

EventMachine

Chapter One: Introduction

It's PeepCode!

Today you will learn to use EventMachine, the event-driven client and server toolkit. The content for this screencast was written by Dan Sinclair of PostRank.

PostRank uses EventMachine to power most of their backend processing work. This includes API servers which handle more than 10 million API calls per day. It also powers PostRank's feed processing systems, and their metric consumers which process tens of thousands of metrics per minute. It's all very high performance and it's done with Ruby using EventMachine.

Over the next hour you'll learn the basics of EventMachine: what it is, how to use its API, and where you'd use specific features.

So let's start! What is EventMachine and why would you want to use it?

EventMachine is a toolkit for writing evented applications. Let's pull that sentence apart.

EventMachine is a toolkit. The toolkit was built to handle the c10k problem, handling 10 thousand concurrent connections; that's serving 10 thousand clients at once from a single machine. These types of problems have many similarities including concurrency, threading, processes and networking. EventMachine provides a well-designed API with which you can do these things, but in order to use it well, you'll still have to think about both lower level threading issues and higher level design and code flow.

EventMachine is evented. It uses a design pattern called the *reactor pattern*. With a non-evented server, you'll make a request, wait for the response, and continue. Code is executed line by line. A line of code isn't executed until the previous line has completed its work.

With an evented program you'll make a request, but, instead of waiting for the response, your program can go do other work. Once the request has finished, an event will fire and you can finish processing the request.

How does it work? EventMachine runs an internal loop called the "run loop" which listens for data from sockets, keyboards, files, the network or wherever, and will make calls to your code when events happen.

This can be very powerful and efficient, but requires that you design your program differently than you are used to. You can now have several requests all happening at the same time. The order in which requests finish is, essentially, random. So, you don't have any guarantee on the ordering of events in the system. Your code will be concerned with

EventMachine

setting up handlers to work with data once it's available but should make as few assumptions as possible about when that will happen.

A main feature of evented servers is the *reactor*.

The reactor is the application's main thread and is responsible for handling all the low level network sockets, sending data, receiving data, and creating and tearing down connections. Along with sockets, it can deal with things like reading and writing files and reading keyboard data. Your program will tell the reactor about your application's code and the reactor will figure out when to run each part of it.

EventMachine provides APIs for things like networking, such as sending and receiving data through TCP or UDP. It provides SSL capabilities, file watching, and file access capabilities.

When you're working with EventMachine, all of your code will be asynchronous. Instead of telling the computer when you want to run a bit of code, it tells you when it's ready to run it. So EventMachine applications are built around callbacks and errbacks to handle information being received and sent through the system.

EventMachine can be used to write both clients and servers, so you can serve data yourself or you can use things like `EM::HttpRequest` to request data from external services.

There are many libraries currently available for EventMachine. They work with services like Postgres, MySQL, AMQP, MemCache, Cassandra, and the Beanstalk message queue. Other synchronous libraries will slow your application down, so you'll need to use software that was written using EventMachine's asynchronous API.

Why would you want to use EventMachine? It's written to be high performance. It's designed to handle thousands of clients connected into the same server at the same time, and to efficiently process all the incoming and outgoing data. There are mechanisms to do fast file reading or streaming off disk. If you need to write high performance client/server implementations, EventMachine definitely deserves a look.

I think EventMachine is under-appreciated by Ruby developers. It's as powerful as popular platforms like Node.js. And now that many browsers support WebSockets, you can write real-time applications that weren't even possible before. I think real-time data delivery is at the heart of an upcoming wave of web applications. And if you like Ruby, you can write these applications in your language of choice without needing to learn the intricacies of JavaScript.

That being said, this is hard stuff. So here's how we're going to tackle it.

EventMachine

We'll start by looking at isolated features of EventMachine. It will be a tour through the toolbox with simple but working examples for each. The examples may seem a bit impractical. We've intentionally crafted them to be as simple as possible so you can establish a solid grip on the concepts and problems solved by the tools that ship with EventMachine.

Then we'll put the ideas together into a functioning client/server application that uses WebSockets to communicate with a web browser. We'll look at a new feature of Ruby 1.9 that makes evented programming a bit easier. Then we'll look at Goliath, a new evented server used in production at PostRank.

More than other PeepCode screencasts, this is one that you'll want to watch a few times. We highly recommend watching it straight through the first time to understand the concepts, then watching a second time if you want to follow along with the code samples.

Speaking as Geoffrey Grosenbach for a moment, I started out completely confused but ended up thoroughly energized about the kinds of applications that can be written with EventMachine. I hope it will do the same for you.

Chapter Two: A Simple Echo Server

- Write a simple server that sends back any information it receives
- Write a client that sends data to the server

To get started, let's write a simple echo server and client written in EventMachine. Full-featured EventMachine applications can become quite complicated, so we'll start by learning the parts of an evented server with the simplest application possible.

We'll write a server that receives messages and sends them straight back to the client.

A separate client will start up, connect to the server, and send a single message. The client will receive the response from the server, print it to the screen, and immediately shut down.

The first thing you'll need to do to get started is to install EventMachine. It works on Ruby 1.8 or Ruby 1.9. I'm using Ruby 1.9 in these examples.

To install EventMachine, run "gem install eventmachine". You'll probably want to do "--pre" if the 1.0 release isn't out yet. You can see the version number after installation or by running "gem search -r eventmachine" to see the current stable version on the RubyGems servers. Pre is stable and it's been in use at PostRank for quite a long time. It

EventMachine

fixes some bugs and provides a few improvements that you'll probably want to use in your software.

Let's get started by writing our echo server. We start with a simple shebang line, and we'll require RubyGems and EventMachine to make them available to the application.

When EventMachine runs, everything happens within the `EM.run` loop. This loop will not return until `EM.stop` is called.

It can't be emphasized enough: everything happens inside this loop and nothing after it will execute. Everything inside this loop will be evented.

You shouldn't do anything that will block the reactor. Infinite loops like `"while(1)"` or `"while(true)"` will block; don't do that. Long running operations will block. And calls to `"sleep"` will block. Traditional network calls that don't use EventMachine will block.

Much more than with single-threaded web applications, you must send all long-running tasks to a separate job queue or use techniques that we'll mention in the next chapter.

In order to start the server, run `EM.start_server`. Bind it to any interface, and run it on port 9000. The third parameter is the Ruby class that will process connections for this server. Finally, print the message "Server running on port 9000."

Next, implement the connection handling code for the server. Even though I said that all code must live in the `EM.run` block, it doesn't have to be written within the `EM.run` block. In fact, it's best to write as little code as possible within `EM.run`.

We'll organize it into a Ruby class: `"class EchoServer"`. That class should inherit from `EM::Connection`. Both clients and servers should inherit from `EM::Connection`. Most things you do in EventMachine will end up inheriting from `EM::Connection` in some way or another.

Inside the server we're going to implement a few methods that will be called by EventMachine. `"post_init"` will be executed immediately after a connection is established to our server but before anything else in the run loop has happened. This allows you to set up instance variables or do other initialization for your class.

In our case, we're going to print a message stating that a client is connecting.

The opposite of `"post_init"` is `"unbind"`. This will fire when the connection to the client has closed. It could be triggered by other parts of our own server code when they call `"close_connection"`. Or it could be triggered by the remote client when it disconnects. Again, we're going to print a simple message `"client disconnecting."`

EventMachine

The third function we're going to implement for our server is "receive_data". This is where the real action happens. It receives a "data" parameter. This method will be called whenever data is received by the server. This is unbuffered by EventMachine so if you expect to receive large payloads, you will need to buffer it in some fashion; assemble smaller chunks into a full file, entire document, or complete packet.

For our example, we don't need to buffer because our data packet is going to be small. Print out that we received data from the client.

Finally, use "send_data" to send the same data back to the client, which is the definition of an echo server.

That is the entire echo server.

TODO: Insert terminal example with telnet

Let's try it as-is. Run "ruby server.rb" and you should see that it has started on port 9000. Open another terminal and use telnet to connect to 0.0.0.0 port 9000. Anything you type will be echoed back to you.

To exit, go back to the server tab and hit Ctrl-c.

CLIENT

With the server written we can now write the other side, an EventMachine client.

As with the server, use RubyGems to "require 'eventmachine'". We'll again need our EM.run block. But instead of calling "server" we're going to call "connect". Clients connect to a server. We want to connect to localhost and port 9000 because that's where the server is.

In a similar fashion, we'll write an EchoClient class. There's also a third parameter which will be passed to the "initialize" method of the EchoClient. I'll pass the first element from the command line arguments, ARGV[0].

Let's implement the EchoClient class. Again we'll inherit from EM::Connection. As I mentioned, we'll implement an "initialize" method which takes a single argument: the user. Store it in an instance variable.

As on the server we have a "post_init" method which is triggered when a connection is established. As soon as the client connection is made, we're going to "send_data" to the server. We'll send "hello from user".

And as with the server example, we can implement the "unbind" method on the client. Print the message "disconnected."

EventMachine

There is also a “receive_data” method. Print the data that was received.

Here’s something different. Servers should run forever, but this client will close the connection to the server once it receives a response. Call EM.stop. This ends the run loop for the client process and exits. That is our echo client.

Let’s run it!

Switch over to the terminal. Open two terminals in this directory. Start the server first. We can see it comes up on port 9000. Start the client on a separate window. This will happen quickly, but I’ll explain it. Remember that the code passes the first command line argument as a username. The client starts up. You can see that it connects to the server which receives “Hello from Dan” from the client.

The client received back “Hello from Dan” from the server. The client disconnects immediately after. You can see the client stops but the server is still running.

Do it as many more times as you wish. Connect to the server again and you’ll see the same series of messages. Pass a different username to see that it’s happening dynamically.

GOTCHAS

Even with this simple example there are a couple of quick gotchas to look out for.

If instead of running the client we attempt to run the server again, we would end up with an ugly error message that tells us that there is a no acceptor, the port is in use or requires user privileges. Usually this means that we’re already running the server elsewhere. If you really did want to restart the server, find the terminal or process that’s running it and kill it. You’ll be able to restart with a new server.

Another common error with a cryptic message is the spelling error. Imagine that you’ve spelled “receive_data” incorrectly or not implemented the “receive_data” method in your class. You’ll see a series of dots, some arrows, and then a number. This is the default EventMachine implementation of “receive_data”. If you see that, go implement “receive_data” or fix your spelling mistake.

You’ve now experienced EventMachine! With a basic server and client out of the way, we’ll take a more in-depth look at EventMachine’s major features. We’ll look at scheduling code to run on the main reactor. We’ll look at the toolkit of timers, ticks, threads, queues, and channels.

EventMachine

Chapter Three: Primitives

- The EventMachine toolkit
- Run Loop
- Next Tick
- Threads and EM.defer
- Queue
- Channel

There are five basic API tools in the EventMachine toolkit, called primitives. Jumping straight into a full application would be confusing, so in this chapter we'll look at each tool in isolation. The examples will focus mainly on printing text to the terminal so you can see what happens in what order. We'll mention real-world examples where possible. In subsequent chapters we'll write a practical application.

We'll start by looking at the run loop and explaining a few features you've seen briefly such as EM.stop. We'll look at scheduling tasks on the next tick of the reactor, timers, deferring to background threads, and a few other goodies along the way.

Every EventMachine application is built around the EM.run loop. You'll notice I'm using the two characters "EM" in most cases. It's a shortcut used by most experienced EventMachine developers. You could also spell out the word "EventMachine". Anywhere I use EM you could use "EventMachine". This works for inheritance as well. "EventMachine::Connection" is equivalent to EM::Connection. Use whichever you prefer. My personal preference is to use EM because I find it shorter and easier to read.

BASIC RUN LOOP

EM.run starts the *run loop*. The EM run loop is also called the reactor. Everything you do happens inside this block, even if you call code that's written elsewhere in the file or in other files. You'll create and start servers or start any connections to other servers. It all happens inside the EM.run block.

In fact, this block will not finish until you've stopped the reactor with EM.stop. This means that any code used by this loop needs to be implemented higher up in the file or required from other files before the EM.run block.

Let's put in a few print statements so we can see what's going on.

EventMachine

Call `EM.stop`. I could have also used `EM.stop_event_loop`. They're the same; `EM.stop` is shorter. `EM.stop` doesn't *immediately* stop the event loop. This would cause any open connections to be closed, and clients would be abandoned with incomplete data. Instead, all the connection termination callbacks and unbinds will be called so each class can clean up after itself. Finally, the run loop is stopped and usually, the program exits. So it's not an immediate stop but it's a signal to start shutting down the reactor and finish serving all requests.

Once everything has been stopped, this block will finish and we'll see the "done" message.

Running it shows "start, before, after, done".

There's no real-world example to mention here because *every* EventMachine application works this way.

NEXT TICK

That's interesting but not entirely useful. Let's take a look at some of the ways we can schedule work to happen in the future. Often you'll want to tell EventMachine to do something over and over every few seconds or at a certain point in time or even just on the next iteration of the run loop.

The first tool for running code at different times is `EM.next_tick`.

Let's look at this in a few different ways.

`next_tick` takes a block of code and schedules it to run in the next iteration of the reactor, on the main reactor thread. This can be used to postpone work so it happens a few milliseconds from now.

I'm going to use a few water analogies to explain how parts of EventMachine work. In this case, consider a flowing river and a water wheel. Calling "next_tick" is like putting a plank on the water wheel. The next time the wheel rolls around, it will hit the water and "do work". In this case, the plank is your code. It will be run at the next opportunity.

Even though EventMachine code should be considered to be asynchronous, the main reactor is single-threaded. `next_tick` allows the reactor to turn over one iteration. It will read any network data, send buffered data, handle firing timers, etc. If you're doing a lot of work on the main reactor loop, it can stop other code from running. Calling `next_tick` is one way to split code into smaller chunks. Otherwise, long-running tasks can cause your timers to drift since they won't get a chance to execute until the reactor iterates.

EventMachine

As an analogy, imagine that one plank causes the water wheel to slow down. The other planks won't see the water for a while.

Another use for `next_tick` is to bring code back to the main reactor thread. If code is running on a background thread, it can call `next_tick` and that code will run on the main thread. Code in `next_tick` always runs on the main thread, no matter where it was called from.

There are other ways to schedule code to run later. An easier technique to understand is the timer.

`“EM.add_timer(10)”` adds a timer that will fire in ten seconds. This is a single shot timer; it will fire only once. For repeating timers, there is `“add_periodic_timer(1)”` which will fire every second.

Here's a nuance that's important to understand. Periodic timers are rescheduled after the block executes. So the actual occurrence of the block will be 1 second plus the time required to execute the block. Assume a worst case scenario where the code in the block needed five seconds to run. There would not be five timers set up to fire. It would fire again one second after the block finished.

To see this work, let's tell EventMachine to stop 5 seconds after it starts.

Run it in the Terminal. Every second, it spits out “tick” and on the fifth second it prints out “BOOM”. You'll notice we don't get the fifth tick. The one-shot timer fires and calls `EM.stop` before the periodic timer fires, so EventMachine exits before the periodic timer can fire a fifth time.

THREADS: EM.DEFER

`EM.next_tick` allows us to schedule code to run in the next iteration of the main reactor thread. Your next question might be “How do we schedule something to run on a *different* thread?”

By default, EventMachine builds a thread pool with 20 threads available. You can send code to them by calling `EM.defer` with a block. Anything inside that block runs concurrently on a thread. And you can write multiple `EM.defer` blocks.

Here's an analogy of how this works. Imagine yourself with up to 20 friends. You ask them to throw a water balloon as soon as they receive it. Then you start handing water balloons to each of them. Effectively, you're throwing 20 water balloons at once. That's `EM.defer`.

EventMachine

A thread might be off doing some long-running process which I'm going to simulate with a call to "sleep". We'll sleep for two seconds and for one second.

Run this in the terminal. We'll see "I'm on a thread" and "a cool thread". The second thread finishes first, and then the first thread finishes. Nothing too surprising there.

If you need to know the results of your operation, you can pass two parameters to `EM.defer`. The first is the operation to execute on the thread. The second is a callback. The callback will be fired on the main reactor thread, and can receive data from the operation when it's done. If you need to process an intensive task, you can do that on the thread and then take the results of that task and pass it back to the callback. It executes on the main reactor thread, and sends the data to your client.

So a task is run on a thread, and the main thread is notified when it completes, along with some extra data if needed.

To use this in EventMachine, you just need to make two procs. I'll name the operation "op". In Ruby style, the return value of the operation is passed to the callback. No "return" keyword is needed. Here I'll return an array with the numbers [1, 2].

Make a second proc as the callback. It takes two parameters which I'll name "first" and "second". Execute them with "`EM.defer(operation, callback)`".

Run it and see that the operation happens, the callback is called, and 1, 2 is printed out.

`EM.defer` is a great way to optimize an existing application or design around tasks that will take more than a few milliseconds.

`EM::QUEUE`

The last two primitives we're going to look at are `EM::Queue` and `EM::Channel`.

`EM::Queue` is a built-in implementation of an ordered message queue.

Here's the analogy: You have, let's say, four water balloons. You give each of them in order to a friend. When you're ready, you ask for them back one at a time. The friend hands you the oldest water balloon and you throw it. This continues until all the water balloons have been handed back to you (but you can keep giving him more balloons and they'll be added to the bottom of the stack).

It has a few common uses centered around working with data. One is if you're implementing a low-level protocol and need to store data before you process it. The queue can keep track of it for you until you ask for it back. You can concatenate it together or do secondary processing with it. The second use is when you need to move data off a thread and execute it on the main reactor thread.

EventMachine

All of the `EM::Queue` process blocks we'll create will fire on the main reactor thread. This means that when you push data onto the queue or take data off the queue, the associated blocks don't fire immediately. Like `"next_tick"`, they will be scheduled by the reactor and will run at the next iteration of the reactor.

If you've used a queue from a web application, you may tend to think of a queue as something that *does work* on data. Procedural web applications use queues as a way to send data to other processes that do the work. But in this case, the queue is more elemental. It's just a temporary *storage place* for data. The main thread is the one that pulls the data off the queue and does work on it.

That being said, there's no problem if the main thread pops an item off the queue and immediately uses `EM.defer` or another EventMachine technique to run that work on a separate thread.

Enough explanation! How do we use it? It's built around "push" and "pop". First, create a new queue. Push data onto the queue. I'll push three separate items. Then I'll pop three items off the queue, printing each item.

Execute it in the terminal. If it worked correctly, you'll see "one, two, three."

Data can be added to the queue from a thread. You've already seen that `EM.defer` runs code on a separate thread. I'll push a few items onto the queue, sleeping briefly between each.

We can use the same `"3.times"` code to pull it off the queue and act on it.

Let's think briefly about the order in which this code will be executed. Because EventMachine blocks run asynchronously and the queue is populated from a separate thread, it's not immediately clear how to step through the logic.

Here's the key: if there are no items on the queue, your "pop" calls will be queued up and will wait for new data (but in a non-blocking way). If something is put onto the queue, the oldest "pop" callback will be used immediately. You can queue up these "pop" calls and they will be executed when data is available.

Going back to our terminal, if we rewrite our application we'll see one, two, three pop out from our queue.

If you're like me, there's still a question in the back of your head. What if I want to work with an endless queue? If I'm processing orders on my shopping cart, I don't want to stop at 3 orders, or 6, or 600.

EventMachine

The secret is to use the same technique that the periodic timer does internally. After processing a single piece of data, call “pop” to get the next item off the queue. It works best with a Proc. Define a proc that does work and then calls “pop” to get the next piece of data. Then launch the whole thing for the first time by making the same “pop” call somewhere in the body of EM.run.

TODO: Code sample of pop in another pop.

EM::CHANNEL

The last fundamental tool in the EventMachine toolkit is EM::Channel. EM::Channel is a publish/subscribe system where parts of your code can be notified when other parts send data. One or more subscribers receive data. Any code can push data to the channel. If you have some long-running code in a “defer” block, you can push data from it to other code that is waiting for a response.

A water analogy for this is a bit more difficult, but let’s give it a shot. It’s a gutter system on a house. Several friends are sitting at the end of their own downspout. You pour water into the top of the gutter and they each receive some water. If anyone has had enough, they can detach from the gutter system.

First we need to create a channel. In this case let’s use two deferred threads. In the first one, we subscribe to the channel. Print out that we’re the first thread and then print the message that was received. Sleep for three seconds and add another message to the channel.

The second thread will also subscribe to messages on this channel. This time we’ll store the result of the call to “subscribe”. This is a unique subscriber ID. I’m storing it in a local variable so I can unsubscribe later. In this case we’ll store it away and go to sleep for two seconds. Then unsubscribe using the subscription ID. You could do this if you only need to receive certain messages. You’ll only receive messages while you are subscribed to the channel.

In our main thread we’ll add a periodic timer. It will fire every second and it will just say “hello” into our channel.

Once I’ve fixed my spelling mistake we can jump to the terminal and run the script. I see “hello” gets printed out a few times, and then after that three second delay you don’t see our hello two getting printed out anymore; the second channel unsubscribed and doesn’t receive the messages. It’s all hello one.

How is EM::Channel used in production? At PostRank it’s used to build a firehose system for the content stream. The main API creates a channel when it starts up. As

EventMachine

content comes in it sends it to the channel. As clients connect, they subscribe to the channel and will receive all the messages as they come through the system.

There are many other useful features built into EventMachine, but these are the major ones that you should know about. I encourage you to take a look at the documentation to learn about things like watching the file system, watching processes, or watching system commands. If you need any of that sort of behavior, EventMachine has all of it built in. It can even read from the keyboard if you need to receive keyboard data.

In the next section we'll take a step upward in complexity and a look at one of EventMachine's most useful features: the Deferrable.

Chapter Four: Deferrable

In this section we're going to take a look at what are called EventMachine deferrables.

What is a deferrable? First, even though the name is similar, deferrables are different from the `EM.defer` method that you've seen a few times already.

A deferrable is a form of light-weight concurrency built into EventMachine. They look simple at first but have a delayed trigger feature that makes them really useful. From another angle, deferrables look like a simple state machine.

Imagine a magic water balloon. It's filled with water but won't explode immediately. You can throw it and it will wait. When you send it a message, it will explode.

Deferrables can be triggered with "succeed" or "fail". To complete the analogy, maybe this is similar to telling the balloon to explode with either hot water or cold water.

The trigger feature is useful for notifying other objects that the Deferrable is done. Objects will attach their own callbacks to it, and those will be run when the Deferrable is triggered with "succeed" or "fail".

Let's write code that uses a deferrable and then we'll look at some tips for using them in production code.

To define a deferrable, first define a Ruby class. I'll call it `MyDef`. To make that class deferrable, "include `EM::Deferrable`". This class is now deferrable.

It does not have any extra methods on it but it will handle all of the callbacks that are placed on it. You can attach callbacks and errbacks to instances of this class. A callback is called on success, an errback is called on failure.

EventMachine

In our case, we're going to make a simple wrapper function to make it easier to succeed or fail the deferrable. If we succeed we do "set_deferred_status" to ":succeeded", and if we wish to fail we just "set_deferred_status" to ":failed". That's it.

When this succeeds, all of the attached callbacks will be executed. If it fails, all of the errbacks will be executed. These will be executed in order and immediately. If you attach three or four callbacks to a single deferrable they will be executed in the order in which they were added. And they will all be executed before the next iteration of the reactor.

Now that we've created a deferrable class, let's create a couple instances of it so we can attach callbacks from external objects. Each callback takes a block. In this case MD1 succeeded. If you wish to attach an error handler, it's called "errback" and again takes a block. We're going to create two handlers in this case, so we can succeed one and fail the other one.

For the errback I'll print "failed". To run them, create a simple timer, wait two seconds, and execute my_function with "true" for the first one (it succeeds), and "false" for the second one (it fails).

Assuming all of your variables are named correctly, jump to the terminal and run the program.

In about two seconds we should see it spit out that MD1 has succeeded and MD2 has failed. We've now created a simple deferrable that we can tell to succeed or fail.

Before we look at other features of the Deferrable, let's review the concepts behind them.

Deferrables are used extensively in the internals of EventMachine and in related protocol adapters. If you look at something like em-http or em-jack, the connections returned by both of these are in fact deferrables. EM::Jack is a client library for the Beanstalk message queue. When your code launches, it can start making requests immediately and they will pile up deferrable callbacks until Jack successfully connects to the server. At that point, it triggers the "succeed" state and sends all your commands through to Beanstalk.

Your application doesn't have to wait for the connection. It just queues up the code that will run when it does finally connect. By that point, the main part of your program may be on to running other tasks while the deferred requests execute separately. I don't even want to think about how I would implement that apart from EventMachine. I'm sure it would be pretty complicated. But with EventMachine it's almost painless.

At this point, deferrables may still seem completely theoretical, without any real purpose.

EventMachine

There are two key concepts that helped me understand deferrables. First, with a deferrable you can attach callback handlers and error handlers at any time after the deferrable is created. This means that one piece of your code can create the deferrable and pass it around to other portions of your code, which will add callbacks and error handlers so their custom code can run when it has been triggered. It allows you to nicely decouple your custom code and just pass around deferrable objects instead of having to know if it's an HTTP deferrable or a Beanstalk deferrable. It's a perfect technique for the Ruby language; we don't care what objects we're working with as long as they implement the core deferrable functionality.

The second key concept is that deferrables often trigger *themselves* when they are ready. Here's an example from a later chapter: A deferrable is instantiated and a custom "query" method is called on the deferrable object. The deferrable class collects data from a remote API. When all the data has been collected and parsed, it tells itself to succeed. All attached callbacks are triggered and other code can work with the data returned by the API.

Let's return to the code.

If it seems like a lot of effort to create an entire class to create a simple deferrable, EventMachine comes with what's called a "default" deferrable. If you don't need any extra behavior attached to your class or you just want to add it later by modifying the class at run time, you can create what's called an `EM::DefaultDeferrable`.

This is an initialized deferrable object. We can attach callbacks to it. In this case you'll notice I'm actually passing the parameter to the block. When we succeed or fail the deferrable we can pass extra data and it will be passed into the callback blocks. In production this might be the data returned from a remote API. In this case we'll print out "hello name". I'll also stop the reactor because we won't need it to keep running.

Attach a quick errback.

Now let's trigger it. Again we're going to trigger it with a timer, two seconds from now. And we're going to succeed the callback. In this instance we're going to use the "succeed" method instead of calling "set_deferred_status". It makes things a bit shorter and clearer, especially with extra arguments such as "dan".

The flip side of "succeed" is the "fail" command. You can also pass extra arguments to "fail" and they will be passed to the "errback" block.

Jump over to the terminal and run it. You should see a delay of about two seconds, then the deferrable succeeds. You should see "hello Dan". You've now used the default deferrable to attach callbacks and errbacks.

EventMachine

MULTIPLE CALLBACKS & RE-SUCCESSING WITH MORE DATA

What if you have multiple blocks of code that need to be fired on success or error?

You can attach multiple callbacks to a single deferrable. The interesting thing about this is we can call “succeed” again and add extra arguments that will be passed to the next callback. Take the example here and call “succeed” again within the the callback. Pass in the name variable and the string “screencast”, which we’ll print to the terminal eventually.

Afterward I have a second callback that now takes two parameters. I’ll call the next callback with no arguments by using “set_deferred_status :succeeded”. The final success callback will end the program by calling EM.stop.

Run this example. You should see “hello Dan” and “Dan made a PeepCode screencast”. Then it exits.

This is a contrived example but in a larger program you might need to augment the callback chain as new information is received or calculated. In your first callback you might receive the HTTP response object, and in a subsequent callback you could pass the response code and body as two separate arguments. Accomplishing that would be easy now that you know how to re-succeed a deferrable. Add arguments when you call “succeed” and your subsequent callbacks will receive the response code and the body without needing to know the details of the HTTP response object.

RE-ATTACH A CALLBACK

Deferrables start simple but have many useful features.

Here’s another: you can continue attach callbacks or errbacks even after a deferrable has succeeded. Those callbacks or errbacks will be executed as soon as they’re added, so you don’t have to check the deferrable’s state before attempting to execute a callback on it. Your code will execute the callback immediately before the next tick of the reactor.

Let’s try it. We’ll create a new deferrable and add a callback to it. It will print out “succeeded”. Then, as usual, add a timer to simulate other work. Then we’ll tell the deferrable to succeed. Now let’s add another timer inside the first one. One second later we’ll add another callback which will print “done” and call EM.stop. The deferrable “df” has already been triggered with “succeed”, so any other callbacks added will be executed as soon as they are added.

Try it in the terminal. I expect to wait two seconds; see the message “succeeded”, then a second after that we should see “done” (which was added by the second callback *after* the deferrable succeeded). Finally, the program will stop and exit.

EventMachine

RESET

There's one other key concept to understand about deferrables: the reset.

Our example here is going to be a bit contrived but the functionality is used in em-jack. When em-jack connects to the Beanstalk server, all the built-up callbacks will be executed immediately. But if the client loses the connection to the Beanstalk server, it can reset the deferrables so any more communication with the Beanstalk server will pile up again. Once the client reconnects, it can execute all the pending callbacks.

Let's visualize this again with some water balloons. You tell a friend to throw any water balloons that you send over. This happens for a while until you send a "reset" message. All existing balloons are gone and any existing directions are forgotten.

You can add other directions, such as "juggle balloons", and you can even send some more water balloons. But your friend is in "reset" mode and won't do anything with the command or the balloons. Finally, you send the "succeed" message and the friend resumes with all the directions since the last reset. The friend starts juggling the balloons instead of throwing them.

In the analogy, you are the main reactor thread. Your friend is a deferrable. Your commands are callbacks. The water balloons are data sent by events. When the "reset" message is received, the deferrable not only pauses work, but also forgets about any callbacks registered up to that point. Other callbacks can be added, but nothing will be done while the deferrable is in reset mode. Once the deferrable receives the message to "succeed", it starts up with any callbacks that have been added since the reset.

Let's write the code for a resettable deferrable.

I'll create a class with three methods: "do_work", "connect", and "reset". These are my names, but we'll call the EventMachine API within them.

Name the class MyDef again. Include EM::Deferrable so this class is a deferrable class. In this example, the "do_work" function will register a callback that does some kind of work when the deferrable succeeds. I'll immediately create a callback when "do_work" is called. It prints "did work".

Now pretending that we have to connect to a server before we can actually do our work, let's write a "connect" method which will call "succeed". That triggers the execution of all this class's callbacks. Then, and this is the focus of this section, we're going to create a "reset" method that resets the deferrable. The EventMachine API for that is the "set_deferred_status" method. To reset, pass nil. Once it is called, any other callbacks or errbacks that are added will start piling up but won't execute.

EventMachine

To reiterate: “reset” is not only a pause; it’s a blank state. All the previously attached callbacks or errbacks will be removed when you call “set_deferred_status” with nil.

Let’s use this class. We’ll instantiate it and call “do_work” a few times. We’ll implement a cascade of timers that start and stop the deferrable, adding callbacks along the way.

The first timer will fire in two seconds and will connect to our service. Existing callbacks will execute. I’m going to add another “do_work” call right after so we can see some output. Those three calls should all be executed at the same time, as soon as “connect” is called.

Add a second timer which will reset the class. This simulates being disconnected from the remote server. I’m going to call do_work again, this is number four. But it won’t be executed yet; the deferrable is in the nil/reset state. Add one last timer: pretend that we’ve connected again and callbacks can execute. Finally, stop.

These timers are nested so they fire in sequence. If they had been defined in separate blocks, they would fire separately at undefined times.

It’s a bit messy, so let’s review the concept in question: callbacks can be added to a deferrable and will execute when it succeeds. When the deferrable is reset, subsequent callbacks will pile up but won’t execute. When the deferrable succeeds again, those callbacks will be executed.

Jump over to the terminal. Run the program. In about two seconds we see it print out our first three “do_work” calls. Reset has been called and a fourth is queued but doesn’t execute. Then a second later we see the fourth...the deferrable is back to “success”.

These are the basic, most useful features of EM::Deferrable. Success, failure, callbacks, errbacks, re-attaching, resetting. EM::Deferrable is an extremely useful tool and you’ll use it frequently in your applications.

In the next section of this screencast we’ll put these ideas together by creating a simple client/server application.