# Tracker.autorun()

1. Creates a new computation.
   1. Tracker.currentComputation becomes parent
   2. When Tracker.onInvalidate() then computation is stopped

# Tracker.flush()

1. Calls \_recompute on all pendingComputations
2. Calls afterFlush callbacks

# Computation Constructor

1. Calls \_compute()

# Computation.\_compute()

1. sets the currentComputation to the current computation

2. sets invalidated = false

3. executes the function associated with the computation

4. sets the currentComputation back to the previous computation

# Computation.\_recompute

1. Calls \_compute()

# Computation.Invalidate()

1. Calls requireFlush()
   1. Calls Tracker.flush with setTimeout(0)
2. Adds self to pendingComputations
3. Calls all \_onInvalidateCallbacks

# Dependency Constructor

1. Creates \_dependentsById collection

# Dependency.depend()

1. Takes id of current computation and adds to \_dependentsById
2. When computation is invalidated, removes id from \_dependentsById

# Dependency.changed()

1. Calls invalidate() on each computation in \_dependentsByid

1. Calling Tracker.autorun()
   1. creates a new Computation object
   2. executes the function
2. Calling Dependency.depends()
   1. adds id of current computation to \_dependentsById collection of Dependency object
3. Calling Dependency.change()
   1. calls invalidate() on each computation in \_dependentsById
4. Calling Computation.invalidate()
   1. Adds self to pendingComputations
   2. Sets up Tracker.flush() to get called on system idle
5. Tracker.flush()
   1. Calls autorun function associated with each computation in pendingComputations

When a computation is invalidated, it removes itself from all dependencies

If you call dependency.changed() multiple times, then it counts only once (calling setTemp() over and over only matters for the last one)

retrieving data is the same as subscribing to change notifications for that data.

Transparent reactivity collapses retrieving data and getting a change notification for that data into a single function call.

# Slide 0

Agenda

Why Reactive Programming?

Deep Dive in Meteor Reactive Programming?

Reactive-Dict, Reactive-var, Minongo

# Slide 1

What is Meteor?

* Javascript Everywhere
* Database Everywhere
* Run Everywhere
* Real-time
* Reactive

# Slide 2

Real-time Twitter Clone

# Slide 3

Animated picture 🡺 changes by one browser get propagated from the server to all browsers through the magic of web sockets

Browser browser browser

Server

# Slide 4

Transparent Reactive Programming

= Alternative to Event-Driven Programming

that enables you to do more with less code

# Slide 5

Messages Change => Render List of Messages

# Slide 6

Event-driven Approach:

2 steps:

1. Create a function that retrieves data and renders data to UI
2. Wire-up function to change event on messages cursor

# Slide 7

Reactive Approach:

1 step

1. Create a function that retrieves data and renders data to UI

Retrieving data is the same as subscribing to change notifications for that data.

# Slide 8

*Transparent* Reactive Programming

Reactive Data Source => App Code => Reactive Consumer

You don’t have to do anything special in your code, using magic reactive data sources just works….

# Slide 9

var spaceship = {

\_velocity: 0,

\_listeners: [],

getVelocity: function() {

return \_velocity;

},

setVelocity: function(value) {

if (value != \_velocity) {

\_velocity = value;

changed();

}

},

changed: function() {

for (var i=0;i<\_listeners.length;i++) {

listeners[i]();

}

},

onChanged: function(func) {

\_listeners.push(func);

}

};

// update dashboard when velocity changes

spacehip.onChanged(function() {

var velocity = spaceship.getVelocity();

// Update dashboard gauge

});

# Slide 10

var spaceship = {

\_velocity: 0,

\_dependency: new Tracker.Dependency,

getVelocity: function() {

\_dependency.depend();

return \_velocity;

},

setVelocity: function(value) {

if (value != \_velocity) {

\_velocity = value;

\_dependency.changed();

}

},

};

Tracker.autorun(function() {

var velocity = spaceship.getVelocity();

// update dashboard gauge

});

# Slide 11

Tracker.autorun(function() {

…. App code

})

Reruns a function automatically when reactive data sources in

Function change.

# Slide 12

Tracker.Dependency.depend()

Adds the current computation to a collection of computations

# Slide 13

Tracker.Dependency.change()

Invalidates any computations associated with this dependency. When system is next idle, any functions associated with the computations are rerun.

# Slide 14

Typically, you do not use Tracker.Dependency.depend() or Tracker.Dependency.change() directly. Instead, use one of the following reactive data sources:

1. Session
2. Reactive-dict
3. Reactive-Var
4. Mongo Cursor

# Slide 15

Typically, you do not use Tracker.autorun. All functions use with Blaze use autorun automatically.

Show spaceship code that uses Blaze

# Slide 16

Using Session

# Slide 17

Using Reactive var

# Slide 18

Using mini-mongo

# Slide 19

Batching updates

# Slide 20

Nested autoruns

# Slide 21

Summary