* Event-driven computing

If we kept just buying hardware to support ever-increasing features or users, it wouldn’t be very cost-effective.Node is an attempt to solve this problem by introducing the architecture called event-driven computing to the programming space for web servers. When you connect to a typical web server, it creates a new instance of a program on the server that represents your request. That program runs from the top to the bottom (following all of the function calls) to provide your web page. This means that the server has to allocate a fixed amount of memory to that request until it is totally finished, including the 100ms+ to send the data back to you. Node doesn’t work that way. Instead, Node keeps all users in a single program. Whenever Node has to do something slow, like wait for a confirmation that you got your data (so it can mark your request as finished), it simply moves on to another user.

* The Event Loop

Node takes the approach that all I/O activities should be nonblocking (for reasons we’ll explain more later). This means that HTTP requests, database queries, file I/O, and other things that require the program to wait do not halt execution until they return data. Instead, they run independently, and then emit an event when their data is available. This means that programming in Node.js has lots of callbacks dealing with all kinds of I/O. Callbacks often initiate other callbacks in a cascading fashion, which is very different from browser programming. There is still a certain amount of linear setup, but the bulk of the code involves dealing with callbacks.

* “single-threaded”

Yes, yes, we can see that you are rubbing your tummy and patting your head at the same time—well done. But if you try to do any serious activities at the same time, it goes wrong pretty quickly. This is like JavaScript. It’s great at letting events drive the action, but it’s “single-threaded” so that only one thing happens at once.

This single-threaded concept is really important. One of the criticisms leveled at Node.js fairly often is its lack of “concurrency.” That is, it doesn’t use all of the CPUs on a machine to run the JavaScript. The problem with running code on multiple CPUs at once is that it requires coordination between multiple “threads” of execution. In order for multiple CPUs to effectively split up work, they would have to talk to each other about the current state of the program, what work they’d each done, etc. Although this is possible, it’s a more complex model that requires more effort from both the programmer and the system. JavaScript’s approach is simple: there is only one thing happening at once. Everything that Node does is nonblocking, so the time between an event being emitted and Node being able to act on that event is very short because it’s not waiting on things such as disk I/O.

**NODE: It is an single threaded, event driven, non-blocking language.**

* Serial vs Parallel I/O

Serial is obvious: do this I/O, and after it is finished, do that I/O. Parallel is more complicated to implement but also easy to understand: do this I/O and that I/O at the same time. The important point here is that ordering is normally considered implicit in serial tasks, but parallel tasks could return in any order. Groups of serial and parallel work can also be combined. For example, two groups of parallel requests could execute serially: do this and that together, then do other and another together.

* In Node, we assume that all I/O has **unbounded latency.** This means that any I/O tasks could take from 0 to infinite time.

 Ordered serial requests are also easy to make by nesting or referencing callbacks together so that the first callback will initiate the second I/O request, the second callback will initiate the third, and so on. Even though each request is asynchronous and doesn’t block the event loop, the requests are made in serial. This pattern of ordered requests is useful when the results of one I/O operation have to inform the details of the next I/O request.

* Req and Res

Now when you point your browser to 'localhost:8080', the connection closure is invoked with a **req and res object**. The req is a readable stream that emits 'data' events for each incoming piece of data (like a form submission or file upload). The res object is a writable stream that is used to send data back to the client. In our case we are simply sending a 200 OK header, as well as the body 'Hello Http'.

* Single-Threaded

As we’ve mentioned, Node is single-threaded. This means Node is using only one processor to do its work. However, most servers have several “multicore” processors, and a single multicore processor has many processors. A server with two physical CPU sockets might have “24 logical cores”—that is, 24 processors exposed to the operating system. To make the best use of Node, we should use those too. So if we don’t have threads, how do we do that?

* Cluster

Node provides a module called cluster that allows you to delegate work to child processes. This means that Node creates a copy of its current program in another process (on Windows, it is actually another thread). Each child process has some special abilities, such as the ability to share a socket with other children. This allows us to write Node programs that start many other Node programs and then delegate work to them. The master process manages thechild processes, but when the children interact with I/O they do it directly, not through the master. This means that if you set up a web server using cluster, requests don’t go through your master process, but directly to the children. Hence, dispatching requests does not create a bottleneck in the system.

* Listening for an event

server.on('event', function(a, b, c) {

//do things

});

* URL

It returns a data structure representing the parts of the parsed URL. The components it produces are:

* href
* protocol
* host
* auth
* hostname
* port
* pathname
* search
* query
* hash
* **Streams**

Many components in Node provide continuous output or can process continuous input. To make these components act in a consistent way, the streamAPI provides an abstract interface for them. This API provides common methods and properties that are available in specific implementations of streams. Streams can be readable, writable, or both. All streams are EventEmitter instances, allowing them to emit events

#### Readable streams

The readable stream API is a set of methods and events that provides access to chunks of data as they are sent by an underlying data source. Fundamentally, readable streams are about emitting data events. These events represent the stream of data as a stream of events. To make this manageable, streams have a number of features that allow you to configure how much data you get and how fast. The basic stream in Example 4-16 simply reads data from a file in chunks. Every time a new chunk is made available, it is exposed to a callback in the variable called data. In this example, we simply log the data to the console.

* Streams

Let’s look in more detail at one of the common patterns used in dealing with streams. The spooling pattern is used when we need an entire resource available before we deal with it.

* DNS

The dns module consists of two main methods and a number of convenience methods. The two main methods are resolve(), which turns a domain name into a DNS record, and reverse(), which turns an IP address into a domain. All of the other methods in the dns module are more specialized forms of these methods

dns.resolve() takes three arguments:

A string containing the domain to be resolved

This can include subdomains, such as www.yahoo.com. The www is technically a hostname, but the system will resolve it for you.

A string containing the types of records being requested

This requires a little more understanding of DNS. Most people are familiar with the “address” or A record type. This type of record maps an IPv4 domain to a domain name (as defined in the previous item). The “canonical name,” or CNAME, records allow you to create an alias of an A record or another CNAME. For example, www.example.com might be a CNAME of the A record at example.com. MX records point to the mail server for a domain for the use of SMTP. When you email person@domain.com, the MX record for domain.com tells your email server where to send their mail. Text records, or TXT, are notes attached to a domain. They have been used for all kinds of functions. The final type supported by this library is service, or SRV, records, which provide information on the services available at a particular domain.

A callback

## Processes

Although Node abstracts a lot of things from the operating system, you are still running in an operating system and may want to interact more directly with it. Node allows you to interact with system processes that already exist, as well as create new child processes to do work of various kinds. Although Node itself is generally a “fat” thread with a single event loop, you are free to start other processes (threads) to do work outside of the event loop.

### process Module

The process module enables you to get information about and change the settings of the current Node process. Unlike most modules, the processmodule is global and is always available as the variable process.

#### process events

process is an instance of EventEmitter, so it provides events based on systems calls to the Node process. The exit event provides a final hook before the Node process exits

rocess.version

Contains the version number of the instance of Node you are running.

process.installPrefix

Contains the install path (/usr/local, ~/local, etc.) used during installation.

process.platform

Lists the platform on which Node is currently running. The output will specify the kernel (linux2, darwin, etc.) rather than “Redhat ES3,” “Windows 7,” “OSX 10.7,” etc.

process.uptime()

Contains the number of seconds the process has been running.

##### process.stdin

stdin is a really useful device for interprocess communication. It’s used to facilitate things such as piping in the shell. When we type cat file.txt | node program.js, it will be the stdin stream that receives the data from the cat command.

### Child Process

The child\_process module allows you to create child processes of your main Node process. Because Node has only one event loop in a single process, sometimes it is helpful to create child processes. For example, you might do this to make use of more cores of your CPU, because a single Node process can use only one of the cores. Or, you could usechild\_process to launch other programs and let Node interact with them. This is extremely helpful when you’re writing command-line scripts.

All child processes have some common properties. They each contain properties for stdin, stdout, and stderr, which we discussed in “Operating system input/output”. There is also a pid property that contains the OS process ID of the child. Children emit the exit event when they exit. Other data events are available via the stream methods of child\_process.stdin, child\_process.stdout, and child\_process.stderr.

#### Using CouchDB

One of the nice things about CouchDB is that its API is actually all just HTTP. Because Node is great at interacting with HTTP, this means it is really easy to work with CouchDB. Exploiting this fact, it is possible to perform database operations directly without any additional client libraries.

### Redis

Redis is a memory-centric key-value store with persistence that will feel very familiar if you have experience with key-value caches such as Memcache. Redis is used when performance and scaling are important; in many cases, developers choose to use it as a cache for data retrieved from a relational database such as MySQL, although it is capable of much more.

#### Hashes

Hashes are objects that contain multiple keys. [Example 6-11](http://chimera.labs.oreilly.com/books/1234000001808/ch06.html#example6-11) sets a single key at a time.

#### Lists

The list type can be thought of as multiple values inside one key (see [Example 6-14](http://chimera.labs.oreilly.com/books/1234000001808/ch06.html#example6-14)). Because it’s possible to push content to the beginning or end of a list, these collections are ideal for showing ordered events, such as lists of users who have recently received an honor.

This example demonstrates a first-in-first-out (FIFO) queue using Redis’s list commands. A real-world use for FIFO is in registration systems: the quantity of incoming registration requests is too great to handle in real time, so registration data is hived off to a queue for processing outside the main application. Registrations will be processed in the order they were received, but the primary application is not slowed down by handling the actual record creation and introductory tasks such as welcome emails.

#### Sets

Sets are used in situations where it is desirable to have lists of nonrepeated items, as in [Example 6-15](http://chimera.labs.oreilly.com/books/1234000001808/ch06.html#example6-15).

#### Subscriptions

Redis supports the publish-subscribe (or pub-sub) messaging pattern, allowing senders (publishers) to issue messages into channels for use by receivers (subscribers) whom they know nothing about (see [Example 6-17](http://chimera.labs.oreilly.com/books/1234000001808/ch06.html#example6-17)). Subscribers register their areas of interests (channels), and Redis pushes all relevant messages to them. Publishers do not need to be registered to specific channels, nor do subscribers need to be listening when messages are sent. Redis takes care of the brokering, which allows for a great deal of flexibility, as neither the publisher nor the subscriber needs to be aware of the other.

### MongoDB

Because Mongo supplies a JavaScript environment with BSON object storage (a binary adaption of JSON), reading and writing data from Node is extremely efficient. Mongo stores incoming records in memory, so it is ideal in high-write situations. Each new version adds improved clustering, replication, and sharding.

Because incoming records are stored in memory, inserting data into Mongo is nonblocking, making it ideal for logging operations and telemetry data. Mongo supports JavaScript functions inside queries, making it very powerful in read situations, including MapReduce queries.

Using MongoDB’s document-based storage allows you to store child records inside parent records. For example, a blog article and all of its associated comments can be stored inside a single record, allowing for incredibly fast retrieval.

### MySQL

MySQL has become the workhorse of the open source world for good reason: it provides many of the same capabilities as larger commercial databases for free. In its current form, MySQL is performant and feature-rich.

#### Using NodeDB

The node-db module provides a native code interface to popular database systems, including MySQL, using a common API that the module exposes to Node. Although node-db supports more than just MySQL, this section focuses on using MySQL in your application code. Since Oracle’s purchase of Sun Microsystems, the future of MySQL and its community has come under much speculation. Some groups advocate moving to a drop-in replacement such as MariaDB or switching to a different relational database management system (RDBMS) entirely. Although MySQL isn’t going away anytime soon, you need to decide for yourself whether it will be the right choice of software for your work.

* Mongoose

##### Node has a tremendous base of support for Mongo through its Mongoose library. Compared to the native drivers, Mongoose is an expressive environment that makes models and schemas more intuitive.

##### Defining schemas

When you use MongoDB, you don’t need to define a data schema as you would with a relational database. Whenever requirements change or you need to store a new piece of information, you just save a new record containing the information you need, and you can query against it immediately. You can transform old data to include default or empty values for the new field, but MongoDB does not require that step.

Even though schemas aren’t important to MongoDB, they are useful because they help humans understand the contents of the database and implicit rules for working with domain data. Mongoose is useful because it works using human-readable schemas, providing a clean interface to communicate with the database.

What is a schema? Many programmers tend to think in terms of models that define data structures, but don’t think much about the underlying databases those models represent. A table inside an SQL database needs to be created before you can write data to it, and the fields inside that table probably closely match the fields in your model. The schema—that is, the definition of the model inside the database—is created separately from your program; therefore, the schema predates your data.

MongoDB—as well as the other NoSQL datastores—is often said to be schemaless because it doesn’t require explicitly defined structure for stored data. In reality, MongoDB does have a schema, but it is defined by the data as it gets stored. You may add a new property to your model months after you begin work on your application, but you don’t have to redefine the schema of previously entered information in order to search against the new field.

#### Sequelize

Sequelize is an object relational mapper (ORM) that takes much of the repetition out of the tasks performed in the preceding sections. You can use Sequelize to define objects shared between the database and your program, then pass data to and from the database using those objects rather than writing a query for every operation. This becomes a major time-saver when you need to perform maintenance or add a new column, and makes overall data management less error-prone. Sequelize supports installation using npm:

Sequelize differs from the other libraries shown in this chapter in that it is based on a listener-driven architecture, rather than the callback-driven architecture used elsewhere. This means that you have to listen for both success and failure events after each operation, rather than having errors and success indicators returned with the operation’s results.

## Connection Pooling

Production environments are often composed of multiple resources: web servers, caching servers, and database servers. The database is typically hosted on a separate machine from the web server, allowing horizontal growth of the public-facing website without the need for setting up and configuring complex database clusters. Application developers must therefore be aware of the performance implications in accessing resources and how those access costs affect their site’s performance.

Connection pooling is an important concept in web development because the performance cost of establishing a database connection is relatively high; creating one or more new connections for every request creates an unnecessary burden on a heavily trafficked site and will contribute to weaker performance. The solution is to maintain database connections inside a cache pool after they are no longer needed, so they can be used immediately by the next incoming request.

Many database drivers provide pooling functionality, but that pattern goes against Node’s “one module, one purpose” philosophy. Instead, Node developers should usethe generic-pool module in front of their data layer to serve new database connections (see [Example 6-34](http://chimera.labs.oreilly.com/books/1234000001808/ch06.html#example6-34)). generic-pool will reuse connections where possible to prevent the overhead of creating new database connections, and the module can be used with any data library.

## MQ Protocols

By way of example, consider a generic user registration process. When a user registers herself, the application saves a new record in the database, sends an email to that user, and perhaps records some statistics about the registration process, such as the number of steps completed or amount of time taken. It probably doesn’t make sense to perform all of those actions right away when the user hits the Submit button on your web page. For one thing, the email process could take several seconds (or if you’re unlucky, minutes) to complete, the database call may not need to finish before the user is welcomed, and the statistics are probably separate from your main application flow. In this case, you might choose to generate a message that notifies other parts of your application instead—perhaps running on a different machine entirely—that a user has registered. This is known as a publish-subscribe pattern.

Message queues allow programmers to publish events and move on, enabling improved performance through parallel processing and higher levels of scalability through inter-process communication channels.

### RabbitMQ

RabbitMQ is a message broker that supports the advanced message queueing protocol (AMQP). It is useful in situations where data needs to be communicated between different servers, or between different processes on the same server. Written in Erlang, RabbitMQ is capable of clustering for high availability, and is fairly straightforward to install and begin using.

RabbitMQ communicates using the standardized protocol AMQP. AMQP comes from the financial services industry, where reliable messaging is a matter of life or death. It provides a vendor-neutral and abstract specification for generic (not just financial) middleware messaging and is intended to solve the problem of communicating between different types of systems. AMQP is conceptually similar to email: email messages have specifications for headers and format, but their contents can be anything from text to photos and video. Just as two companies don’t need to run the same email server software to communicate, AMQP allows messaging between different platforms. For example, a publisher written in PHP can send a message to a consumer written in JavaScript.

## Express

Express, an MVC framework for Node, is probably the most widely used Node module. It was inspired by the Sinatra framework for Ruby and enables a lot of features that make it very easy to throw together a website with Node.

Express works by defining page handlers for routes. The routes can be as simple as a path, or much more complex. The handlers could be as simple as emitting “Hello, world” or as complex as a whole page-rendering system that interacts with a database.