Develop a Reliable Backend with Node and Express

File System and Database Connectivity in Node.js

A Day in the Life of a MERN Stack Developer

Joe is working as a MERN Stack Developer. His company acknowledges and appreciates his commitment and progress, as he has diligently completed all the projects assigned to him.

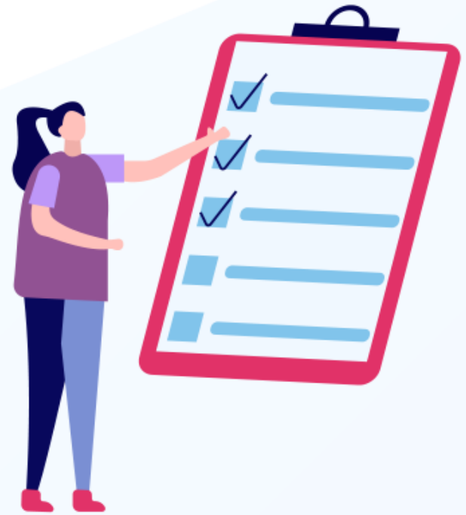
He has now been assigned to an important project and asked to build various operations on Node.js files, such as read, write, and delete, to save time and effort.

In this lesson, Joe will learn how to solve this real-world scenario effectively and quickly.

Learning Objectives

By the end of this lesson, you will be able to:

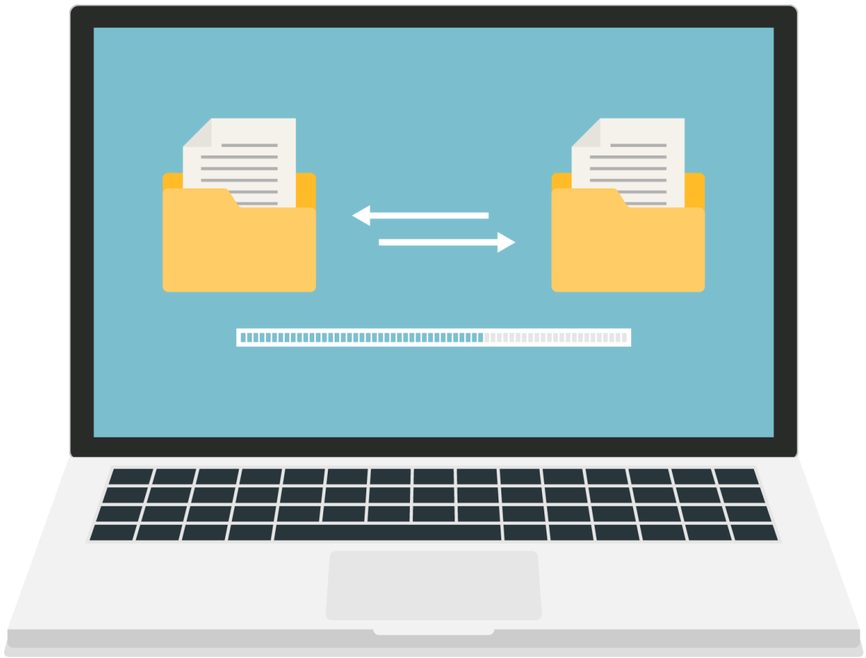
- Implement file system module methods
- Analyze debugging in Node.js
- Configure and create a database connection
- Develop a child process in Node.js



File System

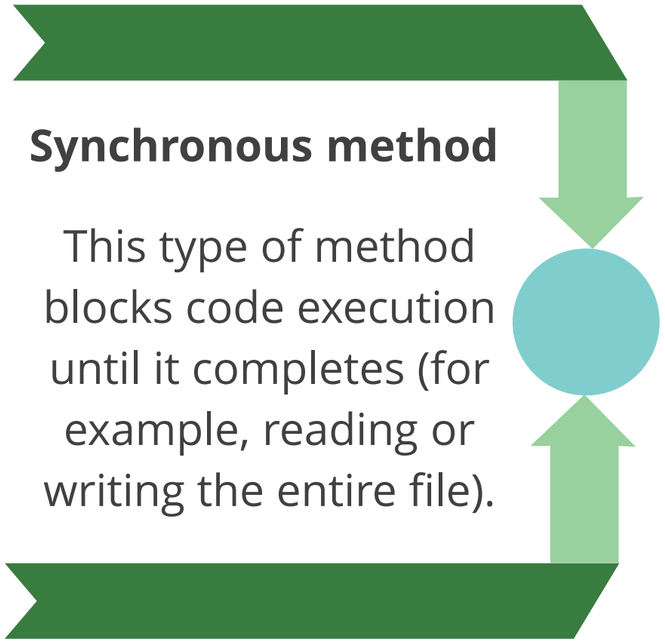
What Is fs Module?

In Node.js, the **fs** (file system) module is used to perform various operations related to file manipulation, reading, and writing.



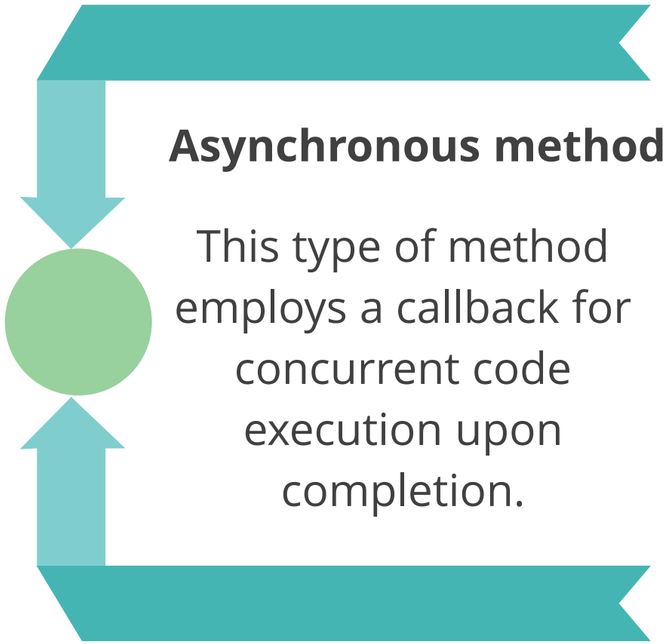
The file can be read and written in node.js in both synchronous and asynchronous ways.

What Are Synchronous and Asynchronous Methods?



Synchronous method

This type of method blocks code execution until it completes (for example, reading or writing the entire file).



Asynchronous method

This type of method employs a callback for concurrent code execution upon completion.

Methods in a File System

Some common operations in file handling in Node.js:

- Checking file or directory existence
- Creating a directory
- Reading a directory
- Reading a file
- Writing to a file
- Appending to a file

Reading a File

The **fs.readFile()** method in Node.js is used to asynchronously read the contents of a file.

```
const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

Reading a File

The **fs.readFileSync()** method in Node.js is used to synchronously read the contents of a file.

```
const fs = require('fs');

try {
  const data = fs.readFileSync('example.txt', 'utf8');
  console.log(data);
} catch (err) {
  console.error(err);
}
```

Writing to a File

The **fs.writeFile()** method in Node.js is used to asynchronously write data to a file.

```
const fs = require('fs');

fs.writeFile('example.txt', 'Hello, Node.js!', 'utf8', (err) =>
{
  if (err) {
    console.error(err);
    return;
  }
  console.log('File written successfully!');
});
```

Writing to a File

The **fs.writeFileSync()** method in Node.js is used to synchronously read the contents of a file.

```
const fs = require('fs');

try {
  fs.writeFileSync('example.txt', 'Hello, Node.js!', 'utf8');
  console.log('File written successfully!');
} catch (err) {
  console.error(err);
}
```

Appending to a File

The **fs.appendFile()** method in Node.js is used to asynchronously append data to a file.

```
const fs = require('fs');

fs.appendFile('example.txt', '\nAppended Text', 'utf8', (err)
=> {
  if (err) {
    console.error(err);
    return;
  }
  console.log('Data appended to file!');
});
```

Appending to a File

The **fs.appendFileSync()** method in Node.js is used to synchronously append data to a file.

```
const fs = require('fs');

try {
  fs.appendFileSync('example.txt', '\nAppended Text', 'utf8');
  console.log('Data appended to file!');
} catch (err) {
  console.error(err);
}
```

Reading a Directory

The **fs.readdir()** method in Node.js is used to asynchronously read the contents of a directory.

```
const fs = require('fs');

fs.readdir('.', 'utf8', (err, files) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('Files in the current directory:', files);
});
```

Reading a Directory

The **fs.readdirSync()** method in Node.js is used to synchronously read the contents of a directory.

```
const fs = require('fs');

try {
  const files = fs.readdirSync('.', 'utf8');
  console.log('Files in the current directory:', files);
} catch (err) {
  console.error(err);
}
```

Creating a Directory

The **fs.mkdir()** method in Node.js is used to asynchronously create a directory.

```
const fs = require('fs');

fs.mkdir('newDirectory', (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('Directory created successfully!');
});
```

Creating a Directory

The **fs.mkdirSync()** method in Node.js is used to synchronously create a directory.

```
const fs = require('fs');

try {
  fs.mkdirSync('newDirectory');
  console.log('Directory created successfully!');
} catch (err) {
  console.error(err);
}
```

Checking If a File or Directory Exists

The **fs.access()** method in Node.js is used to asynchronously check the accessibility of a file or directory.

```
const fs = require('fs');

fs.access('example.txt', fs.constants.F_OK, (err) => {
  if (err) {
    console.error('File does not exist!');
    return;
  }
  console.log('File exists!');
});
```

Checking If a File or Directory Exists

The **fs.accessSync()** method in Node.js is used to synchronously check the accessibility of a file or directory.

```
const fs = require('fs');

try {
  fs.accessSync('example.txt', fs.constants.F_OK);
  console.log('File exists!');
} catch (err) {
  console.error('File does not exist or is not accessible.');
}
```

File System to Perform Operations

Problem Statement: **Duration: 20 min.**

You need to apply file system commands to perform various operations on Node.js file.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Read files into a buffer using fs.readFile() method
2. Write files
3. Delete files

EventEmitter Class

The EventEmitter class is used to create and manage custom events.

It is defined by the node:events module.

Syntax to import the events module:

```
const EventEmitter = require('node:events');
```

EventEmitter Class: Methods

EventEmitter class in Node.js has two methods, **on** and **emit**.

emit method:

- Emit method helps to trigger events
- It adds a listener at the end of the listeners array for the specified event

on method:

- On method helps to add a callback function when an event occurs
- It executes each listener in a sequential order, providing them with the specified arguments.

EventEmitter: Example

EventEmitter instance emits a **newListener** event before adding a listener.

```
const EventEmitter = require('node:events');
class EmitterDemo extends EventEmitter {
    //..
}
const emitter = new EmitterDemo();

emitter.once('newListener', (event, listener) =>{
  if (event === 'notify') {
    emitter.on('notify', () => {
      console.log('Notification Received Once');
    });
  }
});
```

```
emitter.on('notify', () => {
  console.log('Notification
Received');
});

emitter.emit('notify');
```

EventEmitter Class

The EventEmmitter class raises and binds events by:

01 Returning EventEmitter from a function

02 Extending the EventEmitter class

Return an EventEmitter

Code to return an emitter from a function:

Syntax:

```
var emitter = require('events').EventEmitter;
function getEventEmitter(number) {
   var emtrVar = new emitter();
    setTimeout(function () {
        for (var i = 1; i <= number; i++) {
           emtrVar.emit('BeforeProcess', i);
            console.log('Processing number:' + i);
            emtrVar.emit('AfterProcess', i);
        }
    }
    , 2000)
```

Return an EventEmitter

Syntax:

```
    return emtrVar;
}
var emtr = getEventEmitter(3);

emtr.on('BeforeProcess', function (data) {
    console.log('Starting the Process ' + data);
});

emtr.on('AfterProcess', function (data) {
    console.log('Processing Finsihed' + data);
});
```

Inheriting EventEmitter Class

Code to inherit an EventEmitter class:

Syntax:

```
const EventEmitter = require('node:events');

class EmitterDemo extends EventEmitter {
    //..
}

const emitter = new EmitterDemo();

emitter.on('greet', () => {
  console.log('Greet event has occurred');
});

emitter.emit('greet');
```

Types of Events

Listening events:
An event must register functions (callbacks) to listen to the events before emitting them.

Syntax:

```
eventEmitter.addListener(event, listener)
eventEmitter.on(event, listener)
```

Emitting events:
Every event is named in Node.js, which can trigger an event using the emit(event, [arg1], [arg2], [....]) function.

Syntax:

```
eventEmitter.emit(event, [arg1], [arg2], [....])
```

Perform Various Events in Node js

Problem Statement: **Duration: 20 min.**

You have been assigned a task to create and handle custom events in Node.js through EventEmitter class.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new project
2. Import events module
3. Create EventEmitter object
4. Work with error events
5. Implement async vs. sync event handling
6. Write code to handle events only once
7. Execute the code using the node command

Debugging Node.js Application

Debugging in Node

Debugging in the node is a non-graphic built-in Node.js utility.

It can be used on multiple platforms.

It provides commands to debug applications.

To use the debugging function, start node, inspect the argument, and follow the path to the script

```
node inspect myapplication.js
```

Debugging in Node: Example

```
(node:61113) ExperimentalWarning: AbortController is an experimental feature. This feature could
change at any time
(Use 'node --trace-warnings ...' to show where the warning was created)
< Debugger listening on ws://127.0.0.1:9229/379ed20f-480a-9c5c-7dc081d9ab64
< For help, see: https://nodejs.org/en/docs/inspector
<
connecting to 127.0.0.1:9229 ... ok
< Debugger attached.
<
Break on start in myapplication.js:1
> 1 const fs = require ('fs');
  2 const data = 'Search the candle rather than cursing the darkness';
  3 fs.writeFile('quotes.txt', data, (err) => {
  debug>
  debug>
```

Node.js Debugger

The debugger breaks the first executable line automatically.

To run till the breakpoint, set the NODE_INSPECT_RESUME_ON_START environment variable to 1.

Example:

```
NODE_INSPECT_RESUME_ON_START=1 node inspect myapplication.js
```

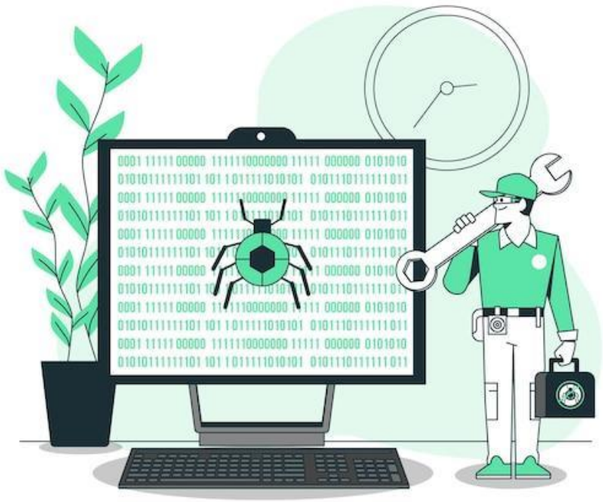
Node.js Debugger: Example

```
(node:61258) ExperimentalWarning: AbortController is an experimental feature. This feature could
change at any time
(Use 'node --trace-warnings ...' to show where the warning was created)
< Debugger listening on ws://127.0.0.1:9229/6c3c6b13-18d3-4c5b-b9e0-c3c07a50c298
< For help, see:https://nodejs.org/en/docs/inspector
<
connecting to 127.0.0.1:9229 ... ok
< Debugger attached.
<
< Data written to the file successfully.
<
< Waiting for the debugger to disconnect...
<
debug>
```

Watching Expressions and Variables

While debugging, it is possible to watch expression and variable values.

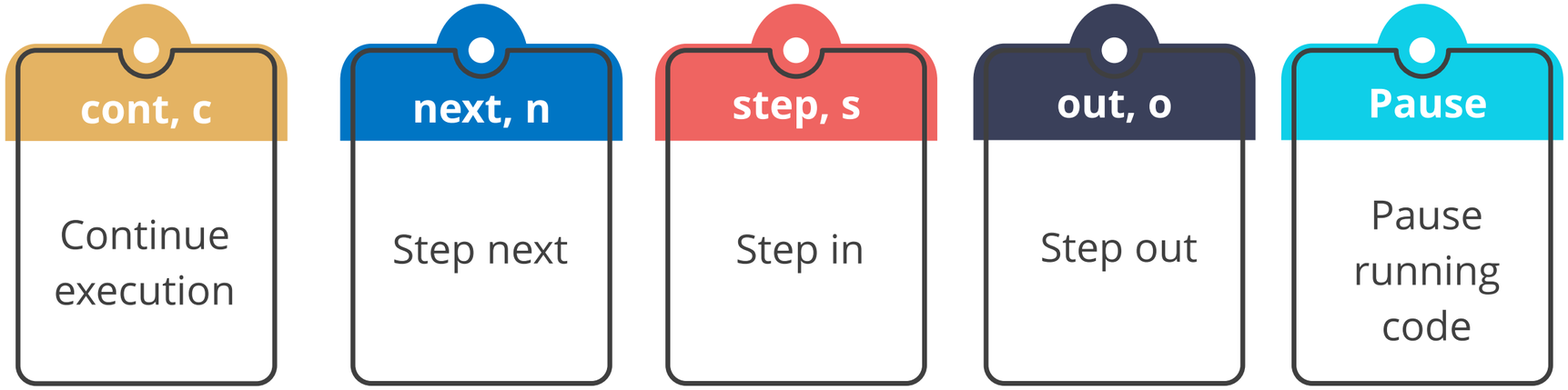
The command watchers will print the active watchers.



```
type watch('my_expression')
```

Stepping Commands in Debugger

Commands used for stepping in the debugger:



Breakpoints in Debugger

setBreakpoint(), sb()

Sets breakpoint on the current line

setBreakpoint(line), sb(line)

Sets breakpoint on a specific line

setBreakpoint('fn()'), sb(...)

Sets breakpoint on a first statement in the function's body

Breakpoints in Debugger

setBreakpoint('script.js', 1), sb(...)

Sets breakpoint on the first line of script.js

setBreakpoint('script.js', 1, 'num < 4'), sb(...)

Sets a conditional breakpoint on the first line of script.js

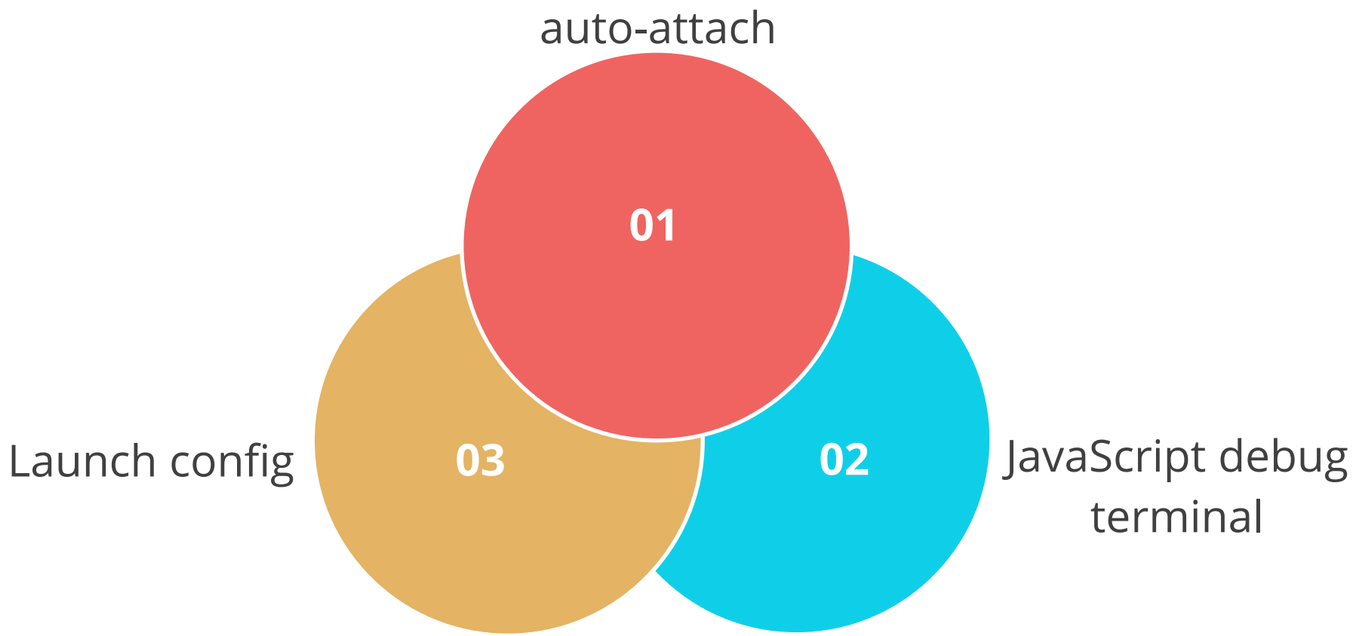
clearBreakpoint('script.js', 1), cb(...)

Clears breakpoint in script.js on line 1

Debugging with Visual Studio Code Editor

The Visual Studio Code editor offers inbuilt debugging.

Ways to debug programs in VS Code:



Ways to Debug with Visual Studio Code Editor

Some of the ways to debug with Visual Studio Code editor are:

01

Auto-attach
It is used to debug processes that are executed in VS Code's integrated terminal.

02

JavaScript debug terminal
It is used to debug using the integrated terminal.

03

Launch config
It is used to start a program or attach to a process outside VS Code editor.

Auto Attach: Example

Auto Attach is enabled using the **Toggle Auto Attach** command from the command palette (⇧⌘P) or the **Auto Attach Status bar** item if it's already activated.

```
1  const fs = require ('fs');
2  const data = 'Search the candle rather than cursing the darkness';
3  fs.writeFile('quotes.txt', data, (err) => {
4
5      if(err) {
6          throw err;
7      }
8
9      console.log('Data written to the file successfully.');
10 });
```

Debugging Using the VS Code Editor

Problem Statement: **Duration: 20 min.**

You have been assigned a task to debug the Node.js application using the Visual Studio Code editor.

Assisted Practice: Guidelines

Steps to be followed:

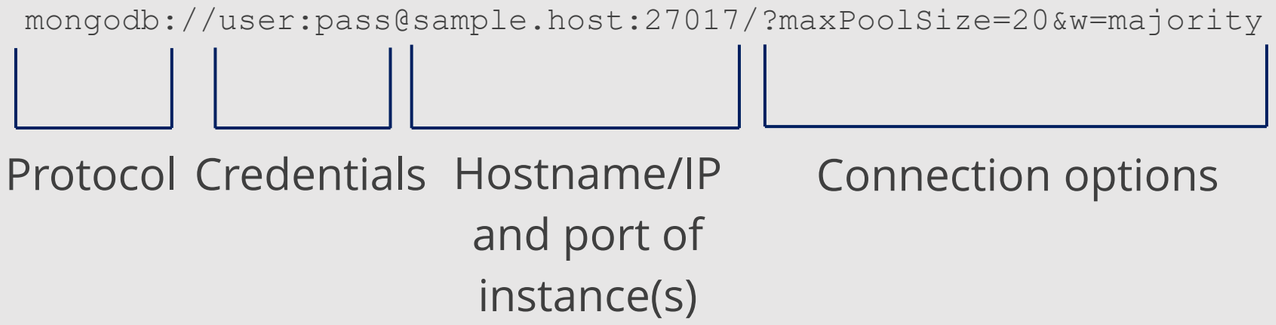
1. Understand the Visual Studio Code editor
2. Use the Auto Attach feature to debug
3. Create breakpoints

Database Connectivity

Connection String

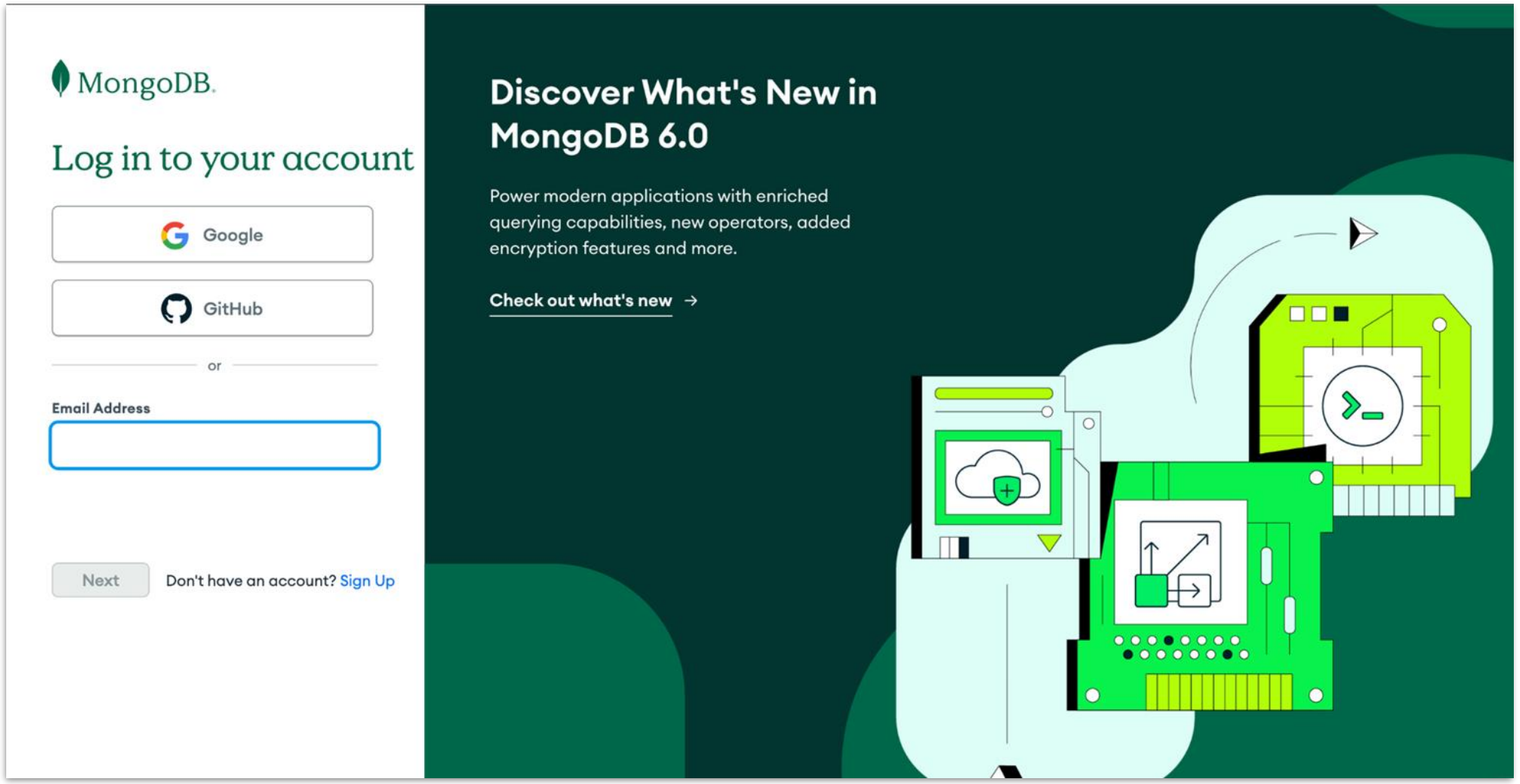
The connection string contains instructions for the driver on how to connect and interact with MongoDB.

Example:



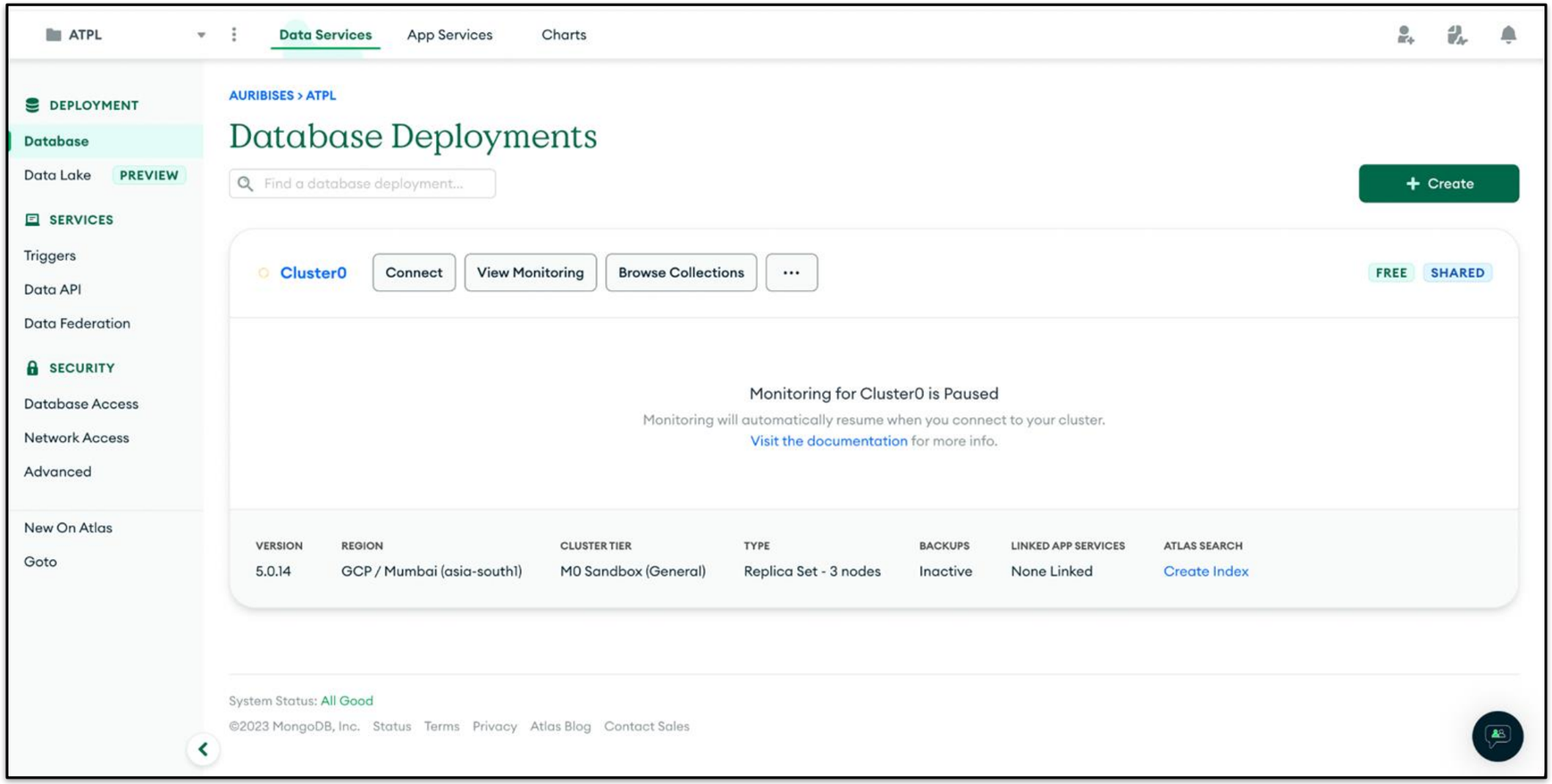
Create a Free MongoDB Atlas Cluster

Use Atlas, a fully-managed database-as-a-service from MongoDB to create a free MongoDB Atlas cluster and load the sample data.



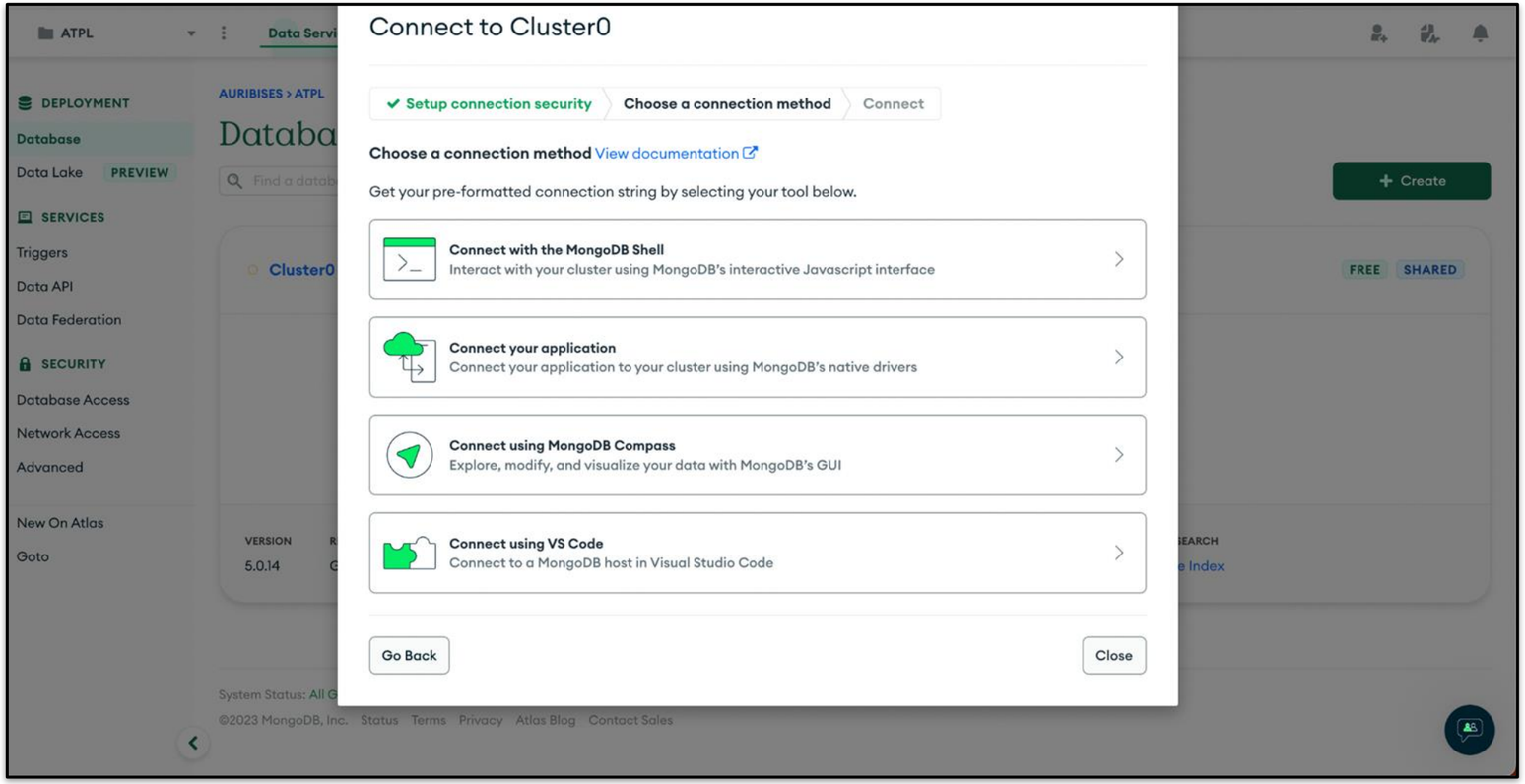
Create a Free MongoDB Atlas Cluster

Navigate to the user cluster in the MongoDB Atlas account



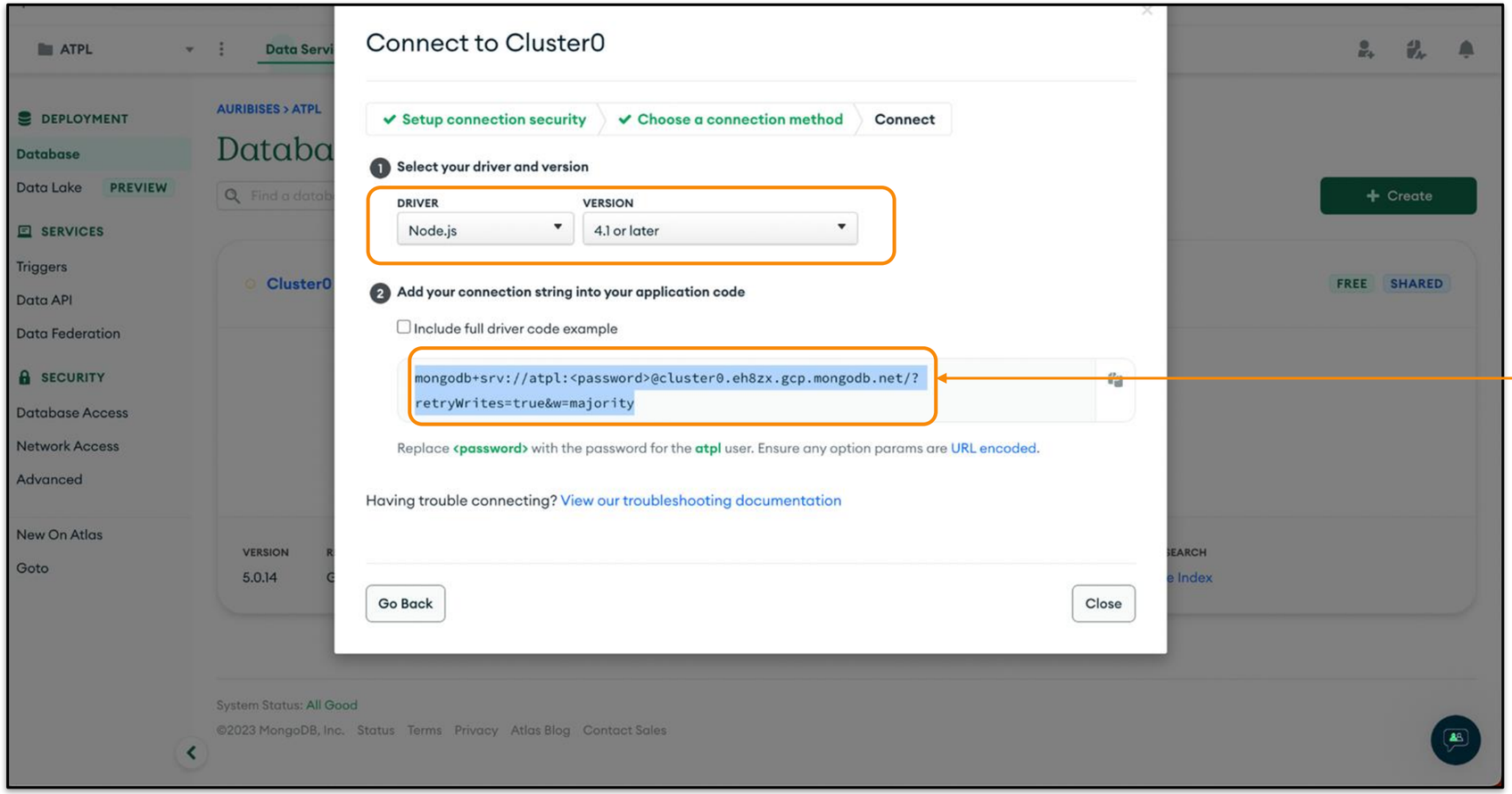
Create a Free MongoDB Atlas Cluster

The cluster connection wizard will show after clicking **CONNECT**.



Create a Free MongoDB Atlas Cluster

Next, when the wizard prompts to select the driver version, connect to user application and select Node.js version 4.1 or later



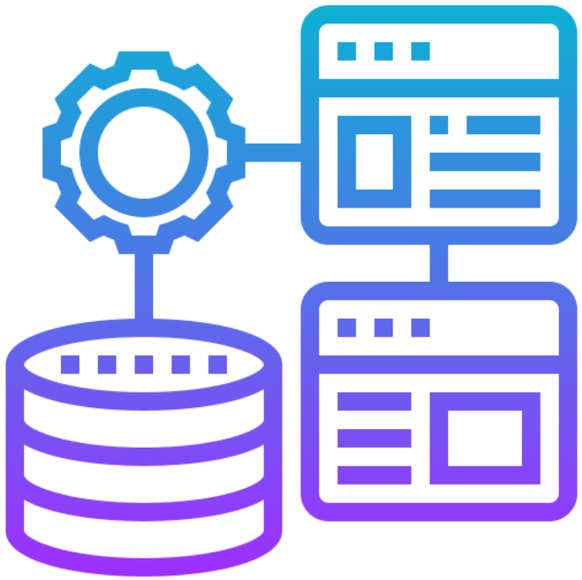
Copy and save the connection string URI for later use

Configuring MongoDB with Node.js

MongoDB database is connected to the Node.js application using MongoDB Node.js driver to execute the queries.

Command to install MongoDB NodeJS driver:

```
npm install mongodb
```



Steps to Connect and List the Database in Cluster

The MongoDB module exports MongoClient to connect with the MongoDB database

Syntax:

```
// Import MongoDB Driver
const { MongoClient, ServerApiVersion, listDatabases } = require('mongodb');

// Copy the URI from MongoDB Atlas with your username and password
Const uri =
"mongodb+srv://username:password@cluster0.eh8zx.gcp.mongodb.net/?retryWrites=true&w=majority";

// Create MongoDB Client Object
const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true, serverApi:
ServerApiVersion.vl });
```

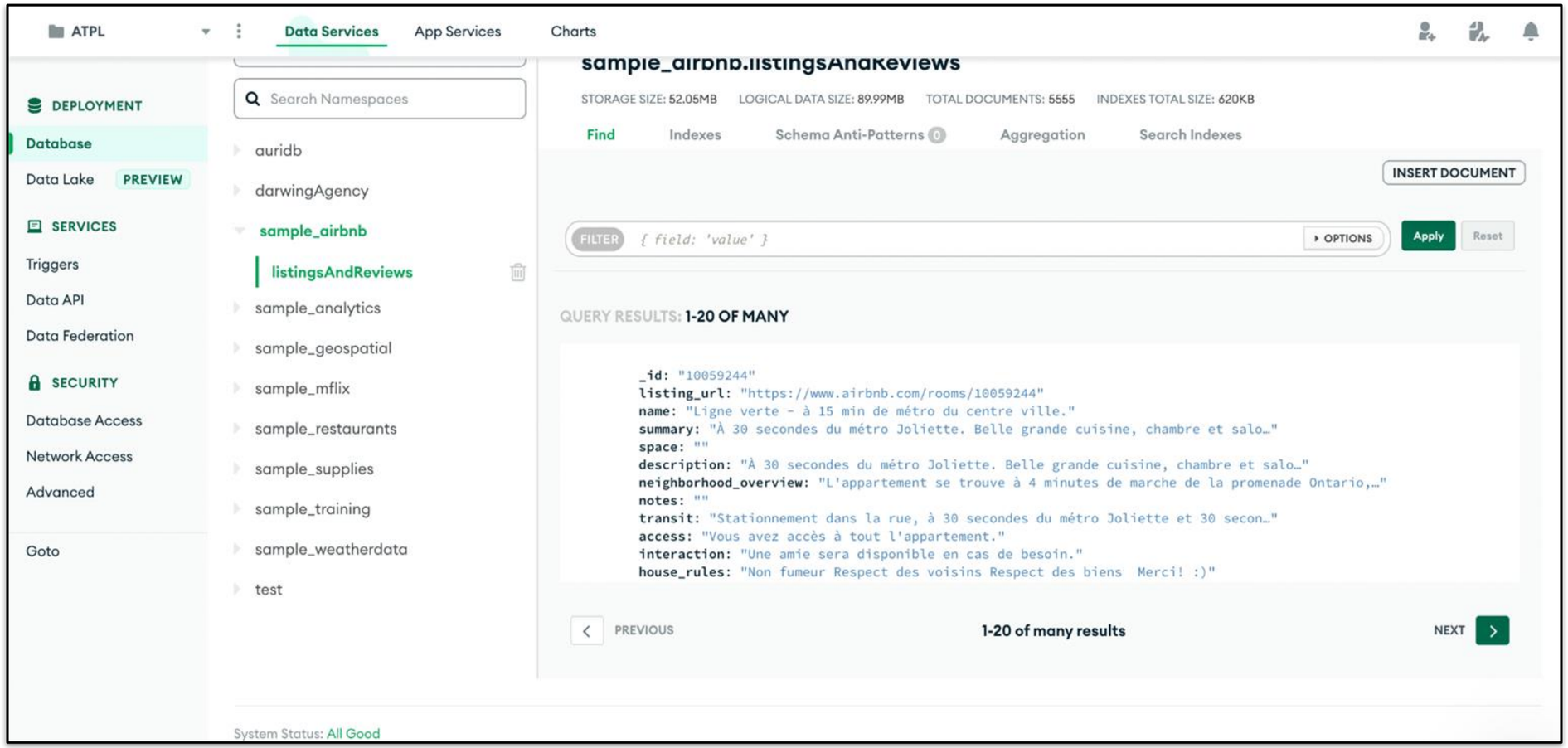
Steps to Connect and List the Database in Cluster (Contd.)

Syntax:

```
try {
 // Connect to MongoDB Cluster
 client. connect();
 console.log("Connection Established...");
} catch (e) {
 console.error(e);
} finally {
 client.close();
}
```

Working with Read Operation

Run name-based search on Airbnb listing in the **listingsAndReviews** collection in the sample database



Working with Read Operation

findOne() is used to query a single document by calling from a collection. It will return the first document that matches the given query.

Syntax:

```
findOne({ name: nameOfListing })
```

Working with Read Operation

An example code to fetch a single document:

Example:

```
async function fetchDocument(client, inputName) {
    const result = await client.db("sample_airbnb")
                               .collection("listingsAndReviews")
                               .findOne({ name: inputName });
    if (result) {
       console.log('Listing Found..');
       console.log(result);
    } else {
      console.log('No Listing Found’);
    }
}
fetchDocument(client, "Room Close to LGA and 35 mins to Times Square");
```

Working with Insert Operation

insertOne() is used to insert a single document into the collection.

The new document must have information about what kind of object is to be inserted into the collection. This is the required parameter.

Syntax:

```
insertOne(object)
```

If a new document does not create the _id field, then MongoDB drive will automatically create one.

Working with Insert Operation

An example code to create a document:

Example:

```
async function createDocument(client, inputDocument){
  const result = await client.db("sample_airbnb")
                .collection("listingsAndReviews")
                .insertOne(inputDocument);
  console.log('A new Document Created with ID: ${result.insertedId}’);
}
document = {
  name: "Home Sweet Home",
  summary: "A Penthouse in South America",
  bedrooms: 2,
  bathrooms: 3
}
createDocument(client, document);
```

Updating Records

updateOne() is used to update a single document in a collection.

It requires two parameters:

filter (object)

- The **filter** is used to select the document to update.
- Example:
  It is similar to **findOne().**

update (object)

- The **update** is used to update the selected documents.
- Example:
  db.dish.updateOne({name: "Veggie Noodles"},{$set:{price:250}})

Updating Records

An example code to update the record:

Example:

```
async function updateDocument(client, inputName, updatedName) {
  const result = await client.db("sample_airbnb").collection("listingsAndReviews")
            .updateOne({ name: inputName }, { $set: updatedName });
  console.log(`${result.modifiedCount} document(s) Updated.`);
}
updateDocument(client, "Room Close to LGA and 35 mins to Times Square", { bedrooms: 3, beds: 6 });
```

Working with Delete Operation

deleteOne() is used to delete a single document into the collection and requires a **filter of type object** to select the document to delete.

An example code to delete the document:

Example:

```
async function deleteDocument(client, inputName) {
  const result = await client.db("sample_airbnb").collection("listingsAndReviews")
       .deleteOne({ name: inputName });
  console.log(`${result.deletedCount} document(s) Deleted.');
}
deleteDocument(client, "Room Close to LGA and 35 mins to Times Square");
```

Database Connectivity

Problem Statement:

Duration: 20 min.

You have been assigned a task to work with Database CRUD Operations.

ASSISTED PRACTICE

Assisted Practice: Guidelines

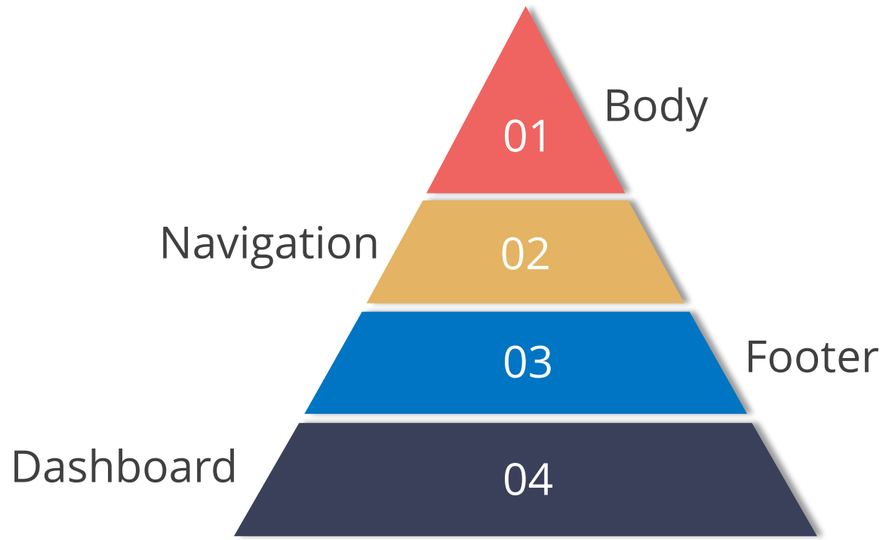
Steps to be followed:

1. Understand MySQL
2. Install MySQL Node.js driver
3. Configure MySQL database server connection
4. Close the database connection
5. Create MySQL tables using Node.js
6. Insert data in MySQL tables using Node.js
7. Query MySQL databases using Node.js
8. Modify MySQL tables with Node.js
9. Delete data from MySQL tables using Node.js

Template Engines

Template Engines

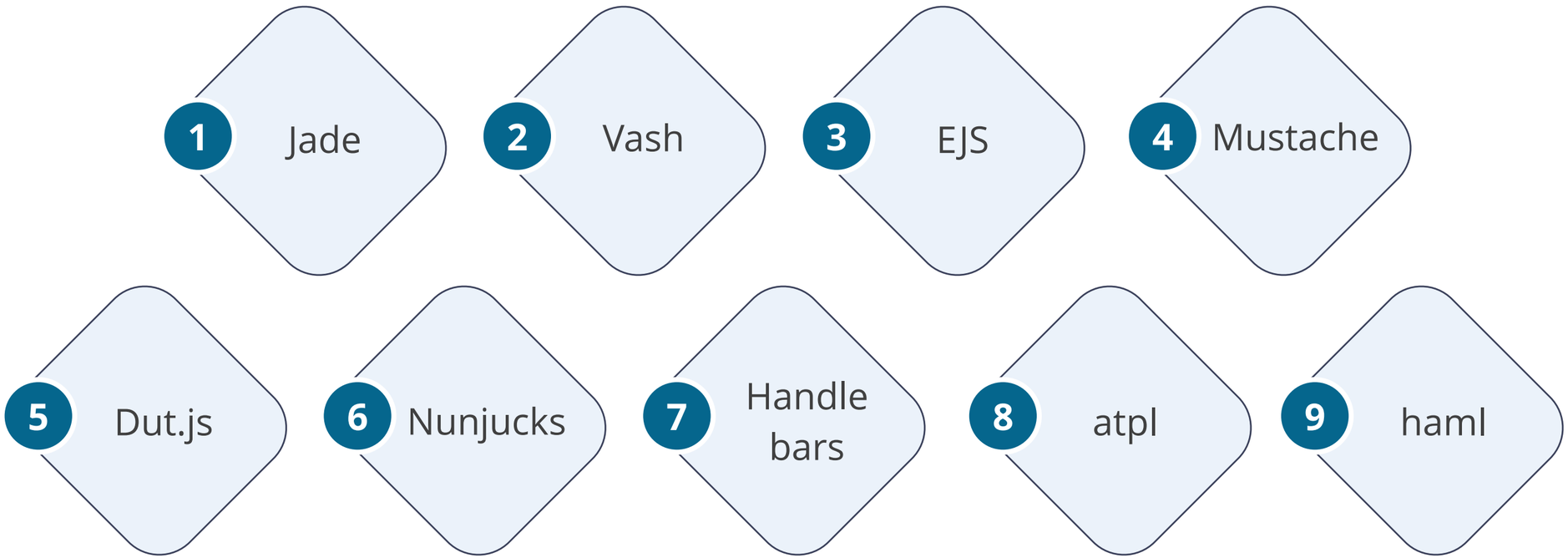
Template engines are used to divide the presentation layer from the application layer to maintain the code and edit the page.



Templates allow quick rendering of the server-side data that must be transmitted to the application.

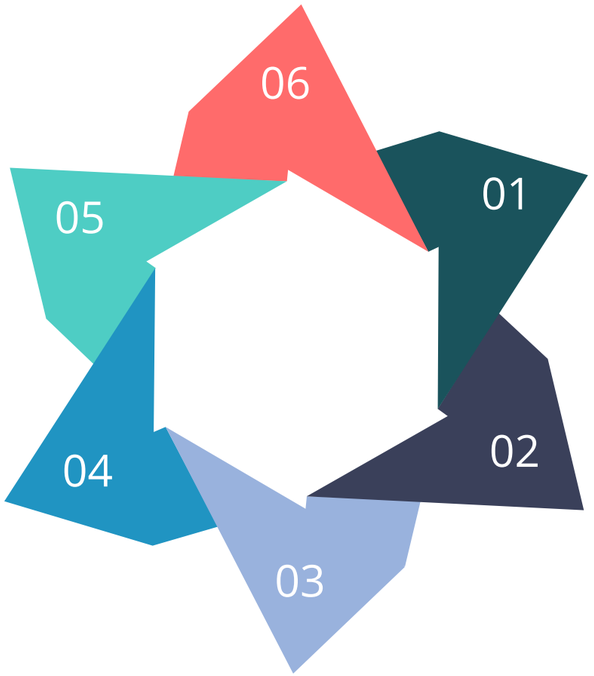
Template Engines in Node.js

Commonly used template engines for Node.js:



Advantages of Template Engines

Helps to access templates from CDN



Increases the code effectiveness

Maximizes client-side processing

Increases maintainability and readability

Uses single template for multiple pages

Increases performance

Jade

Jade is a template engine for Node.js.

It is easy to learn and uses whitespace and indentation within its syntax.

Jade: Features

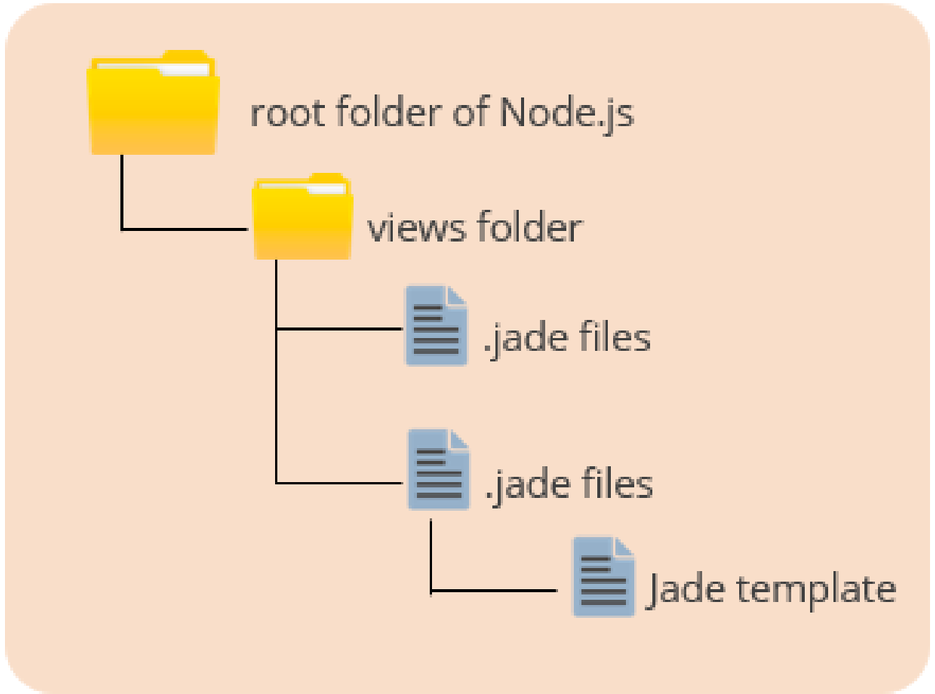
Jade provides a powerful new way to:

- Use simple tags
- Add attributes to the tags
- Use blocks of text

Jade: Installation

Syntax to install jade:

```
npm install jade
```



Vash

Vash offers a swift flow between code and content using Razor syntax.

Syntax to install Vash:

```
npm install vash
```

- It includes characteristics such as partials, layout inheritance, and model binding.
- It supports several different template languages, including HTML, JSX, and CSS.

Vash: Features

- It mixes code and content without delineators like <?, <%, or {{.
- It is just HTML-aware JavaScript.
- It works with markup and can be used with any other language.
- It allows API for extensibility and meta programming.
- It works in the browser or in Node.js
- It is inspired by the Jade layout engine.

Using Vash: Example

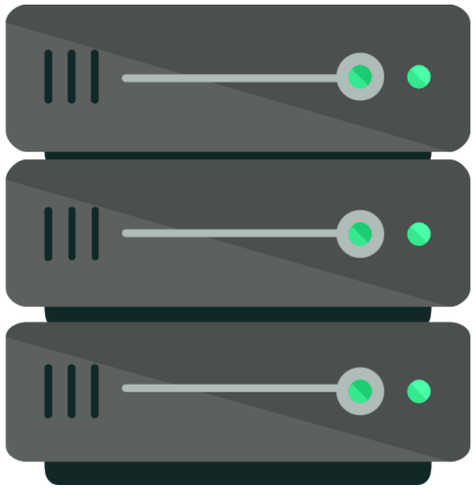
Example of HTML file to use Vash as a template engine:

```
<html>
    <body>
    <p>Welcome @model.name?</p>
<p>Your Account Status is @model.accountStatus?</p>
<p>Your Transactions</p>
<ul class="@(model.active ? 'highlight' : ' ')">
    @model.forEach(function(m){
    <li>@m.amount</li>
})
</ul>
</body>
</html>
```

Multiprocessing in Node.js

Multiprocessing

Multiprocessing is a server-side scaling technique that helps to execute multiple instances.



A crash in one thread will cause the other to handle incoming requests.

Multiprocessing on Node.js

Node.js is a JavaScript runtime that provides a variety of modules for developers.

It allows single-threaded non-blocking performance.

It uses a child process module to spawn child processes.

Child Processes

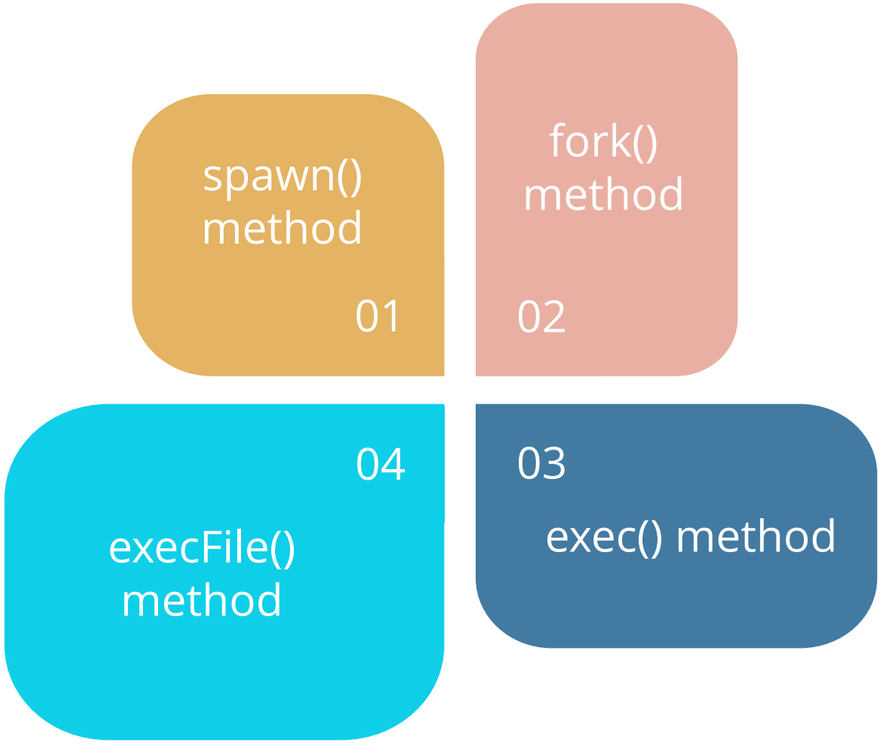
Child processes can execute commands, spawn new processes, and perform other tasks.

Uses of child processes:

- From main process, it is used to offload non-essential tasks and free up resources for further use.
- Between multiple process, it is used to facilitate inter-process communication.

Ways to Create the Child Process

There are four ways to create a child process in Node.js, namely:



spawn() Method

spawn() method creates a new process through command rather than running on the current node process.

Syntax:

```
child_process.spawn(command [, args ], [, options ])
```

It is used when the user wants the child process to return data to the parent process.

In this method, a new V8 instance is not created.

fork() Method

fork() method makes the child processes run on the same node process as the parent.

It separates the intensive computation task from the main event loop.

Syntax:

```
child_process.fork(modulePath[,
args] [,options])
```

In this method, a new V8 instance is created.

exec() Method

exec() is used to execute external programs or scripts.

Syntax:

```
child_process.exec(command [, options] [, callback])
```

It creates a shell and then executes the command.

execFile() Method

execFile() can detect when a process completes, allowing resources to be free for other tasks.

Syntax:

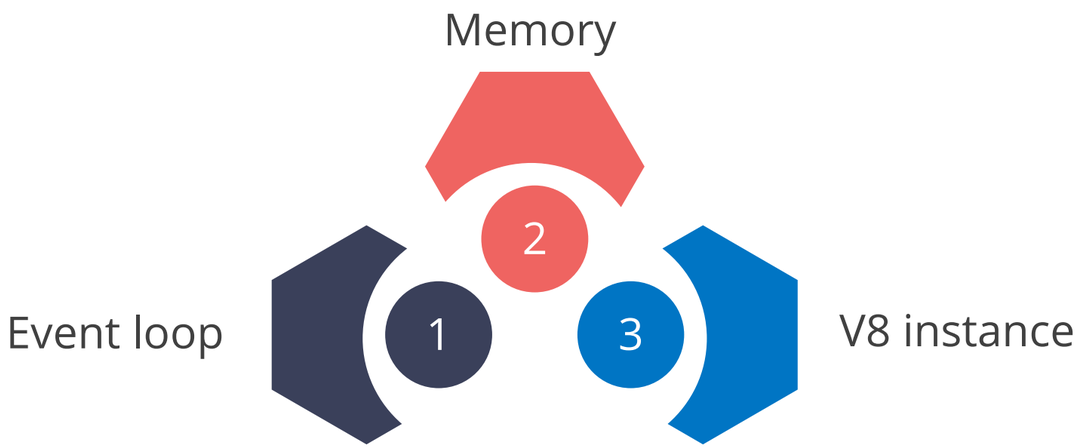
```
child_process.execFile(file[, args ] [, options ] [, callback])
```

- By default, **child_process.execFile()** function does not spawn a shell.
- The execFile() method is more efficient than the exec() method.

Node.js Cluster

Node.js cluster is a module that enables the creation of child processes.

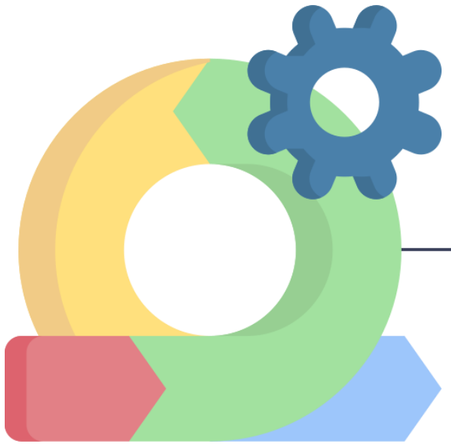
Each spawned child has its own:



The child processes communicate with the parent process using inter-process communication (IPC).

Methods in Node.js Cluster

Node.js cluster can distribute incoming connections using two methods:

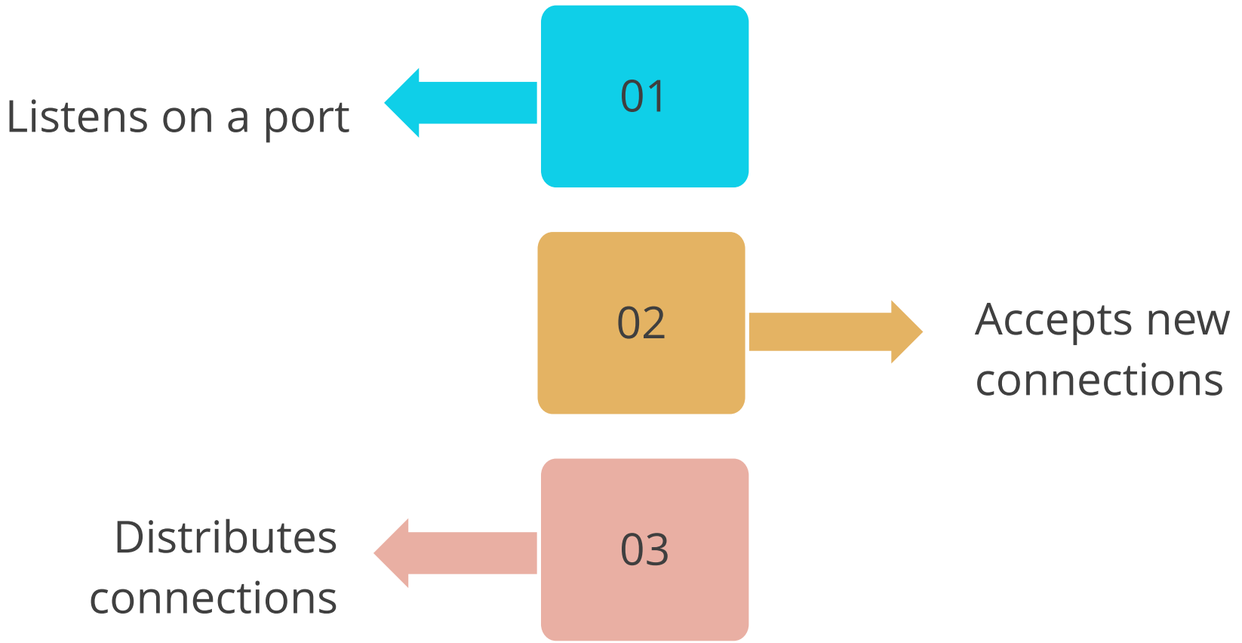


- **Round-Robin**
- **Socket**

Round-Robin Method

Round-robin in Node.js evenly distributes tasks among servers in a circular order.

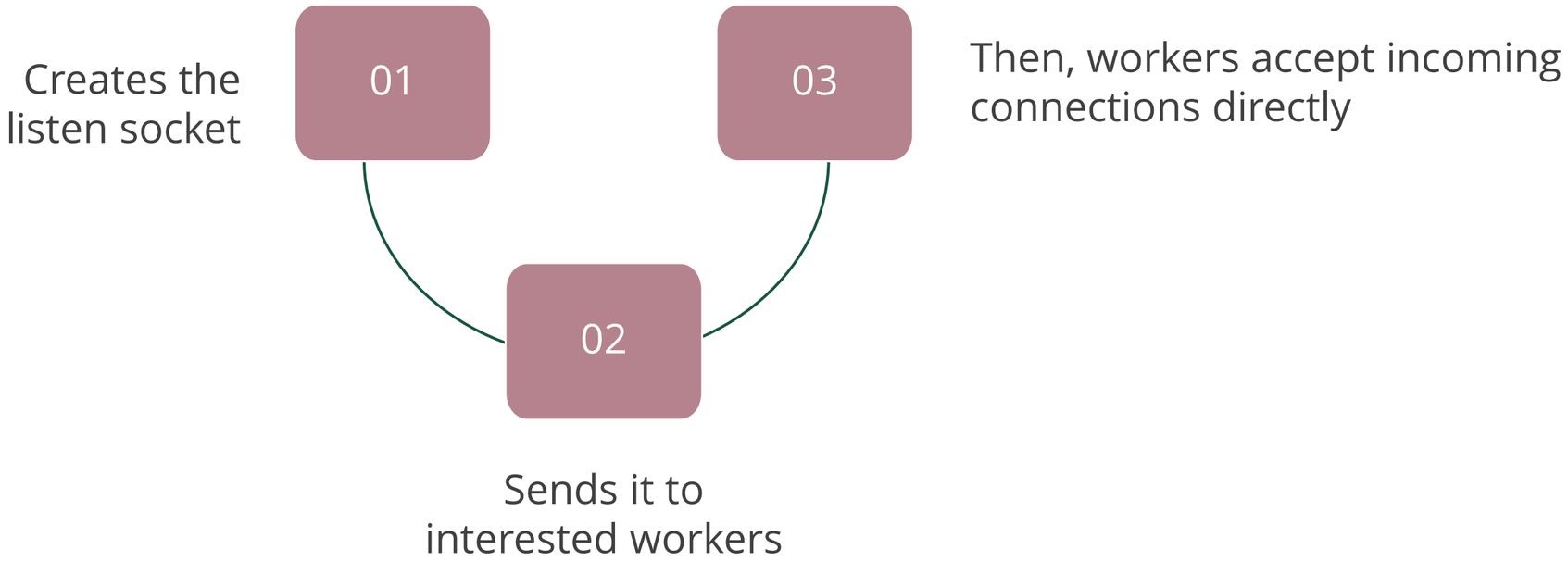
The primary process in round-robin approach:



Socket Method

The socket method is a full-duplex communication channel used to send and receive data between nodes in a Node.js cluster and between the client and server in a Node.js application.

The primary process in the socket method:



Worker Object in Cluster

The worker object lets developers specify the maximum number of workers to create and to be active at any given time.

It includes public information and methods about the workers.

It is fetched using the **cluster.workers** in the primary cluster.

It is obtained using the **cluster.worker** in worker cluster.

Worker Object in Cluster

Cluster API is used to create workers based on the number of available CPUs for multi-core processing.

Syntax:

```
import cluster from 'node:cluster';
import { availableParallelism } from 'node:os';
import process from 'node:process';

const availableCPUs = availableParallelism();

if (cluster.isPrimary) {
  console.log('Primary Cluster Process${process.pid} is running');
// Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log('worker ${worker.process.pid} died');
  });
}
```

Creating a Child Process

Problem Statement:

Duration: 20 min.

You have been assigned a task to create a child process in Node.js.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Understand Multiprocessing
2. Work with cluster API for multi-core servers

Key Takeaways

- fs.readFile() method is an inbuilt method used to read the file.
- Asynchronous I/O (AIO) operations run in the background and do not block user applications.
- Every file in the system has a path.
- The fs.writeFileSync() is a synchronous method to write data in the file.

Thank You