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, handing 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

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.

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

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."

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."

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.

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.

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.

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.

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.

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.

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

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.

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.

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.

MULTIPLE CALLBACKS & RE-SUCCEEDING 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.

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.

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.

Chapter Five: Servers & Clients

In this section we’ll look at the ways you can define low level clients and servers in EventMachine. We’ll mostly focus on the server portion but most of the server API has client side counterparts as well, with client creation methods instead of server creation methods. Let’s get started.

TCP SERVER

EventMachine

There are various ways to define a server or client handler in EventMachine. There is the class style which we saw earlier. I can define a server and inherit from EM::Connection. This will create a connection handler.

Or, implement it as a module. When you use the module it will be mixed into an anonymous class that EventMachine creates, and all of your methods in the module will be available.

Third, define a handler in the server block: “start_server”

When you use “start_server”, you can pass a block which will receive your connection object. There we can define our methods.

Note that if you’re working in a block, you will not be able to hook into the “post_init” method, or the “initialize” method. Both will have already been executed by the time the block is executed. So use the class if you want to implement those methods.

We can also work with instance variables. Suppose that our class had an attr_accessor, for “name”. When we define the server, we can pass in the connection object and set the “name” variable. Again, this will not be available in “post_init” but will be available in the rest of the methods of your connection class.

The third way is to pass it into the initializer. That’s done by providing arguments to the start_server method. In our module we just write an “initialize” method and take arguments. Those variables are now available in the module.

The third method where you pass it into “initialize” is probably the best option. Everything passed in will be available in “post_init”, so if you need access to it in “post_init”, that’s the best way to do it.

The block method is useful if you want to define throwaway servers. Typically for our work at PostRank we always use a class.

As I mentioned, all these methods will work in a similar way for an EventMachine client. The major difference is the kick off method. Clients use EM.connect instead of start_server. Everything we just mentioned will start exactly the same for an outgoing connection as for a running server.

I encourage you to go take a look at the documentation for more information on any of these. You can build a proxy server that re-broadcasts data from another server. You can do TLS, so secure SSL connections can be created from your servers or through your clients. There are also functions to send file-based data. There is send_file_data which sends data as a single chunk, or stream_file_data which uses HTTP chunked encoding. And there’s close_connection and close_connection_after_writing. The difference being

that close_connection_after_writing will make sure that any outbound data is sent before the connection is closed.

UDP SERVERS

There are several other servers we can create in EventMachine. A UDP server works similarly to TCP servers. We're going to use a handler that inherits from EM::Connection and receives data. To create our UDP server, it's just "EM.open_datagram_socket", give it your address, your port, and your handler.

As with TCP servers, we could have used a module, or we could have used an anonymous class that we create in a block. We can pass in data to "initialize" or call methods inside the block. It's all set up exactly the same. You basically just change the name of the function you're calling.

In order to send a datagram it's a bit different. You have to call open_datagram_socket, but it doesn't actually have to be connected to anything. If we had wanted to, we could have saved the previous datagram but in this case I'll create a new socket. We create a socket and send a datagram. There are limits on the amount of data you can send based on your host operating system but we're not going to go over them with our simple "hi". This will send a datagram packet.

Switch over to the terminal. Run the client. You'll see we received the command "hi". UDP is useful for doing things like sending data to monitoring systems like Ganglia or other data where you don't need the reliability of TCP.

UNIX SOCKETS

The third type of server works with Unix sockets. If you're a single host, you can create UNIX sockets (domain sockets).

There are two ways to start one. You can use "EM.start_server" the same way we did with the TCP server but pass in a filename instead of an address. You don't need a port. It will create a domain server. This looks similar to what we've seen before.

The other way you can start a server is with the "start_unix_domain_server" method. It takes the same arguments.

To connect to a UNIX domain server from a client, there's an "EM.connect_unix_domain" method which takes the file. In this case we're going to create an anonymous body and send data to the server. We can create the same thing to connect to our second server.

Jump over to the terminal and run the client. We can see that it will connect to the socket and print out "hello" and "hello2".

Domain servers, UDP servers and TCP servers are all fairly simple to implement in EventMachine. The fact that they all use the same EM.connection subclass or anonymous class makes is easy to mix, match, and reuse.

The keyboard server uses the exact same connection class, so you can see how once you've learned the basic methods and connection classes of EventMachine, everything you do will use the same ideas. You can reuse the same knowledge over and over again. It makes it really simple.

One type of server that we did not mention is the HTTP server. Async Sinatra is an exciting new project that makes it easy to work with basic routing, query string and form parsing, and other web-related programming. Check out the source files for this screencast to see an example that uses Async Sinatra.

But in the next chapter we'll put these ideas together to write a simple mapping application that can be viewed in a web browser.

Chapter Six: Websocket server

We've covered several concepts at a simple level. Let's do something a little bit more interesting.

In this example we're going to create a WebSocket server in EventMachine. When you send in a postal or zip code it will send a request to popular location-based social network APIs. It will plot recent check-ins in the area.

Here is a postal code. When I submit the form, it pops up on Google Maps and drops icons to show recent check-ins on Gowalla, Facebook, and Foursquare. You can see the places where these people are at currently. If you enter a different postal code, it shifts the map and populates results for that area. You can move around to different areas and it keeps adding onto the map and re-centering the map as you request it.

Writing applications with WebSockets differs from the standard request/response cycle that most web developers are used to. The server can send data to the client at any time instead of waiting for the client to ask for more. EventMachine is perfect for this style of data delivery. I think that we're on the verge of an explosion of new kinds of web applications that are now possible with WebSockets. And EventMachine is a great way to write them.

INDEX.HTML

Let's start building the app. Jump over to TextMate. Let's scan through the existing index file; I won't go into detail about how the JavaScript works with WebSockets. It's fairly straightforward here.

This example requires a browser that implements WebSockets. Safari or Chrome will work. You'll see an alert otherwise.

Here's a brief summary of the HTML and JavaScript. We're grabbing jQuery from Google. We're also using the Google Maps API. When we start up, we create a websocket connection. When the websocket receives a location message from a server it will recenter the map using the GoogleMaps API. If it receives any other message it sets a marker at the given longitude and latitude. We write a message to the console when the connection closes. The "submit" button posts data to the server via the WebSocket. It calls WebSocket.send. The form is a simple input label.

The more interesting code is on the server.

SERVER

We're going to need a few things to get started. First, you'll need EventMachine 1.0, or the prerelease version until the official 1.0 release. You'll need the current release of EM::HttpRequest. You'll need EM::Websocket. And I'll use the nokogiri and JSON libraries to do a bit of extra input and output work. Set up the requires and then as we've seen several times, create an EM.run block.

The first thing we need to do is set up a websocket server; EM::WebSocket makes that easy. Create a websocket, call start on it, tell it the host and port to run on: localhost port 8080. The HTML file is currently hard-coded to look for this port. That will provide us with a websocket to the block.

I can then have an "onopen" handler for the websocket. I'll print a message to standard out to show that a client has connected. Websocket has an "onmessage" callback which will receive any messages sent from the client. There's also a websocket "onclose" which is called when the connection is closed. There's a websocket "onerror" which gets called if any errors happen. It receives the error exception object.

There's nothing too exciting in the boilerplate. The real work starts in our onmessage callback.

Switch over to the terminal and do an initial test to see if the initial code is running correctly. Run the server.

Drag the "index.html" to your browser; it's not served by the application but will run as a static file. Refresh the browser and you should see on the terminal that the client is

connected. Refresh the browser again and we can see the client closed its connection, then reconnected. If we kill it we can see we're terminating the websocket server and the client connection closes too.

ONMESSAGE

Everything is working as expected. Let's move on to implement the "onmessage" handler that loads data from remote APIs and sends it back to the browser.

First we need to call a geocoder to convert a given zip code or postal code into latitude and longitude. Lat/long can be displayed on a Google map. Yes, we're writing a server, but we're also using EventMachine as a client to make API requests. If you were to use a non-EventMachine client library, you would be in danger of blocking the entire process while the request was returned from the remote server. So when writing an EventMachine server, you should use the appropriate EventMachine client to make API requests.

Here's the sequence of events that we're trying to accomplish. The HTML form sends data to the server. The server will use that data to make an API request to another server. When that data is received, the server will parse the data and tell a deferrable to "succeed". This triggers another callback that will use WebSockets to send data back to the browser.

TODO: Diagram of form data, GeoCode.new, API request, geocode.callback, ws.send to browser

Most of this code will happen in a deferrable class that I'm going to name Geocode. As we saw earlier, create a Ruby class and include EM::Deferrable. It will build the deferrable object. We're going to implement one custom method: "query". It takes a postal code.

I'll use the geocoder.ca API to do the conversion. It works for both Canadian postal codes and US zipcodes. You can pass in either a postal code or a zip code, but the key is "postal". The geocode needs to be set to use XML. The API will return XML data which we can parse with nokogiri. Use EM::HttpRequest to make the API request. Note the slash on the end of the URL is required; EM::HttpRequest requires a valid path in order to make a query, so the geocoder URL must end in a slash.

Do a GET request. The querystring parameters are specified with the "query" key. Now here's where another concept comes into play. EM::HttpRequest is a deferrable object. We'll attach a callback to it and when it succeeds, we'll parse the response.

Which happens now. We want to grab the latitude: the "latt" entry. And we'll need to grab the longitude: the "longt" entry. Then we will call "succeed" our own deferrable,

passing in the latitude and longitude. This will pass those two parameters to all external callbacks, one of which we will define later.

This technique confused me at first, but now seems completely sensible. The class knows when it's ready to trigger the callback, and has access to the data, so it calls "succeed" on itself.

If EM::HttpRequest fails, we'll fail our own deferrable on the Geocoder class. That's all that's needed for a simple API client. You can use this pattern for your own API requests. And in fact, we'll reuse it again, too in a few minutes.

Scroll down to the block that received the message. We'll create a new GeoCode object. We'll look for the provided postal code sent on the web form. We'll attach a callback to the geocoder deferrable. Here's the spot that was just mentioned: it receives the latitude and longitude as arguments.

The final actions are less complicated: send "type => location" to the websocket which tells the browser to focus the map on the given location. Pass the latitude and longitude. Our response must be in JSON format.

The geocoder request could fail. Let's print a message to the console in an errback. We will also send a message to the websocket that shows that the "geocode failed." This notifies both the local server and the remote client that something went wrong.

That should allow us to send in a postal or zip code and our map should zoom to the corresponding location.

Jump over to the terminal and run the server. Back on the browser we just need to reload the page. You should see on the server side that a client connected. Submit a location in the form and the map should zoom to where you are.

Try entering another location. You'll see the map shifts over to the new location.

The geocoder is up and running.

GOWALLA

The next step is to take the latitude and longitude provided by the geocoder and query several online APIs. In this example we'll implement three location-based APIs: Gowalla, Facebook, and Foursquare. They will look very similar to the GeoCode class. They will all be EM::Deferrables. They'll will all use EM::HttpRequest to make their outbound requests. They will all call "succeed" with some data that will be sent back to the browser over the websocket.

Each of these use the same concepts that we just implemented in the GeoCode class. The details for each API differ only slightly.

I'll paste in the Gowalla code and we'll look through it. You can see we're using an EM::Deferrable. We have a similar query method that takes the latitude and longitude. We'll pass them into Gowalla, specifying the latitude, the longitude and the radius in meters. We're looking for check-ins within a 10,000 meters radius around this point. We'll make an API request to the Gowalla Spots API using these query parameters. We'll tell EM::HttpRequest to timeout if nothing was heard within 20 seconds. We'll pass in header fields to specify that we want a JSON response.

We then define our callback. I'll use a technique to show how to do a long running operation. If you think that parsing the data will be heavy, you can do what I'm doing here. Use EM.defer to run the long operation separately. It will execute the "parse_callback" proc when it's finished.

The long operation is a proc which parses the XML response from Gowalla. We'll shift all the data spots entries into our locations array. This includes "gowalla" as the type, the name of the spot, the latitude, and the longitude. When the locations array has been assembled, we return it. It can easily be converted to JSON and sent back to the browser.

The return value of the long_operation will be passed into the parse_callback. All it needs to do is call "succeed" with the locations array so other callbacks can use it. We will also create a simple Gowalla errback which will print a message to standard out and fail the deferrable.

As before, let's use it. It's not possible to start querying location-based services until the latitude and longitude are available, so I'll use the Gowalla class inside the geocoder callback.

We'll create a new instance of the Gowalla class. We will then tell Gowalla to run a query with a given latitude and longitude. A callback on the Gowalla object will receive the locations data array. When we call "succeed" with "locations" as an argument, this is where it ends up. We'll take that location data and send it to the websocket as JSON data.

The rest happens on the client; each of the locations is plotted on the map.

As an errback we'll send a message to the websocket that this type of request failed. It is also JSON data.

Try it out.

Jump over to the terminal and restart the server. Switch to the browser, refresh the page, and submit your postal code. If you live in a populated area, you should see a bunch of Gowalla pins drop onto the map. It happens fairly quickly.

We're now retrieving, parsing, and plotting Gowalla data.

FACEBOOK

Next up is Facebook. The Facebook client will be similar to Gowalla, with a few tweaks. Again I'll paste in the code and explain it.

Class Facebook includes EM::Deferrable. We define a custom "query" method which takes a latitude and longitude. Facebook requires extra authentication, so it takes two requests to get the checkin data. First we have to make a request that will grant us access to the API with an OAuth token. We make a request to the OAuth access token URL. If that succeeds, we then receive a token which we can pull out of the response using a regular expression.

I'm providing an API key for you, but if it doesn't work, you can register for your own at developers.facebook.com.

Now we can make the query to Facebook for the data. Build the query parameters. We're making a place query centered on the given latitude and longitude within a distance of 1,000 meters. Provide the access token.

Then make a Facebook request to graph.facebook.com/search. Pass in the query data. If it is successful, we'll parse the response as JSON and build up an array of locations as before. The final method in the callback is the call to "succeed" with the locations array. Code that uses this class will receive the locations array in any callbacks they define.

I'll omit the long operation example from the Gowalla class. Generally, parsing JSON happens pretty quickly and isn't a bottleneck.

Similar to Gowalla, we have the same sort of errback for Facebook. In this case we have two of them based on whether it's the query or the access token that fails. You can distinguish them based on the different levels of nesting.

REFACTOR FINAL CALLBACK

Now let's use this Facebook object in the EM.run loop. I need to do the same thing that was done for Gowalla: make a new object, send a query, receive the locations in a callback and send them to the browser. But instead of rewriting or copying all that code again, let's refactor the existing code. I designed both classes to have the same inputs and outputs, so this will be easy to do with a few Ruby proc objects.

Write a proc that receives the array of locations and handles the functionality that was done for Gowalla. It needs to take each one of the locations and send them back through the websocket.

We can also define an error callback, which is the same as the error callback we created for Gowalla. I'll paste it in, and most of this code will be the same.

Let's take the refactoring a step further. We'll run the same code for each class, so start with an array of class names that can be processed in a loop. Gowalla, Facebook, each class. The correct spelling of "class" is a keyword in Ruby, so a convention is to use "klass".

Make an instance of the class. Call the query method, and then instead of defining the callbacks and errbacks, we'll just set the callback and errback to the existing procs, using the ampersand.

That should set up both Gowalla and Facebook to run queries and send the data back to the browser.

Jump over to the terminal and restart the server. In the browser, refresh the page. Submit the request, and you should see pins for Gowalla and Facebook on the map.

FOURSQUARE

The last service we're going to implement is Foursquare. This already looks a bit repetitive. Foursquare is going to be very similar to the other two services I've implemented. We'll add Foursquare to our array since we'll need to create it at the end.

I will again paste in the code for the class. You can see that we've created our Foursquare class. It's a deferrable. We have our query method. We pass in the longitude and latitude, on a limit of distance. We want check-ins from Foursquare and we have our client ID and secret. We then make an API request to their venues search method and pass in our Foursquare query data.

The callback is similar. It takes the JSON data back from Foursquare, parses the response groups, sets the items into our locations array, and calls succeed with the array of locations.

If for some reason the request fails, we just set some errback information as before.

For the last time, jump over to the terminal. Restart the server. Refresh the browser and submit a request. Now the icons for the three different services dropping in almost at the same time. You can see that the three requests are happening asynchronously and are being sent back through the websocket to the client.

That is it for our simple little example program.

You can see a bit of repetition in the code. We created a deferrable, we attached our callbacks, we attached our errbacks, and then we told EventMachine to do its thing.

It does start to get messy for a two-call API like Facebook where you have callbacks nested within callbacks; your code can start to get confusing. Or the code can be difficult to read if you're working with a long operation deferrable. Callback and errback code may be listed far before the code that uses them.

In the next section we'll take a look at something a few people are starting to use in order to solve these problems. Ruby 1.9 has a feature called Fibers that can take your asynchronous EventMachine code and give it a more synchronous appearance. We'll refactor the websocket app to use Fibers.

Chapter Seven: Ruby 1.9 Fibers

In this section we're going to look at EM.synchrony.

EM.synchrony is a wrapper on top of EventMachine and a feature of Ruby 1.9 called Fibers. It gives a more synchronous feel to your asynchronous code.

Instead of working with callbacks and errbacks, you can assign variables as you would in synchronous code, but Ruby 1.9 makes the assignment happen asynchronously.

HIGH-LEVEL OVERVIEW

Under the hood, synchrony automatically hooks up a callback and errback which transparently pause and resume the current Fiber. For example, after it makes an EM::HttpRequest, it yields to the current Fiber so it goes to sleep until the deferrable is finished. From the perspective of your code, it looks like you're making a synchronous call and assigning the result to a variable.

TODO: Diagram of Fiber vs. Ruby 1.8

```
# Ruby 1.8
g = Gowalla.new
g.callback do |locations|
  locations...
end

# Ruby 1.9 Fibers
g = Gowalla.new
locations = g.query(lat, long)
```

There are no callbacks, no errbacks. You just take the response from EM::HttpRequest and work with it as though it's the value passed to the callback.

In order to do this we need to install em-synchrony. This will only work under Ruby 1.9. It does work in Rubinius and jRuby although there are certain performance issues which are being worked out in those two implementations.

To patch this into our server, require em-synchrony and em-synchrony/em-http. This patches the em-http library to be em-synchrony aware. If you look at the em-synchrony code you'll see there's a bunch of wrappers for other clients like em-redis, em-mongo. EM::Jack has Fiber capabilities built-in. Many other EventMachine drivers also work either through synchrony or built-in to the library.

EM.SYNCHRONY

The one major change we need to make to use synchrony is to call EM.synchrony instead of EM.run. This kicks off a new Fiber and runs the EM run loop inside the Fiber.

CONVERT FACEBOOK CLASS

In this example I'll convert the Facebook request to work with synchrony. If you remember from the last section, in the Facebook deferrable we made a query to get an access token. Inside the callback for *that* query we made a second query to the search API. When that callback finished, we got the location data.

If we were to use em-synchrony to implement this method, we would end up with something similar to this code here. Make the query, get the access token, and then just start checking the response header that's returned from EM::HttpRequest. There is no callback or errback.

We then immediately pull the response token out of the Facebook request and create our second EM::HttpRequest to the search API. Again we access the response immediately as though the code was synchronous. You can see how this makes the code a bit easier to read. It's a bit more of the normal flow that people would expect when they're reading and writing networking code.

But of course, it's *not* synchronous code, so if you're writing a library or even just application code, make sure that it's communicated that you're using Fibers.

Chapter Eight: Goliath

For the last year, a special EventMachine server has been in use at PostRank. It's called Goliath.

EventMachine

The Goliath framework is built around Fibers and em-synchrony drivers. It creates fully synchronous looking APIs. We've found that for us it makes it a lot easier to create, debug, and train people on how to write and work with these APIs.

Right now I'll give you a quick example of creating a Goliath API that is a proxy server to any arbitrary URL. It will download a document from a webserver and pass it through.

Goliath has a few dependencies. Run "gem install goliath" and pull in any requirements.

In order to use Goliath we just "require 'goliath'". In this case we're going to be using em-http, so require that in through em-synchrony. One thing to note with Goliath is that the class you create has to be named the same as the file you're going to be running with (or at least a camel cased version, as is standard for most Ruby libraries). It uses the filename to do a lookup and execute your application. In this case my file is named "get" so I'll create a "Get" class.

Goliath APIs all inherit from Goliath::API.

Goliath is Rack aware. If you're using asynchronous Rack middleware, it will work well with Goliath. There's an async Rack project which Goliath requires and bundles, or there are other Rack middlewares already wrapped for you. Look at the documentation to see what is available to be used.

An important distinction is that Goliath is Rack aware, but not Rack compatible. The Goliath server does add some content to the environment, or ENV, that isn't in the standard Rack implementation. A server created with Goliath may not work with other Rack code correctly. That's something to keep in mind if you're trying to move from Goliath to Unicorn or from Goliath to Rainbows. If you do need to run one of those servers for other code, design your system so the Goliath server can run on its own.

When you want to use middlewares in Goliath, instead of creating a rackup file as you would with a normal rack application, you "use" it inside the class. In this case we'll use the Goliath parameters parser. Any middleware you want to use can be used directly in the class like this and they'll all be loaded up when the class starts.

Next, implement a "response" method. Goliath's main method is named "response" as opposed to Rack's "call". "call" is implemented inside the Goliath API and sets up the Fiber context. It will execute the "response" method on your class. It also handles sending back the correct asynchronous response code to the base server implementation so it can be delivered asynchronously.

In this case we'll make an EM::HttpRequest with the URL from the parsed parameters. Run a "get" on it. I want to return a 200 response. Inside the headers we're going to

return a status. As the body I'll return the body that I received back from the remote server. That right there is a Goliath API.

Switch over to the terminal and run the "get" application. Run it with the "-sv" flag. This runs it on standard out and in verbose mode.

You should see that the Goliath server has started. By default it runs on port 9000 and in development mode.

Switch to another terminal and run a curl request for google.ca. You can see that it runs. We get back a bunch of data. The server responded with a 200 status. It sent back 9K and it took 89 milliseconds.

That's a fairly trivial example of Goliath. You can do a lot more stuff with Goliath, including things like streaming requests and responses. It's fully pipeline aware.

Goliath works on MRI, Rubinius and JRuby. MRI is currently the fastest implementation because of the way Fibers are implemented. But they are catching up and there are patches out for JRuby that make Fibers a lot faster.

So in the near future, all Ruby implementations will support Fibers and Goliath can run on any of them.

If you look inside the "examples" directory you'll find a simple HTTP proxy. There are examples for using it with ActiveRecord and Rack routing.

So take a look at Goliath, if for nothing else than an example how to use em-synchrony wrapped EventMachine drivers in production applications.

Chapter Nine: Conclusion

I hope this screencast will inspire you to think about the kinds of applications that can be written with EventMachine. The quick success of Node.js and other evented servers has shown how useful evented servers can be.

It's common to think of Ruby as slow and inefficient, but a basic EventMachine server can handle over 10,000 events per second. And Jordan Sissel recently started an experimental implementation of EventMachine on JRuby that clocked 70,000 events per second! Only a small percentage of websites or backend processors would ever need better performance than that.

In our PeepCode screencast on Node.js, we built a real-time map that plots visitors to your website. It took me only a few hours to rewrite it in EventMachine. I think the

resulting application is a bit cleaner and easier to understand. I've included the code for that application in the code download for this screencast at PeepCode. It uses Async Sinatra, Faye, and other EventMachine libraries.

With increased browser support for WebSockets, I think we're on the verge of a new wave of web applications that could be as big as Ajax was. There's no way you can write efficient, always-connected web applications with traditional frameworks, but now with EventMachine and WebSockets, a whole new frontier of applications is open for exploration!