Slide 1

Node Js was initially written by Ryan Dahl.

It was thirteen years after the introduction of the first server-side JavaScript environment.

That is Netscape’s LiveWireProWeb.

The initial release was only supported by Linux and MAC OS X.

It was later sponsored by Joyent

Slide 2

Node Js is an open source technology and it has huge community online.

A popular misconception goes, that it is a programming language or a framework like ASP.NET, but

It is a cross-platform JavaScript runtime environment, remember it is a JS runtime environment which is built on Chrome’s V8 Engine.

It uses an event-driven Non-Blocking model that makes it lightweight and efficient

Node Js represents JavaScript everywhere paradigm , unifying web apps development around a single programming language rather than different language for Server and Client Side

Unlike certain apps, Node Js applications are single-threaded.

Slide 3:

Now, coming to the versions of Node Js, major releases are cut from the master branch of github repository every Month. Check the link below. The recent release is 12.10.0.

We can move to the official site of Node to check the stable releases and latest releases.

Slide 4:

So, In a nutshell, the basic functionality of Node Js comprises of the following:

1. It is a platform which allows us to run JavaScript on a computer/server
2. Read, Delete and Update the file system
3. It can be used to easily communicate with both SQL and Non-SQL Databases

Slide 5:

* Advantages of Node Js

1. As I said earlier, It is very lightweight and fast
2. It is highly customizable and easy to configure, we will get to the installation part shortly.
3. And it comes with Node Package Manager which is popularly called (NPM) that contains 140,000+ packages that are available for free
4. It allows one to build the application in JS “top-to-bottom”, even down to DB level even if NoSQL DataBase that stores objects in JSON(like MongDB and Firebase) is used. This makes development easier
5. It allows code to be reused across the client-side and server-side of the application.

Slide 6:

There are several companies who prefers Node Js to make their websites and applications. Like

NETFLIX

At netflix the whole user interface is built with Node, and that is why it is so fast.

UBER

If you remember,

At the end of a ride, the complete fare is automatically billed to the customer’s credit card. As one of the first companies that put Node.js into full production, Uber has built its massive matching system on Node.js

Linked In(Linked Mobile Development Lead Said) His reason to use node :

“One reason was scale. The second is, if you look at Node, the thing it’s best at doing is talking to other services.”

If all of these do not make much sense now, sit tight, It will get clear as we dive further into the course.

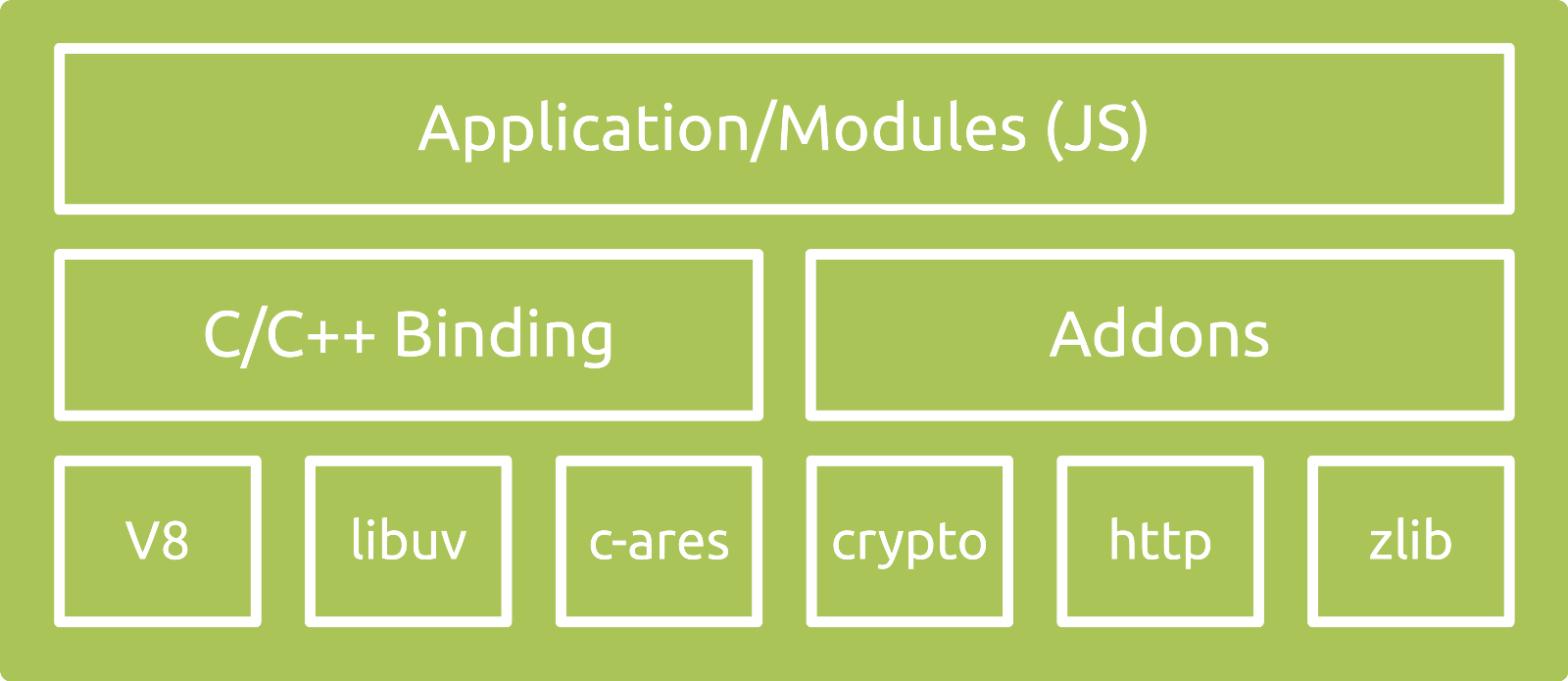
**Architecture:**

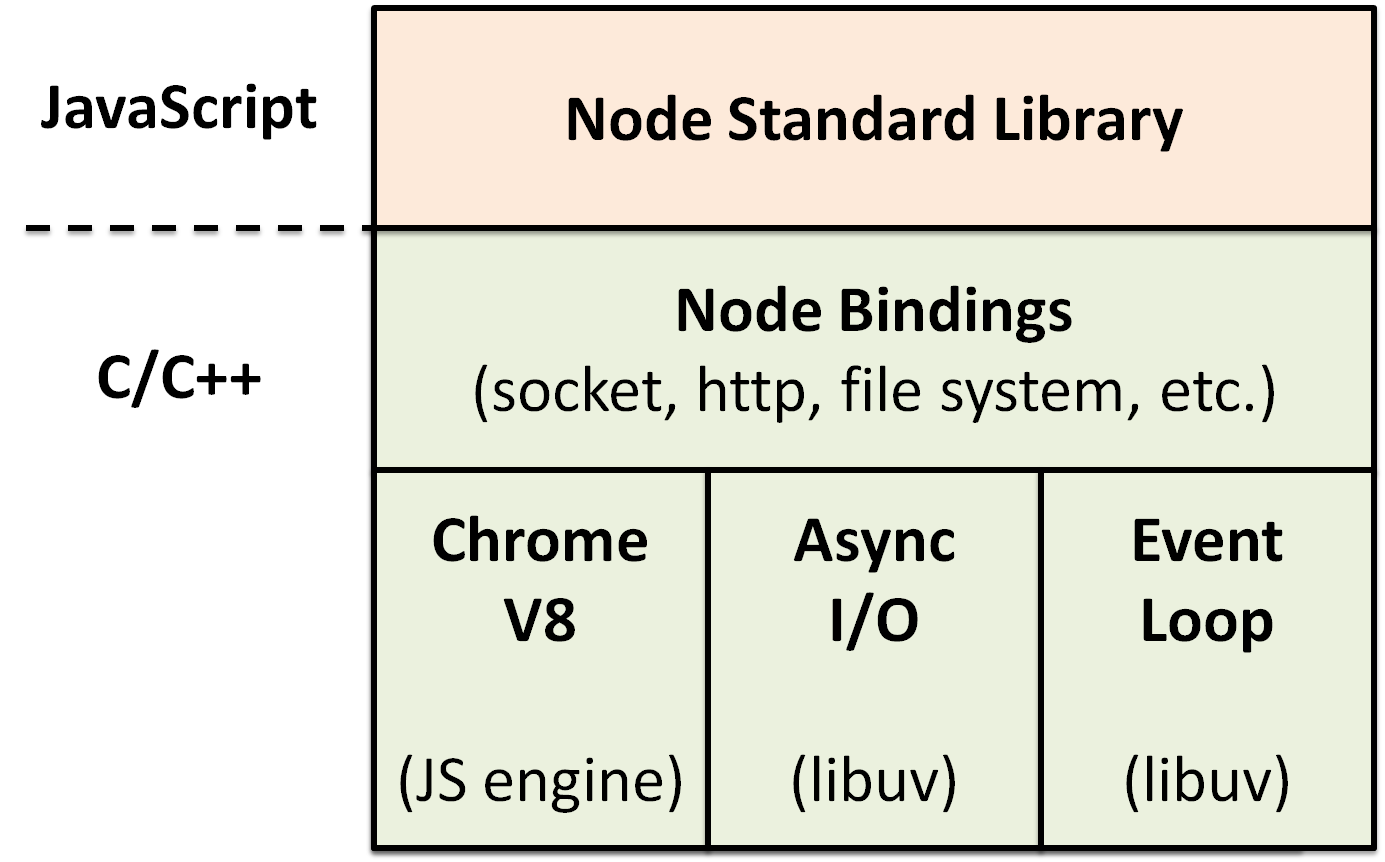
In this module I will first give an analogy of how Node JS works and then a brief explanation of the various components in action behind Node Js, which will help us in understanding the workflow of Node Js.

The official doc isn’t very helpful at explaining what Node Js is:

a JavaScript runtime built on Chrome’s V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model .…..

In order to understand this statement and the actual power behind it, let’s break down Node.js’ components, elaborate on some key terms, and then explain how different pieces interact with one another to make Node.js the powerful runtime it is:





Analogy:

Various Components in Node Js:

**V8:** V8 is a **JavaScript engine** built at the google development center, in Germany. It is [**open source**](https://code.google.com/p/v8/wiki/Source) and written in **C++**. It is used for both client side (Google Chrome) and server side (node.js) JavaScript applications. V8 was first designed to increase the performance of the JavaScript execution inside web browsers. In order to obtain speed, V8 translates JavaScript code into more efficient machine code instead of using an interpreter. It compiles JavaScript code into machine code at execution by implementing a **JIT (Just-In-Time) compiler** like a lot of modern JavaScript engines such as SpiderMonkey or Rhino (Mozilla) are doing. The main difference with V8 is that it doesn’t produce bytecode or any intermediate code.

**Libuv**: libuv is originally developed to provide asynchronous I/O that includes asynchronous TCP & UDP sockets, (famous) event loop, asynchronous DNS resolution, file system read/write, and etc. libuv is written in C.

**Application** - here is your code, modules, and Node.js' [built in modules](https://github.com/nodejs/node/tree/master/lib), written in JavaScript (or compiled to JS through TypeScript, CoffeeScript, etc.)

**Binding** - you probably have noticed by this time that Node.js is written in both JavaScript and C/C++. The reason that there are so many C/C++ code/libraries is simple: they are fast. However, how is it possible that the code you write in JavaScript end up communicating smoothly with code written in C/C++? Aren’t they three difference programming languages? Yes they are. And normally code written in different languages cannot be communicate with each other. Not without bindings. Bindings, as the name implies, are glue codes that “bind” one language with another so that they can talk with each other. In this case (Node.js), bindings simply expose core Node.js internal libraries written in C/C++ (c-ares, zlib, OpenSSL, http-parser, etc.) to JavaScript. One motivation behind writing bindings is code reuse: if a desired functionality is already implemented, why write the entire thing again, just because they are in different languages? Why not just bridge them? Another motivation is performance: system programming languages such as C/C++ are generally much faster than other high-level languages (e.g. Python, JavaScript, Ruby, etc.). Therefore it might be wise to designate CPU-intensive operations to codes written in C/C++, for example.

**C/C++ Addons**: Bindings only provide glue code for Node.js’ core internal libraries, i.e. zlib, OpenSSL, c-ares, http-parser, etc. If you want to include a third-party or your own C/C++ library in your application, you would have to write the glue code for that library yourself. These glue code you write are called addons. Think of bindings and addons as **bridges** between your JavaScript code and Node.js’ C/C++ code.

[**Other C/C++ Components/Dependencies**](https://nodejs.org/en/docs/meta/topics/dependencies/): such as [c-ares](http://c-ares.haxx.se/), [crypto (OpenSSL)](https://www.openssl.org/), [http-parser](https://github.com/nodejs/http-parser), and [zlib](http://zlib.net/). These dependencies provide low-level interactions with servers to establish important functionalities such as networking, compressing, encrypting, etc.

Some Concepts:

Ex:-

**Event Driven Programming Language:**

Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.

Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern. The functions that listen to events act as **Observers**. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners as follows –

**What the hell are callbacks?**

In a synchronous program, you would write something along the lines of:

function processData () {

var data = fetchData ();

data += 1;

return data;

}

This works just fine and is very typical in other development environments. However, if fetchData takes a long time to load the data (maybe it is streaming it off the drive or the internet), then this causes the whole program to 'block' - otherwise known as sitting still and waiting - until it loads the data. Node.js, being an asynchronous platform, doesn't wait around for things like file I/O to finish - Node.js uses callbacks. A callback is a function called at the completion of a given task; this prevents any blocking, and allows other code to be run in the meantime.

The node.js way to deal with the above would look a bit more like this:

function processData (callback) {

fetchData(function (err, data) {

if (err) {

console.log("An error has occurred. Abort everything!");

return callback(err);

}

data += 1;

callback(data);

});

}

At first glance, it may look unnecessarily complicated, but callbacks are the foundation of Node.js. Callbacks give you an interface with which to say, "and when you're done doing that, do all this." This allows you to have as many IO operations as your OS can handle happening at the same time. For example, in a web server with hundreds or thousands of pending requests with multiple blocking queries, performing the blocking queries asynchronously gives you the ability to be able to continue working and not just sit still and wait until the blocking operations come back. This is a major improvement.

The typical convention with asynchronous functions (which almost all of your functions should be):

function asyncOperation ( a, b, c, callback ) {

// ... lots of hard work ...

if ( /\* an error occurs \*/ ) {

return callback(new Error("An error has occurred"));

}

// ... more work ...

callback(null, d, e, f);

}

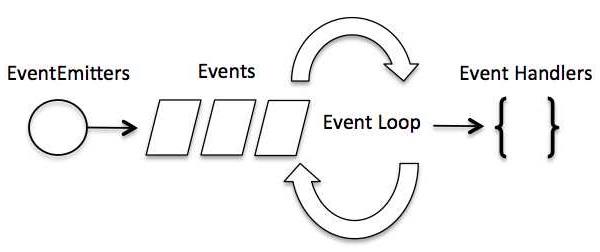
asyncOperation ( params.., function ( err, returnValues.. ) {

//This code gets run after the async operation gets run

});

You will almost always want to follow the [error callback convention](https://nodejs.org/en/knowledge/errors/what-are-the-error-conventions), since most Node.js users will expect your project to follow them. The general idea is that the callback is the last parameter. The callback gets called after the function is done with all of its operations. Traditionally, the first parameter of the callback is the error value. If the function hits an error, then they typically call the callback with the first parameter being an Error object. If it cleanly exits, then they will call the callback with the first parameter being null and the rest being the return value(s).

Event Emittor

****

**Event Emittor:-**

Many objects in a Node emit events, for example, a net.Server emits an event each time a peer connects to it, an fs.readStream emits an event when the file is opened. All objects which emit events are the instances of events.EventEmitter.

## **EventEmitter Class**

As we have seen in the previous section, EventEmitter class lies in the events module. It is accessible via the following code −

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple properties like **on** and **emit**. **on** property is used to bind a function with the event and **emit** is used to fire an event.

## **Methods**

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **addListener(event, listener)**  Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 2 | **on(event, listener)**  Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 3 | **once(event, listener)**  Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained. |
| 4 | **removeListener(event, listener)**  Removes a listener from the listener array for the specified event. **Caution −** It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained. |
| 5 | **removeAllListeners([event])**  Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained. |
| 6 | **setMaxListeners(n)**  By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited. |
| 7 | **listeners(event)**  Returns an array of listeners for the specified event. |
| 8 | **emit(event, [arg1], [arg2], [...])**  Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise. |

## **Class Methods**

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **listenerCount(emitter, event)**  Returns the number of listeners for a given event. |

## **Events**

|  |  |
| --- | --- |
| **Sr.No.** | **Events & Description** |
| 1 | **newListener**   * **event** − String: the event name * **listener** − Function: the event handler function   This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event. |
| 2 | **removeListener**   * **event** − String The event name * **listener** − Function The event handler function   This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event. |

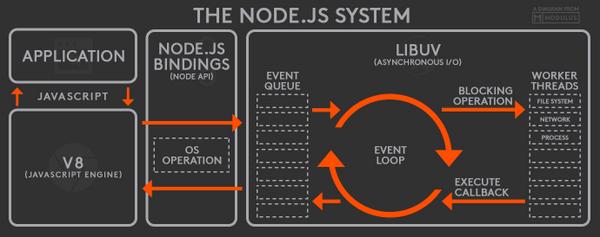
PUTTING EVERYTHING TOGETHER

Now that we have a high-level overview of Node.js’ components, we’ll investigate its workflow to get a better sense of its architecture and how different components interact with one another.

When a Node.js application starts running, the V8 engine will run the application code you write. Objects in your application will keep a list of observers (functions registered to events). These observers will get notified when their respective events are emitted.

When an event is emitted, its callback function will be enqueued into an**event queue**. As long as there are remaining events in the queue, the **event loop** will keep dequeuing events in the queue and putting them onto the**call stack**. It should be noted that only when the previous event is processed (the call stack is cleared) will the event loop put the next event onto the call stack.

On the call stack, when an I/O operation is encountered, it will be handed over to libuv for processing. By default, libuv maintains a thread pool of four worker threads, although the number can be altered to add more threads. Depending on the type of I/O operation, the worker threads will interact with Node.js’ low-level libraries to perform operations such as database transaction, file system access, networking, etc. When the processing is over, libuv will enqueue the event back into the event queue again for the main tread to work on. During the time that libuv handles asynchronous I/O operations, the main thread does not wait for the outcome of processing, but moves on instead. The event returned by libuv will have the opportunity to be handled by the main thread again when it is put back onto the call stack by the event loop.This completes a the life cycle of an event in a Node.js application.



Rest Analogy

[mbq](http://stackoverflow.com/users/370756/mbq) once made an [excellent analogy](http://stackoverflow.com/a/3491931/4603550) between Node.js and a restaurant. I will borrow and modify his example to make the Node.js cycle easier to understand. Think of a Node.js application as a Starbucks cafe. A highly efficient and well-trained waiter (the one and only main thread) will take order. When a large number of customers visits the cafe at the same time, they will wait in line (enqueued in the event queue) to be served by the waiter. Once a customer is served by the waiter, he passes the order to a manager (libuv), who assigns each order to a barista (worker thread). The barista will use different ingredients and machines (low-level C/C++ components) to make different kinds of drinks, depending on the customers’ requests. Typically there will be four baristas on duty (tread pool). However, when the peak hits, more baristas can be called to work (however this should be done at the beginning of the day, NOT during lunch break for example). Once the waiter passes the order to the manager, he does not wait for the coffee to be made to serve another customer. Instead, he calls the next customer (the next event dequeued by the event loop and pushed onto the call stack). You can think of an event currently on the call stack as a customer at the counter being served. \_When the coffee is done, the coffee will be sent to the end of the customer line. The waiter will call out the name when the coffee moves to the counter, and the customer will get his coffee. \_(The underlined part of the analogy sounds weird in real life; however when you consider the process from a program’s perspective, it makes sense.)

**Node is Non-Blocking Asynchronous, single-threaded application.**

Let us understand it with an analogy.

Imagine you go to a restaurant. There is a single waiter who takes Orders for food. The customer at Table 1 gives an order, the waiter takes the order and goes to the kitchen and asks the chef to cook it, the waiter waits there until the food is prepared. Meanwhile, the customer at table 2 also wants to order, but the waiter is in the kitchen and unable to serve his request so the table2 customer gets starved. Once the food is ready, the waiter brings the food to Table1 and then takes the order from Table 2 and this cycle continues.

So in this example, the waiter is idle in the kitchen which means this architecture is not utilizing the full power of the CPU

This is called Synchronous

**What is Synchronous Programming?**

In most traditional programming practice , most I/O operation occurs synchronously. That generally means that your flow of program will wait until the I/O operation is completed. If we take up an austere situation where you need to read a big file, write that in your database and then send http request to a server to display something. All these operations are expensive and time consuming. So if reading the file and writing it to DB takes time, the HTTP request job scheduled for later gets starved.

Imagine another scenario, where the waiter takes the order and gives it to the kitchen, and when the food is being prepared, he comes back and wait for the second order, once table 2 places the order, the waiter takes it to the kitchen and add it in his note. Once the Table 1 order gets prepared, the waiter will bring the order for Table 1 and the chef starts preparing the next order in the note.

This is called **Asynchronity**

**Asynchronity of Node.**

Well to understand this let us reconsider the above scenario. To utilize the CPU while it is sitting idle, we must consider threads.To handle the above situation we need to bifurcate our jobs into threads, so they work in parallel and our program don’t have to wait for the I/O to complete and reading our file and making our HTTP request occurs in parallel

Node on the other hand has a single thread. It has a mother thread and an event queue. Any request from the server gets stacked up in the queue and executes accordingly. So, when node stops for the I/O operation to complete, unlike our traditional languages like Java, it fetches the very next job from the queue and starts executing it. In this model, the CPU is not sitting idle and we utilizing 100% of CPU performance.

Don’t worry if all this makes little sense to you now, as we dive deep into the topic, you will have a better clarity by the end.