Interview Question for Web Back-End Development

Ali-Akbor

**Q1: What do you mean by Asynchronous API?**

**Ans:**

All APIs of Node.js library are aynchronous that is non-blocking. It essentially means a Node.js based server never waits for a API to return data. Server moves to next API after calling it and a notification mechanism of Events of Node.js helps server to get response from the previous API call.

***Q2*: What is *Sharding* in MongoDB?**

**Ans:**

**Sharding** is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations. MongoDB supports horizontal scaling through sharding. MongoDB shards data at the collection level, distributing the collection data across the shards in the cluster.

## *Q3*: How does React work?

**Ans:**

## React creates a virtual DOM. When state changes in a component it firstly runs a "diffing" algorithm, which identifies what has changed in the virtual DOM. The second step is reconciliation, where it updates the DOM with the results of diff.

## *Q4*: What are React Hooks?

**Ans:**

## ****Hooks**** are a new addition in React 16.8. They let you use state and other React features without writing a class. With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components or with the community.

## *Q5*: What's the difference between useRefand createRef?

**Ans:**

The difference is:

* createRef will always create a new ref. In a class-based component, you would typically put the ref in an instance property during construction (e.g. this.input = createRef()). You don't have this option in a function component.
* useRef takes care of returning the same ref each time as on the initial rendering.

## *Q6*: What are props in React?

**Ans:**

**Props** are inputs to a React component. They are single values or objects containing a set of values that are passed to React Components on creation using a naming convention similar to HTML-tag attributes. i.e, *They are data passed down from a parent component to a child component.*

The primary purpose of props in React is to provide following component functionality:

1. Pass custom data to your React component.
2. Trigger state changes.
3. Use via this.props.reactProp inside component's render() method.

For example, let us create an element with reactProp property,

<Element reactProp = "1" />

This reactProp (or whatever you came up with) name then becomes a property attached to React's native props object which originally already exists on all components created using React library.

props.reactProp;

## *Q7*: What are the advantages of ReactJS?

**Ans:**

Below are the advantages of ReactJS:

1. Increases the application’s performance with Virtual DOM
2. JSX makes code is easy to read and write
3. It renders both on client and server side
4. Easy to integrate with other frameworks (Angular, BackboneJS) since it is only a view library
5. Easy to write UI Test cases and integration with tools such as JEST.

## *Q8*: How is React different from AngularJS (1.x)?

**Ans:**

For example, AngularJS (1.x) approaches building an application by extending HTML markup and injecting various constructs (e.g. Directives, Controllers, Services) at runtime. As a result, AngularJS is very opinionated about the greater architecture of your application — these abstractions are certainly useful in some cases, but they come at the cost of flexibility.

By contrast, React focuses exclusively on the creation of components, and has few (if any) opinions about an application’s architecture. This allows a developer an incredible amount of flexibility in choosing the architecture they deem “best” — though it also places the responsibility of choosing (or building) those parts on the developer.

## *Q9*: What Is Replication In MongoDB?

**Ans:**

**Replication** is the process of synchronizing data across multiple servers.

Replication provides redundancy and increases [data availability](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-high-availability). With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centres can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

## *Q10*: What are Higher-Order Components (HOC) in React?

**Ans:**

A higher-order component **(HOC)** is a function that takes a component and returns a new component. Basically, it’s a pattern that is derived from React’s compositional nature We call them as **“pure’ components”** because they can accept any dynamically provided child component but they won’t modify or copy any behavior from their input components.

const EnhancedComponent =

higherOrderComponent(WrappedComponent);

HOC can be used for many use cases as below,

1. Code reuse, logic and bootstrap abstraction
2. Render High jacking
3. State abstraction and manipulation
4. Props manipulation

## *Q11*: What are advantages of using React Hooks?

**Ans:**

Primarily, hooks in general enable the extraction and reuse of stateful logic that is common across multiple components without the burden of higher order components or render props. Hooks allow to easily manipulate the state of our functional component without needing to convert them into class components.

Hooks don’t work inside classes (because they let you use React without classes). By using them, we can totally avoid using lifecycle methods, such as componentDidMount, componentDidUpdate, componentWillUnmount. Instead, we will use built-in hooks like useEffect .

## *Q12*: What are the limitations of React?

**Ans:**

Below are the list of limitations:

1. React is just a view library, not a full-blown framework
2. There is a learning curve for beginners who are new to web development.
3. Integrating React.js into a traditional MVC framework requires some additional configuration
4. The code complexity increases with inline templating and JSX.
5. Too many smaller components leading to over-engineering or boilerplate

## *Q13*: What are the differences between a Class component and Functional component?

**Ans:**

**Class Components**

* Class-based Components uses ES6 class syntax. It can make use of the lifecycle methods.
* Class components extend from React.Component.
* In here you have to use this keyword to access the props and functions that you declare inside the class components.

**Functional Components**

* Functional Components are simpler comparing to class-based functions.
* Functional Components mainly focuses on the UI of the application, not on the behavior.
* To be more precise these are basically render function in the class component.
* Functional Components can have state and mimic lifecycle events using Reach Hooks

## *Q14*: What are the key features of Node.js?

**Ans:**

Let’s look at some of the key features of Node.js.

* **Asynchronous event driven IO helps concurrent request handling –** All APIs of Node.js are asynchronous. This feature means that if a Node receives a request for some Input/Output operation, it will execute that operation in the background and continue with the processing of other requests. Thus it will not wait for the response from the previous requests.
* **Fast in Code execution –** Node.js uses the V8 JavaScript Runtime engine, the one which is used by Google Chrome. Node has a wrapper over the JavaScript engine which makes the runtime engine much faster and hence processing of requests within Node.js also become faster.
* **Single Threaded but Highly Scalable –** Node.js uses a single thread model for event looping. The response from these events may or may not reach the server immediately. However, this does not block other operations. Thus making Node.js highly scalable. Traditional servers create limited threads to handle requests while Node.js creates a single thread that provides service to much larger numbers of such requests.
* **Node.js library uses JavaScript –** This is another important aspect of Node.js from the developer’s point of view. The majority of developers are already well-versed in JavaScript. Hence, development in Node.js becomes easier for a developer who knows JavaScript.
* **There is an Active and vibrant community for the Node.js framework –** The active community always keeps the framework updated with the latest trends in the web development.
* **No Buffering –** Node.js applications never buffer any data. They simply output the data in chunks.

## *Q15*: What is Callback Hell and what is the main cause of it?

**Ans:**

Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

fs.readdir(source, function (err, files) {

if (err) {

console.log('Error finding files: ' + err)

} else {

files.forEach(function (filename, fileIndex) {

console.log(filename)

gm(source + filename).size(function (err, values) {

if (err) {

console.log('Error identifying file size: ' + err)

} else {

console.log(filename + ' : ' + values)

aspect = (values.width / values.height)

widths.forEach(function (width, widthIndex) {

height = Math.round(width / aspect)

console.log('resizing ' + filename + 'to ' + height + 'x' + height)

this.resize(width, height).write(dest + 'w' + width + '\_' + filename, function(err) {

if (err) console.log('Error writing file: ' + err)

})

}.bind(this))

}

})

})

}

})

See the pyramid shape and all the }) at the end? This is affectionately known as **callback hell**.

The **cause of callback hell** is when people try to write JavaScript in a way where execution happens visually from top to bottom. Lots of people make this mistake! In other languages like C, Ruby or Python there is the expectation that whatever happens on line 1 will finish before the code on line 2 starts running and so on down the file.

## *Q16*: What is Reconciliation in ReactJS?

**Ans:**

## When a component’s props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called ****reconciliation****.

## *Q17*: What is Sharding in MongoDB?

**Ans:**

## ****Sharding**** is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations. MongoDB supports horizontal scaling through [sharding](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-sharding). MongoDB shards data at the collection level, distributing the collection data across the shards in the cluster.

## *Q18*: What is the difference between returning a callback and just calling a callback?

**Ans:**

return callback();

//some more lines of code; - won't be executed

callback();

//some more lines of code; - will be executed

Of course returning will help the context calling async function get the value returned by callback.

function do2(callback) {

log.trace('Execute function: do2');

return callback('do2 callback param');

}

var do2Result = do2((param) => {

log.trace(`print ${param}`);

return `return from callback(${param})`; // we could use that return

});

log.trace(`print ${do2Result}`);

Output:

C:\Work\Node>node --use-strict main.js

[0] Execute function: do2

[0] print do2 callback param

[0] print return from callback(do2 callback param)

## *Q19*: When should we embed one document within another in MongoDB?

**Ans:**

You should consider embedded documents (subdocuments) for:

* When the relationship is one-to-few (not many, not unlimited). For unlimited use case, you should start considering separating subdocuments into another collection.
* When retrieval is likely to happen together, that will improve performance
* When updates are likely to happen at the same time. Although starting from MongoDB 4.0, you can use multi-documents transactions, a single document transaction would be more performant
* When the field is rarely updated

## *Q20*: Does Mongodb support Foreign Key constraints?

**Ans:**

No.

One of the great things about relational database is that it is really good at keeping the data consistent within the database. One of the ways it does that is by using foreign keys. A foreign key constraint is that let's say there's a table with some column which will have a foreign key column with values from another table's column.

In MongoDB, there's no guarantee that foreign keys will be preserved. It's upto the programmer to make sure that the data is consistent in that manner. Constraints can not be enforced by MongoDB either. It can't even enforce a specific type for a field, due to the schemaless nature of MongoDB.

## *Q21*: Explain advantages of BSON over JSON in MongoDB?

**Ans:**

* **BSON** is designed to be efficient in space, but in some cases is not much more efficient than JSON. In some cases BSON uses even more space than JSON. The reason for this is another of the BSON design goals: traversability. BSON adds some "extra" information to documents, like length of strings and subobjects. This makes traversal faster.
* BSON is also designed to be fast to encode and decode. For example, integers are stored as 32 (or 64) bit integers, so they don't need to be parsed to and from text. This uses more space than JSON for small integers, but is much faster to parse.
* In addition to compactness, BSON adds additional data types unavailable in JSON, notably the BinData and Date data types.

## *Q22*: Given the React code defined above, can you identify two problems?

***Problem***

Take a look at the code below:

class MyComponent extends React.Component {

constructor(props) {

// set the default internal state

this.state = {

clicks: 0

};

}

componentDidMount() {

this.refs.myComponentDiv.addEventListener('click', this.clickHandler);

}

componentWillUnmount() {

this.refs.myComponentDiv.removeEventListener('click', this.clickHandler);

}

clickHandler() {

this.setState({

clicks: this.clicks + 1

});

}

render() {

let children = this.props.children;

return (

<div className="my-component" ref="myComponentDiv">

<h2>My Component ({this.state.clicks} clicks})</h2>

<h3>{this.props.headerText}</h3>

{children}

</div>

);

}

}

Given the code defined above, can you identify two problems?

**Ans:**

1. The constructor does not pass its props to the super class. It should include the following line:

constructor(props) {

super(props);

// ...

}

1. The event listener (when assigned via addEventListener()) is not properly scoped because [ES2015 doesn’t provide autobinding](https://facebook.github.io/react/docs/reusable-components.html#no-autobinding). Therefore the developer can re-assign clickHandler in the constructor to include the correct binding to this:

constructor(props) {

super(props);

this.clickHandler = this.clickHandler.bind(this);

// ...

}

## *Q23*: How can you achieve Transaction in MongoDB?

**Ans:**

In MongoDB, an operation on a single document is **atomic**.

Because you can use *embedded documents* and *arrays* to capture *relationships* between data in a single document structure instead of normalizing across multiple documents and collections, this single-document atomicity obviates the need for multi-document transactions for many practical use cases.

For situations that require atomicity of reads and writes to multiple documents (in single or multiple collections), MongoDB supports **multi-document (distributed) transactions**.

## *Q24*: How does Node.js handle Child Threads?

**Ans:**

Node.js, in its essence, is a single **thread** process. It does not expose **child threads** and **thread** management methods to the developer. **js** does spawn **child threads** for certain tasks such as asynchronous I/O, but these run behind the scenes and do not execute **any** application **JavaScript** code, nor block the main event loop.

If threading support is desired in a Node.js application, there are tools available to enable it, such as the ChildProcess module.

## *Q25*: How does concurrency work in Node.js?

**Ans:**

The thing with node.js is that everything runs concurrently, except for your code.

So, what that means is that there are actually lots of threads running inside Node.js virtual machine (or a thread pool if you wish), and those threads are utilized whenever you call an async function like performing i/o operations on files, accessing databases, requesting urls, etc.

However, for your code, there is only a single thread, and it processes events from an [event queue](http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/). So, when you register a callback its reference is actually passed to the background worker thread, and once the async operation is done, new event is added to the event-queue with that callback

When Node gets I/O request it creates or uses a thread to perform that I/O operation and once the operation is done, it pushes the result to the **event queue**. On each such event, **event loop** runs and checks the queue and if the execution stack of Node is empty then it adds the queue result to execution stack.

This is how Node manages concurrency.

## *Q26*: How to avoid Callback Hell in Node.js?

**Ans:**

Node.js internally uses a single-threaded event loop to process queued events. But this approach may lead to blocking the entire process if there is a task running longer than expected. Node.js addresses this problem by incorporating callbacks also known as higher-order functions. So whenever a long-running process finishes its execution, it triggers the callback associated. Sometimes, it could lead to complex and unreadable code. More the no. of callbacks, longer the chain of returning callbacks would be.

There are four solutions which can address the callback hell problem:

* **Make your program modular** - It proposes to split the logic into smaller modules. And then join them together from the main module to achieve the desired result.
* **Use async/await mechanism** - Async /await is another alternative for consuming promises, and it was implemented in ES8, or ES2017. Async/await is a new way of writing promises that are based on asynchronous code but make asynchronous code look and behave more like synchronous code.
* **Use promises mechanism** - Promises give an alternate way to write async code. They either return the result of execution or the error/exception. Implementing promises requires the use of .then() function which waits for the promise object to return. It takes two optional arguments, both functions. Depending on the state of the promise only one of them will get called. The first function call proceeds if the promise gets fulfilled. However, if the promise gets rejected, then the second function will get called.
* **Use generators** - Generators are lightweight routines, they make a function wait and resume via the yield keyword. Generator functions uses a special syntax function\* (). They can also suspend and resume asynchronous operations using constructs such as promises or thunks and turn a synchronous code into asynchronous.
* function\* HelloGen() {
* yield 100;
* yield 400;
* }
* var gen = HelloGen();
* console.log(gen.next()); // {value: 100, done: false}
* console.log(gen.next()); // {value: 400, done: false}

. console.log(gen.next()); // {value: undefined, done: true}

## *Q27*: How to query MongoDB with like?

I want to query something as SQL's like query:

select \*

from users

where name like '%m%'

How to do the same in MongoDB?

**Ans:**

db.users.find({"name": /.\*m.\*/})

// or

db.users.find({"name": /m/})

You're looking for something that contains "m" somewhere (SQL's '%' operator is equivalent to Regexp's '.\*'), not something that has "m" anchored to the beginning of the string.

## *Q28*: Rewrite promise-based Node.js applications to async/await

**Problem**

Rewrite this code to async/await:

function asyncTask() {

return functionA()

.then((valueA) => functionB(valueA))

.then((valueB) => functionC(valueB))

.then((valueC) => functionD(valueC))

.catch((err) => logger.error(err))

}

**Ans:**

async function asyncTask() {

try {

const valueA = await functionA()

const valueB = await functionB(valueA)

const valueC = await functionC(valueB)

return await functionD(valueC)

} catch (err) {

logger.error(err)

}

}

## *Q29*: What are Pure Components?

**Ans:**

**PureComponent** is exactly the same as **Component** except that it handles the shouldComponentUpdate method for you.

When props or state changes, PureComponent will do a shallow comparison on both props and state. Component, on the other hand, won’t compare current props and state to next out of the box. Thus, the component will re-render by default whenever shouldComponentUpdate is called.

## *Q30*: What is prop drilling and how can you avoid it?

**Ans:**

When building a React application, there is often the need for a deeply nested component to use data provided by another component that is much higher in the hierarchy. The simplest approach is to simply pass a prop from each component to the next in the hierarchy from the source component to the deeply nested component. This is called **prop drilling**.

The primary disadvantage of prop drilling is that components that should not otherwise be aware of the data become unnecessarily complicated and are harder to maintain.

To avoid prop drilling, a common approach is to use React context. This allows a Provider component that supplies data to be defined, and allows nested components to consume context data via either a Consumer component or a useContext hook.

## *Q31*: What is Key and benefit of using it in lists?

**Ans:**

A **key** is a special string attribute you need to include when creating lists of elements. Keys help React identify which items have changed, are added, or are removed.

For example, most often we use IDs from your data as keys

const todoItems = todos.map((todo) =>

<li key={todo.id}>

{todo.text}

</li>

);

When you don’t have stable IDs for rendered items, you may use the item index as a key as a last resort:

const todoItems = todos.map((todo, index) =>

<li key={index}>

{todo.text}

</li>

);

**Note:**

1. We don’t recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state
2. If you extract list item as separate component then apply keys on list component instead li tag.

There will be a warning in the console if the key is not present on list items.

## *Q32*: What is stream and what are types of streams available in Node.js?

**Ans:**

A stream is an abstract interface for working with streaming data in Node.js.

Streams basically provide two major advantages over using other data handling methods:

* **Memory efficiency**: you don't need to load large amounts of data in memory before you are able to process it
* **Time efficiency**: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

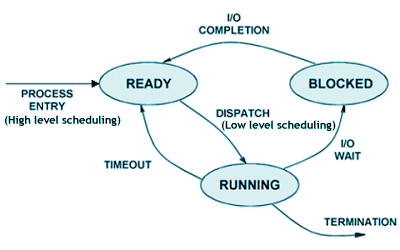
There are 4 types of streams in Node.js:

1. **Writable:** streams to which we can write data. For example, fs.createWriteStream() lets us write data to a file using streams.
2. **Readable:** streams from which data can be read. For example: fs.createReadStream() lets us read the contents of a file.
3. **Duplex:** streams that are both Readable and Writable. For example, net.Socket
4. **Transform:** streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read decompressed data to and from a file.

## *Q33*: What is a Blocking Code in Node.js?

**Ans:**

A **blocking** call causes results to be returned synchronously.



Performing a blocking system call causes the process to enter the *blocked* state. Control is let back to the process only after the I/O event that is being waited upon occurs.

const fs = require("fs");

const contents = fs.readFileSync("file.txt", "utf8");

// this line is not reached until the read results are in

console.log(contents);

## *Q34*: What is an Aggregation Pipeline in MongoDB?

**Ans:**

Aggregation operations process multiple documents and return computed results. You can use aggregation operations to:

* **Group** values from multiple documents together.
* **Perform operations** on the grouped data to return a single result.
* **Analyze** data changes over time.

MongoDB provides aggregation operations through aggregation pipelines — a series of operations that process data documents sequentially. An aggregation pipeline consists of one or more [stages](https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/#std-label-aggregation-pipeline-operator-reference) that process documents:

* Each stage performs an operation on the input documents. For example, a stage can **filter** documents, **group** documents, and **calculate** values.
* The documents that are output from a stage are passed to the next stage.
* An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

Consider:

db.orders.aggregate([

// Stage 1: Filter pizza order documents by pizza size

{

$match: { size: "medium" }

},

// Stage 2: Group remaining documents by pizza name and calculate total quantity

{

$group: { \_id: "$name", totalQuantity: { $sum: "$quantity" } }

}

])

## *Q35*: What is the difference between ShadowDOM and VirtualDOM?

**Ans:**

***Virtual DOM***

Virtual DOM is about avoiding unnecessary changes to the DOM, which are expensive performance-wise, because changes to the DOM usually cause re-rendering of the page. Virtual DOM also allows to collect several changes to be applied at once, so not every single change causes a re-render, but instead re-rendering only happens once after a set of changes was applied to the DOM.

***Shadow DOM***

Shadow dom is mostly about encapsulation of the implementation. A single custom element can implement more-or-less complex logic combined with more-or-less complex DOM. An entire web application of arbitrary complexity can be added to a page by an import and <body><my-app></my-app> but also simpler reusable and composable components can be implemented as custom elements where the internal representation is hidden in the shadow DOM like <date-picker></date-picker>.

## *Q36*: What's the Event Loop?

**Ans:**

**The event loop** is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

Every I/O requires a callback - once they are done they are pushed onto the event loop for execution. Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

**Q37: Why should you separate Express app and server?**

**Ans:**

Applying a similar concept to the project structuring of Express, the separation of the application logic from the server allows the code to be modular and follow an MVC (Model-View-Controller) model. The separation is essential to reduce coupling and to encapsulate and abstract the inside logic of application.

## Q38: Write equivalent MongoDB statement for this SQL aggregation statement.

Problem:

Consider the SQL query:

SELECT by, SUM(price) AS total FROM book GROUP BY price ORDER BY total

Write down the MongoDB equivalent.

**Ans:**

Consider:

db.book.aggregate([

{

$group: {

\_id: "$by",

total: { $sum:"$price" }

}

},

{

$sort: { total:1 }

}

])

## *Q39*: What types of One-to-N schema designs you can use in MongoDB?

**Ans:**

There are three basic One-to-N schema designs:

* *Embed* the N side if the cardinality is **one-to-few** and there is no need to access the embedded object outside the context of the parent object. For example, a list of person's addresses.
* Use an *array of references* to the N-side objects if the cardinality is **one-to-many** or if the N-side objects should stand alone for any reason. For example, parts for a product.
* Use a *reference* to the One-side in the N-side objects (parent-referencing) if the cardinality is **one-to-squillions**. For example, logs from N machines.

## *Q40*: What is the difference between One-to-Many vs One-to-Few in MongoDB Schema Design?

**Ans:**

An example of one-to-few might be the addresses for a person. This is a good use case for embedding – you’d put the addresses in an array inside of your Person object:

> db.person.findOne()

{

name: 'Kate Monster',

ssn: '123-456-7890',

addresses : [

{ street: '123 Sesame St', city: 'Anytown', cc: 'USA' },

{ street: '123 Avenue Q', city: 'New York', cc: 'USA' }

]

}

An example of one-to-many might be parts for a product in a replacement parts ordering system. his is a good use case for referencing – you’d put the ObjectIDs of the parts in an array in product document.

Each Product would have its own document, which would contain an array of ObjectID references to the Parts that make up that Product:

> db.products.findOne()

{

name : 'left-handed smoke shifter',

manufacturer : 'Acme Corp',

catalog\_number: 1234,

parts : [ // array of references to Part documents

ObjectID('AAAA'), // reference to the #4 grommet above

ObjectID('F17C'), // reference to a different Part

ObjectID('D2AA'),

// etc

]

## *Q41*: What is a Covered Query in MongoDB?

**Ans:**

A **covered query** is a query that can be satisfied entirely using an index and does not have to examine any documents. An index [covers](https://www.mongodb.com/docs/manual/core/query-optimization/#std-label-indexes-covered-queries) a query when all of the following apply:

* all the fields in the [query](https://www.mongodb.com/docs/manual/tutorial/query-documents/#std-label-read-operations-query-document) are part of an index, **and**
* all the fields returned in the results are in the same index.
* no fields in the query are equal to null (i.e. {"field" : null} or {"field" : {$eq : null}} ).

For example, a collection inventory has the following index on the type and item fields:

db.inventory.createIndex( { type: 1, item: 1 } )

This index will cover the following operation which queries on the type and item fields and returns only the item field:

db.inventory.find( { type: "food", item:/^c/ }, { item: 1, id: 0 })

## *Q42*: What is BSON in MongoDB?

**Ans:**

**BSON** is a binary serialization format used to store documents and make remote procedure calls in MongoDB. BSON extends the JSON model to provide additional data types, and ordered fields, and to be efficient for encoding and decoding within different languages.

BSON encodes type and length information, too, making it easier for machines to parse. Consider:

{"hello": "world"} →

\x16\x00\x00\x00 // total document size

\x02 // 0x02 = type String

hello\x00 // field name

\x06\x00\x00\x00world\x00 // field value

\x00 // 0x00 = type EOO ('end of object')

## *Q43*: Can you create an index on an array field in MongoDB? If yes, what happens in this case?

**Ans:**

Yes. An array field can be indexed in MongoDB. In this case, MongoDB would index each value of the array so you can query for individual items:

> db.col1.save({'colors': ['red','blue']})

> db.col1.ensureIndex({'colors':1})

> db.col1.find({'colors': 'red'})

{ "\_id" : ObjectId("4ccc78f97cf9bdc2a2e54ee9"), "colors" : [ "red", "blue" ] }

> db.col1.find({'colors': 'blue'})

{ "\_id" : ObjectId("4ccc78f97cf9bdc2a2e54ee9"), "colors" : [ "red", "blue" ] }

## *Q44*: Explain the difference between local and global npm packages installation

**Ans:**

The main difference between local and global packages is this:

* **local packages** are installed in the directory where you run npm install <package-name>, and they are put in the node\_modules folder under this directory
* **global packages** are all put in a single place in your system (exactly where depends on your setup), regardless of where you run npm install -g <package-name>

In general, **all packages should be installed locally**.

* This makes sure you can have dozens of applications in your computer, all running a different version of each package if needed.
* Updating a global package would make all your projects use the new release, and as you can imagine this might cause nightmares in terms of maintenance, as some packages might break compatibility with further dependencies, and so on.

## *Q45*: What are the benefits of using Node.js?

**Ans:**

Following are main benefits of using Node.js

* **Aynchronous and Event Driven** - All APIs of Node.js library are aynchronous that is non-blocking. It essentially means a Node.js based server never waits for a API to return data. Server moves to next API after calling it and a notification mechanism of Events of Node.js helps server to get response from the previous API call.
* **Very Fast** - Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
* **Single Threaded but highly Scalable** - Node.js uses a single threaded model with event looping. Event mechanism helps server to respond in a non-bloking ways and makes server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and same program can services much larger number of requests than traditional server like Apache HTTP Server.
* **No Buffering** - Node.js applications never buffer any data. These applications simply output the data in chunks.

## *Q46*: What is Callback?

**Ans:**

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. 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."

var myCallback = function(data) {

console.log('got data: '+data);

};

var usingItNow = function(callback) {

callback('get it?');

};

## *Q47*: What is V8?

**Ans:**

The V8 library provides Node.js with a JavaScript engine (a program that converts Javascript code into lower level or machine code that microprocessors can understand), which Node.js controls via the V8 C++ API. V8 is maintained by Google, for use in Chrome.

The Chrome V8 engine :

* The V8 engine is written in C++ and used in Chrome and Nodejs.
* It implements ECMAScript as specified in ECMA-262.
* The V8 engine can run standalone we can embed it with our own C++ program.

## *Q48*: Are you familiar with differences between Node.js modules and ES6 modules?

**Ans:**

The modules used in Node.js follow a module specification known as the **CommonJS** specification. The recent updates to the JavaScript programming language, in the form of ES6, specify changes to the language, adding things like new class syntax and a module system. This module system is different from Node.js modules. To import ES6 module, we'd use the ES6 import functionality.

Now ES6 modules are incompatible with Node.js modules. This has to do with the way modules are loaded differently between the two formats. If you use a compiler like Babel, you can mix and match module formats.

## *Q49*: How would you read files in sequence in Node.js? Provide a code example

**Ans:**

async function printFiles () {

const files = await getFilePaths();

for (const file of files) {

const contents = await fs.readFile(file, 'utf8');

console.log(contents);

}

}

## *Q50*:  What is libuv?

**Ans:**

## **libuv** is a C library that is used to abstract non-blocking I/O operations to a consistent interface across all supported platforms. It provides mechanisms to handle file system, DNS, network, child processes, pipes, signal handling, polling and streaming. It also includes a thread pool for offloading work for some things that can't be done asynchronously at the operating system level.