Meteor has a simple dependency tracking system which allows it to automatically rerun templates and other computations whenever `Session` variables, database queries, and other data sources change.

Unlike most other systems, you don't have to manually declare these dependencies — it "just works". The mechanism is simple and efficient. When you call a function that supports reactive updates (such as a database query), it automatically saves the current Computation object, if any (representing, for example, the current template being rendered). Later, when the data changes, the function can "invalidate" the Computation, causing it to rerun (rerendering the template).

Applications will find `Tracker.autorun` useful, while more advanced facilities such as `Tracker.Dependency` and `onInvalidate` callbacks are intended primarily for package authors implementing new reactive data sources.

{% apibox "Tracker.autorun" %}

`Tracker.autorun` allows you to run a function that depends on reactive data sources, in such a way that if there are changes to the data later, the function will be rerun.

For example, you can monitor a cursor (which is a reactive data source) and aggregate it into a session variable:

```
Tracker.autorun(() => {
  const oldest = _.max(Monkeys.find().fetch(), (monkey) => {
    return monkey.age;
  });

  if (oldest) {
    Session.set('oldest', oldest.name);
  }
});
```

Or you can wait for a session variable to have a certain value, and do something the first time it does, calling `stop` on the computation to prevent further rerunning:

```
Tracker.autorun((computation) => {
  if (!Session.equals('shouldAlert', true)) {
    return;
  }

  computation.stop();
  alert('Oh no!');
});
```

The function is invoked immediately, at which point it may alert and stop right away if `shouldAlert` is already true. If not, the function is run again when `shouldAlert` becomes true.

A change to a data dependency does not cause an immediate rerun, but rather "invalidates" the computation, causing it to rerun the next time a flush occurs. A flush will occur automatically as soon as the system is idle if there are invalidated computations. You can also use `Tracker.flush` to cause an immediate flush of all pending reruns.

If you nest calls to `Tracker.autorun`, then when the outer call stops or reruns, the inner call will stop automatically. Subscriptions and observers are also automatically stopped when used as part of a computation that is

rerun, allowing new ones to be established. See `Meteor.subscribe` for more information about subscriptions and reactivity.

If the initial run of an autorun throws an exception, the computation is automatically stopped and won't be rerun.

Tracker.autorun and async callbacks

`Tracker.autorun` can accept an `async` callback function. However, the async call back function will only be dependent on reactive functions called prior to any called functions that return a promise.

Example 1 - autorun `example1()` **is not** dependent on reactive changes to the `Meteor.users` collection. Because it is dependent on nothing reactive it will run only once:

```
Tracker.autorun(async function example1() {
  let asyncData = await  asyncDataFunction();
  let users = Meteor.users.find({}).fetch();
});
```

However, simply changing the order so there are no `async` calls prior to the reactive call to `Meteor.users.find`, will make the async autorun `example2()` dependent on reactive changes to the `Meteor.users` collection.

Example 2 - autorun `example2()` **is** dependent on reactive changes to the Meteor.users collection. Changes to the `Meteor.users` collection will cause a rerun of `example2()` :

```
Tracker.autorun(async function example2()  {
  let users = Meteor.users.find({}).fetch();
  let asyncData = await  asyncDataFunction();
});
```

{% apibox "Tracker.flush" %}

Normally, when you make changes (like writing to the database), their impact (like updating the DOM) is delayed until the system is idle. This keeps things predictable — you can know that the DOM won't go changing out from under your code as it runs. It's also one of the things that makes Meteor fast.

`Tracker.flush` forces all of the pending reactive updates to complete. For example, if an event handler changes a Session variable that will cause part of the user interface to rerender, the handler can call `flush` to perform the rerender immediately and then access the resulting DOM.

An automatic flush occurs whenever the system is idle which performs exactly the same work as `Tracker.flush`. The flushing process consists of rerunning any invalidated computations. If additional invalidations happen while flushing, they are processed as part of the same flush until there is no more work to be done. Callbacks registered with `Tracker.afterFlush` are called after processing outstanding invalidations.

It is illegal to call `flush` from inside a `flush` or from a running computation.

The Tracker manual describes the motivation for the flush cycle and the guarantees made by `Tracker.flush` and `Tracker.afterFlush`.

{% apibox "Tracker.nonreactive" %}

Calls `func` with `Tracker.currentComputation` temporarily set to `null` and returns `func`'s own return value. If `func` accesses reactive data sources, these data sources will never cause a rerun of the enclosing computation.

{% apibox "Tracker.active" %}

This value is useful for data source implementations to determine whether they are being accessed reactively or not.

{% apibox "Tracker.inFlush" %}

This value indicates, whether a flush is in progress or not.

{% apibox "Tracker.currentComputation" %}

It's very rare to need to access `currentComputation` directly. The current computation is used implicitly by `Tracker.active` (which tests whether there is one), `dependency.depend()` (which registers that it depends on a dependency), and `Tracker.onInvalidate` (which registers a callback with it).

{% apibox "Tracker.onInvalidate" %}

See *`computation`* `.onInvalidate` for more details.

{% apibox "Tracker.afterFlush" %}

Functions scheduled by multiple calls to `afterFlush` are guaranteed to run in the order that `afterFlush` was called. Functions are guaranteed to be called at a time when there are no invalidated computations that need rerunning. This means that if an `afterFlush` function invalidates a computation, that computation will be rerun before any other `afterFlush` functions are called.

Tracker.Computation

A Computation object represents code that is repeatedly rerun in response to reactive data changes. Computations don't have return values; they just perform actions, such as rerendering a template on the screen. Computations are created using `Tracker.autorun`. Use `stop` to prevent further rerunning of a computation.

Each time a computation runs, it may access various reactive data sources that serve as inputs to the computation, which are called its dependencies. At some future time, one of these dependencies may trigger the computation to be rerun by invalidating it. When this happens, the dependencies are cleared, and the computation is scheduled to be rerun at flush time.

The *current computation* (`Tracker.currentComputation`) is the computation that is currently being run or rerun (computed), and the one that gains a dependency when a reactive data source is accessed. Data sources are responsible for tracking these dependencies using `Tracker.Dependency` objects.

Invalidating a computation sets its `invalidated` property to true and immediately calls all of the computation's `onInvalidate` callbacks. When a flush occurs, if the computation has been invalidated and not stopped, then the computation is rerun by setting the `invalidated` property to `false` and calling the original function that was passed to `Tracker.autorun`. A flush will occur when the current code finishes running, or sooner if `Tracker.flush` is called.

Stopping a computation invalidates it (if it is valid) for the purpose of calling callbacks, but ensures that it will never be rerun.

Example:

```
// If we're in a computation, then perform some clean-up when the current
// computation is invalidated (rerun or stopped).
if (Tracker.active) {
  Tracker.onInvalidate(() => {
    x.destroy();
    y.finalize();
  });
}
```

{% apibox "Tracker.Computation#stop" %}

Stopping a computation is irreversible and guarantees that it will never be rerun. You can stop a computation at any time, including from the computation's own run function. Stopping a computation that is already stopped has no effect.

Stopping a computation causes its `onInvalidate` callbacks to run immediately if it is not currently invalidated, as well as its `stop` callbacks.

Nested computations are stopped automatically when their enclosing computation is rerun.

{% apibox "Tracker.Computation#invalidate" %}

Invalidating a computation marks it to be rerun at flush time, at which point the computation becomes valid again. It is rare to invalidate a computation manually, because reactive data sources invalidate their calling computations when they change. Reactive data sources in turn perform this invalidation using one or more `Tracker.Dependency` objects.

Invalidating a computation immediately calls all `onInvalidate` callbacks registered on it. Invalidating a computation that is currently invalidated or is stopped has no effect. A computation can invalidate itself, but if it continues to do so indefinitely, the result will be an infinite loop.

{% apibox "Tracker.Computation#onInvalidate" %}

`onInvalidate` registers a one-time callback that either fires immediately or as soon as the computation is next invalidated or stopped. It is used by reactive data sources to clean up resources or break dependencies when a computation is rerun or stopped.

To get a callback after a computation has been recomputed, you can call `Tracker.afterFlush` from `onInvalidate`.

{% apibox "Tracker.Computation#onStop" %}

{% apibox "Tracker.Computation#stopped" %}

{% apibox "Tracker.Computation#invalidated" %}

This property is initially false. It is set to true by `stop()` and `invalidate()`. It is reset to false when the computation is recomputed at flush time.

{% apibox "Tracker.Computation#firstRun" %}

This property is a convenience to support the common pattern where a computation has logic specific to the first run.

Tracker.Dependency

A Dependency represents an atomic unit of reactive data that a computation might depend on. Reactive data sources such as Session or Minimongo internally create different Dependency objects for different pieces of data, each of which may be depended on by multiple computations. When the data changes, the computations are invalidated.

Dependencies don't store data, they just track the set of computations to invalidate if something changes. Typically, a data value will be accompanied by a Dependency object that tracks the computations that depend on it, as in this example:

```
let weather = 'sunny';
const weatherDep = new Tracker.Dependency();

function getWeather() {
  weatherDep.depend();
  return weather;
}

function setWeather(newWeather) {
  weather = newWeather;

  // Note: We could add logic here to only call `changed` if the new value is
  // different from the old value.
  weatherDep.changed();
}
```

This example implements a weather data source with a simple getter and setter. The getter records that the current computation depends on the `weatherDep` dependency using `depend()`, while the setter signals the dependency to invalidate all dependent computations by calling `changed()`.

The reason Dependencies do not store data themselves is that it can be useful to associate multiple Dependencies with the same piece of data. For example, one Dependency might represent the result of a database query, while another might represent just the number of documents in the result. A Dependency could represent whether the weather is sunny or not, or whether the temperature is above freezing. `Session.equals` is implemented this way for efficiency. When you call `Session.equals('weather', 'sunny')`, the current computation is made to depend on an internal Dependency that does not change if the weather goes from, say, `rainy` to `cloudy`.

Conceptually, the only two things a Dependency can do are gain a dependent and change.

A Dependency's dependent computations are always valid (they have `invalidated === false`). If a dependent is invalidated at any time, either by the Dependency itself or some other way, it is immediately removed.

See the Tracker manual to learn how to create a reactive data source using `Tracker.Dependency`.

{% apibox "Tracker.Dependency#changed" %}

{% apibox "Tracker.Dependency#depend" %}

`dep.depend()` is used in reactive data source implementations to record the fact that `dep` is being accessed from the current computation.

{% apibox "Tracker.Dependency#hasDependents" %}

For reactive data sources that create many internal Dependencies, this function is useful to determine whether a particular Dependency is still tracking any dependency relationships or if it can be cleaned up to save memory.