("alice", "2");
balances.set("bob", "1");

Tracker.autorun(function() {
  console.log("Alice:", balances.get("alice"), "Bob:", balances.get("bob"),
              "Total:", balances.get("alice") + balances.get("bob"));
});
// "Alice: 2 Bob: 1 Total: 3"
```

Here's a function that transfers money from Alice to Bob. The autorun doesn't trigger until all of the balances have been updated and a consistent snapshot can be read. We never do an update with an inconsistent state where the balances in the accounts don't add up to 3.

```
var transferMoney = function (amount) {
  bank.set("alice", bank.get("alice") - amount);
  bank.set("bob", bank.get("bob") + amount);
};

transferMoney(1);
// "Alice: 1 Bob: 2 Total: 3"
```

If we didn't want this consistency, and really wanted the updates to happen immediately, we could manually flush after each change.

```
bank.set("alice", bank.get("alice") + 1);
Tracker.flush();
// "Alice: 2 Bob: 2 Total: 4"
bank.set("bob", bank.get("bob") - 1);
Tracker.flush();

// "Alice: 2 Bob: 1 Total: 3"
```

Because a flush was manually performed before all of the account balances had been updated, we see the inconsistent state where the account balances

momentarily add up to 4 instead of 3.

Performance Finally, batching up the updates is simply more efficient. For example, we may want to re-render portions of a `<div>` depending on various reactive values. If several of the values change at once, then the screen should be updated just once, with the full set of new values. We wouldn't want to update the DOM over and and over again, once for each value that had changed. That would slow down the app and could potentially also cause flicker in the user interface.

Hooking Into the Flush Cycle

You can use `Tracker.afterFlush()` to run a callback function once at the completion of the next flush cycle (after all the autoruns have been re-run).

```
var data = new ReactiveDict;
data.set("favoriteFood", "cake");

Tracker.autorun(function () {
  console.log("My favorite food is " + data.get("favoriteFood") + "!");
});

var setUnpopularFood = function (what) {
  data.set("favoriteFood", what);
  Tracker.afterFlush(function () {
    console.log("Sounds gross to you, but from where I'm from it's considered a delicacy!");
  });
};
```

In this example, when `setUnpopularFood` is used – presumably to change the favorite food to something dubious – a message is printed defending the choice.

```
set("favoriteFood", "candy");
// "My favorite food is candy!"
setUnpopularFood("lizards");
// "My favorite food is lizards!"
// "Sounds gross to you, but from where I'm from it's considered a delicacy!"
set("favoriteFood", "ice cream");
// "My favorite food is ice cream!"
```

`Tracker.afterFlush` is one-shot. It runs the provided function once, at the end of the next flush cycle. It won't run again unless another call to `Tracker.afterFlush` is made. That's why the `afterFlush` handler doesn't run when "favoriteFood" is set to ice cream.

How the Flush Cycle Works

Every autorun (represented by a `Tracker.Computation` object) has a flag, `invalidated`, that indicates whether it needs to be rerun. When a reactive value changes, this flag is set to true on all affected `Computation`s, meaning that they need to be rerun during the next flush. When a flush is triggered, either by a call to `Tracker.flush()` or by the system being idle, all of the invalidated `Computation`s are rerun and their `invalidated` flag is cleared. Finally, at the end of the flush, any `afterFlush` handlers are run.

The following guarantees are made:

- When `Tracker.flush()` returns, everything will have been fully updated. Every autorun affected by a change will have finished rerunning. The autoruns are not guaranteed to be rerun in any particular order. This guarantee applies even if reactive values are changed during the flush cycle.
- When `Tracker.flush()` returns, any `afterFlush` handlers will have been run, even if they were not registered until after flushing started. `afterFlush` handlers will not be called until there are no more autoruns that need to be rerun. `afterFlush` handlers will run in the order they were registered.

These guarantees apply even if reactive values change during the flush cycle, or if additional afterFlush handlers are registered during the flush cycle. Some implications of this:

- At times, rerunning an autorun may change a reactive value. When that happens, the autoruns that depend on that reactive value are rerun as part of the same flush cycle.
- **You can't call `flush()` from inside an autorun**, or else you'll get stuck in an infinite loop. `flush()` only returns when all of the autoruns have returned, but the autorun can't return until `flush()` returns!
- If an `afterFlush` callback changes a reactive value, all of the autoruns that depend on that reactive value get rerun before any of the other `afterFlush` callbacks are executed.
- If an `afterFlush` handler registers an `afterFlush` handler, it will be run as part of the current flush cycle.

Here's an example that illustrates some of these guarantees . If the balance of the checking account goes below zero, an "insufficient funds" message will be printed and check writing will be disabled. However, there is an autorun that provides overdraft protection, transferring $25 from savings to checking whenever the checking account is empty.

```
var bank = new ReactiveDict;
bank.set("checking", 10);
bank.set("savings", 50);
bank.set("checkWritingAllowed", true);
```

```
Tracker.autorun(function () {
  console.log("There is $" + bank.get("checking") + " in your checking account.");

  Tracker.afterFlush(function () {
    // Since this is inside afterFlush, it only runs after every autorun affected
    // by the most recent update has finished rerunning. So the overdraft protection
    // code below will run before this check happens.
    if (bank.get("checking") < 0) {
      console.log("Insufficient funds! No more checks for you!");
      bank.set("checkWritingAllowed", false);
    }
  });
});

Tracker.autorun(function () {
  if (bank.get("checking") < 0 &&
      bank.get("savings") >= 25) {
    bank.set("checking", bank.get("checking") + 25);
    bank.set("savings", bank.get("savings") - 25);
    console.log("Automatically transferred $25 from savings to checking.");
  }
});

Tracker.autorun(function () {
 if (bank.get("checkWritingAllowed"))
   console.log("Go ahead, write some checks!");
 else
   console.log("Your check writing privileges have been suspended!");
});

var writeACheck = function (amount) {
  if (bank.get("checkWritingAllowed"))
    bank.set("checking", bank.get("checking") - amount);
};

// "There is $10 in your checking account."
// "Go ahead, write some checks!"
writeACheck(5);
// "There is $5 in your checking account."
writeACheck(20);
// "There is $-15 in your checking account."
// "Automatically transferred $25 from savings to checking."
// "There is $10 in your checking account."
writeACheck(30);
// "There is $-20 in your checking account."
// "Automatically transferred $25 from savings to checking."
```

```
// "There is $5 in your checking account."
writeACheck(15);
// "There is $-10 in your checking account."
// "Insufficient funds! No more checks for you!"
// "Your check writing privileges have been suspended!"
```

Because `Tracker.afterFlush` is used to do the insufficient funds check after all autoruns have finished running, the insufficient funds message will only be printed if the overdraft protection wasn't able to make the checking account balance positive again. (Because the call to `Tracker.afterFlush` is inside an autorun, the insufficient funds check happens every time the checking account balance changes, not just once.)

When the insufficient funds check fails, the "checkWritingAllowed" variable is set from inside the `Tracker.afterFlush` handler. This causes one more autorun to rerun before the flush cycle is complete, even though `Tracker.afterFlush` handlers had started to run.

Avoiding Change Loops

Just as carelessly written recursive code can recurse forever, carelessly written reactive code can get `Tracker.flush` stuck in an infinite loop. This is easy to avoid by following two simple rules.

Rule #1: Don't Call Tracker.changed() Unless Something Has Actually Changed If you are creating your own new type of reactive value, don't call `changed()` unless your reactive value has actually changed! (ReactiveDict and all of the standard reactive data sources handle this automatically for you.)

```
// WRONG
var firstVar = 0;
var firstDep = new Tracker.Dependency();
var secondVar = 0;
var secondDep = new Tracker.Dependency();

var setFirstVar = function(newValue) {
  firstVar = newValue;
  firstDep.changed();
};

var setSecondVar = function(newValue) {
  secondVar = newValue;
  secondDep.changed();
};

Tracker.autorun(function () {
  secondDep.depend();
```

```
  firstVar = secondVar;
  firstDep.changed();
});

Tracker.autorun(function () {
  firstDep.depend();
  secondVar = firstVar;
  secondDep.changed();
});
```

In the example above, the first autorun sets `firstVar` to `secondVar` without checking to see if the two are different. If `secondVar` changes, the first autorun is rerun during flush, updating `firstVar`. This sets off the second autorun, which similarly sets `secondVar` to `firstVar` without confirming that they are different. Because `secondVar` was changed, the first autorun runs again, resulting in an infinite loop.

This is easily fixed with two more lines of code:

```
// RIGHT
var firstVar = 0;
var firstDep = new Tracker.Dependency();
var secondVar = 0;
var secondDep = new Tracker.Dependency();

var setFirstVar = function(newValue) {
  firstVar = newValue;
  firstDep.changed();
};

var setSecondVar = function(newValue) {
  secondVar = newValue;
  secondDep.changed();
};

var handle1 = Tracker.autorun(function () {
  secondDep.depend();
  if (firstVar !== secondVar) { // ADDED LINE
    firstVar = secondVar;
    firstDep.changed();
  }
});

var handle2 = Tracker.autorun(function () {
  firstDep.depend();
  if (secondVar !== firstVar) { // ADDED LINE
    secondVar = firstVar;
```

```
    secondDep.changed();
  }
});
```

In the example above, `firstVar` is only changed if it's different from `secondVar`, and vice versa. When `secondVar` changes, and `firstVar` is different from `secondVar`, the first autorun will set `firstVar` to be equal to `secondVar`. The change to `firstVar` sets off the second autorun, but this time it doesn't change the value of `secondVar` because it's equivalent to `firstVar`.

Rule #2: Don't Create Any Circular References Beware of *circular references* — situations where a reactive value (directly or indirectly) refers back to itself, resulting in a contradiction.

```
var scores = new ReactiveDict;

Tracker.autorun(function () {
  scores.set("a", scores.get("b") + 1);
});

Tracker.autorun(function () {
  scores.set("b", scores.get("a") + 1);
});

Tracker.flush();
```

In this example, there are no possible values for "a" and "b" that will satisfy both autoruns! "a" wants to be higher than "b", but "b" wants to be higher than "a"! There's no way that both of these can be true simultaneously, so the system will never rest.

The first autorun uses "b" to set the value of "a"; therefore, it updates "a" whenever the value of "b" changes. When "a" changes, though, the second autorun reruns, changing the value of "b" based on "a". This forces the first autorun to rerun, creating an infinite loop. `Tracker.flush()` will never return because the flush will never complete.

`Tracker.flush` guarantees that, when it returns, all recomputed reactive values are consistent with one another. But there is no way to assign consistent values to "a" and "b" that satisfy both autoruns, so `Tracker.flush` can never return.

You can avoid this problem by avoiding circular references. If "a" depends on "b", don't have "b" depend on "a".