Integration of Notification Service with the CRM application

Relevel
by Unacademy

Topics

- Introduction to Asynchronous communications
- Sync Vs Async communications
- Call the Notification service every time a new ticket is created, to send email to the reporter and assignee
- Call the Notification service every time a ticket is updated, to send email to all the stakeholders
- Challenges of hardcoding the URLs in the code
- Things could change if load on the system increases

Features to be Developed in this session

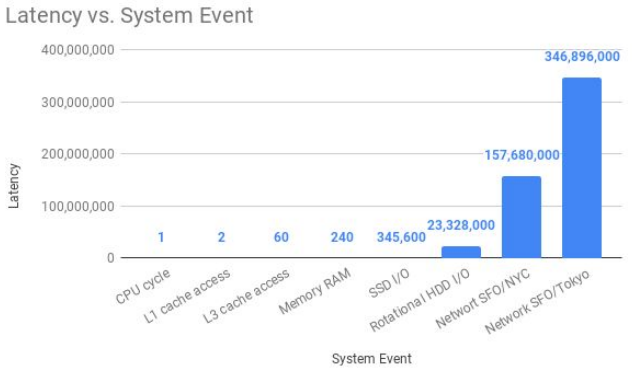
- Call the Notification service every time a new ticket is created, to send email to the reporter and assignee
- Call the Notification service every time a ticket is updated, to send email to all the stakeholders

Introduction to Synchronous and Asynchronous Communications

- The task of simultaneously running and managing multiple computations can be handled in a variety of ways in programming languages can be handled in a variety of ways in programming languages. Some languages make use of multiple threads, while others make use of the asynchronous model. We'll go over the latter in depth and provide examples to differentiate between synchronous and asynchronous.But what does the CPU do most of time? It is Idle
- Our computer's processor is waiting for a network request. It pauses for external events (I/O) and idles for the hard drive to spin out the requested data.

Introduction to Synchronous and Asynchronous Communications

Look at the graph below to see how long this system event takes on average (in nanoseconds)
Here Latency is nothing but measure of delay,
A system event is a point in a program's execution when an identifiable computing task occurs. Writing to a file, inserting data into a database table, calling a subroutine, and instantiating a programme thread are all examples of system events.



Introduction to Synchronous and Asynchronous Communications

As shown in the graph above, one CPU can execute an instruction every ns (approx.). However, if you are in New York and send a request to a website in San Francisco, the CPU will "waste" 157 million cycles waiting for it to return!
If we use non-blocking (asynchronous) code in our programmes, we can use that time to do other things!

Synchronous and Asynchronous in Node Js

We'll compare two approaches to reading files:
one using a blocking I/O model and the other
using a non-blocking I/O model.
You can find the code here
https://github.com/lavanya23/Sync

Synchronous code for reading from a file in Node.js

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

console.log('start');

const data = fs.readFileSync('./file.txt', 'utf-8'); // blocks here until file is read
console.log('data: ', data.trim());

console.log('end');
```

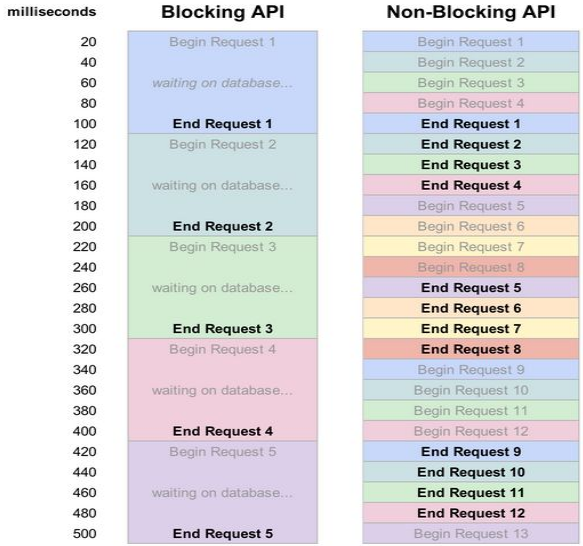
Synchronous = Blocking I/O model

Output

```
start
data:  Hello World! 👋 🌎
end
```

Asynchronous Non-Blocking Code

Most Node.js functions are non-blocking, To handle multiple requests efficiently, Node.js combines a JavaScript engine, an event loop, and an I/O layer.



milliseconds	Blocking API	Non-Blocking API
20	Begin Request 1	Begin Request 1
40		Begin Request 2
60	*waiting on database...*	Begin Request 3
80		Begin Request 4
100	**End Request 1**	**End Request 1**
120	Begin Request 2	**End Request 2**
140		**End Request 3**
160	waiting on database...	**End Request 4**
180		Begin Request 5
200	**End Request 2**	Begin Request 6
220	Begin Request 3	Begin Request 7
240		Begin Request 8
260	waiting on database...	**End Request 5**
280		**End Request 6**
300	**End Request 3**	**End Request 7**
320	Begin Request 4	**End Request 8**
340		Begin Request 9
360	waiting on database...	Begin Request 10
380		Begin Request 11
400	**End Request 4**	Begin Request 12
420	Begin Request 5	**End Request 9**
440		**End Request 10**
460	waiting on database...	**End Request 11**
480		**End Request 12**
500	**End Request 5**	Begin Request 13

Asynchronous Non-Blocking Code

As can be seen, asynchronous functions can handle more operations while waiting for IO resources to become available.
Let's look at an example of asynchronous code reading from a file.
https://github.com/lavanya23/Async

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

console.log('start');

fs.readFile('./file.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('file.txt data: ', data.trim());
});

console.log('end');
```

Output of the program

Because the programme does not halt and continues executing whatever is next, the end comes before the file output.

```
start
end
file.txt data:  Hello World! 👋 🌎
```

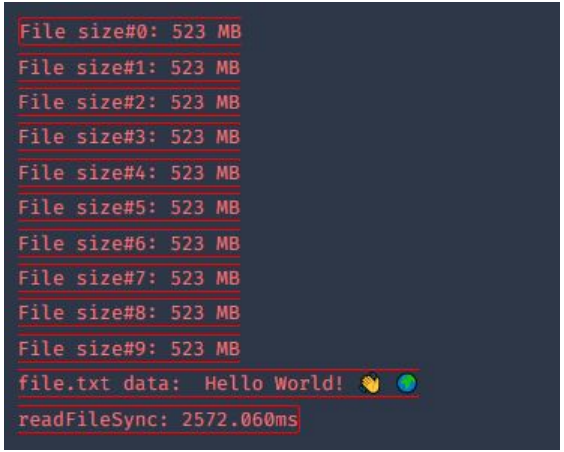
Blocking vs. Non-Blocking I/O model

https://github.com/lavanya23/blocking

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

console.time('readFileSync');

for (let x = 0; x < 10; x++) {
  const largeFile = fs.readFileSync('/users/admejiar/Downloads/Docker.dmg');
  console.log(`File size#${x}: ${Math.round(largeFile.length / 1e6)} MB`);
}

const data = fs.readFileSync('./file.txt', 'utf-8'); // blocks here until file is read
console.log('file.txt data: ', data.trim());

console.timeEnd('readFileSync');
```

Blocking vs. Non-Blocking I/O model

It's worth noting that we're using console.time, which is useful for benchmarking because it calculates how many milliseconds it took. The result is as follows:



All ten files and the file.txt took 2.5 seconds to read.

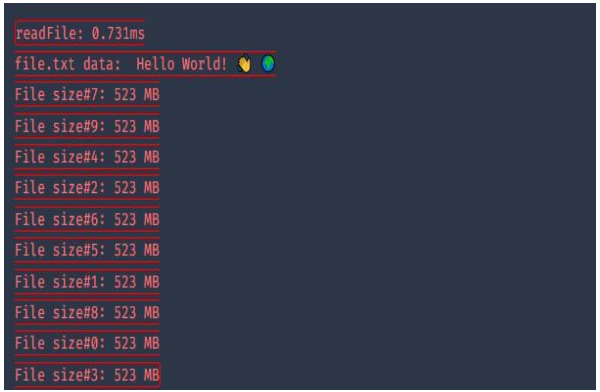
Blocking vs. Non-Blocking I/O model

Let's try it again with non-blocking: https://github.com/lavanya23/non-blocking

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

console.time('readFile');

for (let x = 0; x < 10; x++) {
  fs.readFile('/users/admejiar/Downloads/Docker.dmg', (err, data) => {
    if (err) throw err;
    console.log(`File size#${x}: ${Math.round(data.length / 1e6)} MB`);
  });
}

fs.readFile('./file.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('file.txt data: ', data.trim());
});

console.timeEnd('readFile');
```

Blocking vs. Non-Blocking I/O model

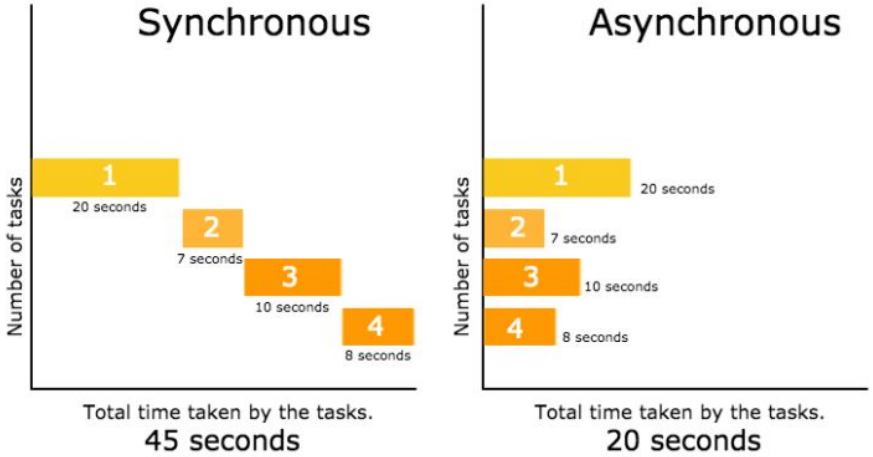
Output will be



It made it to the console. End of time in less than a millisecond! The small file.txt followed, followed by the large files, all in a different order. As can be seen, non-blocking waits for no one. Whoever is the most prepared will be the first to emerge. It has many advantages despite the fact that it is not deterministic.

Blocking vs. Non-Blocking I/O model

Async programmes will take the same amount of time as the most time-consuming task. It runs tasks in parallel, whereas the blocking model runs them sequentially.



Blocking vs. Non-Blocking I/O model

Advantages of Non-Blocking Code

Non-blocking code is far more efficient. Blocking code consumes approximately 90% of CPU cycles while waiting for data from the network or disc. Non-blocking code is a simpler way to achieve concurrency without having to deal with multiple execution threads.

Blocking vs. Non-Blocking I/O model

As previously stated, the blocking API server can only handle one request at a time. It serves request #1, then idles for the database before proceeding to serve the other requests. The non-blocking API, on the other hand, can handle multiple requests while waiting for the database to respond.

We can handle asynchronous code in JavaScript/Node JS by using:

- Callbacks
- Promises
- Async/Await functions
- Generators

Sync Vs Async

Sr.no	Synchronous methods	Asynchronous methods
1.	Synchronous functions are called blocking functions	Asynchronous functions are called non-blocking functions.
2.	It blocks the execution of the program until the file operation has finished processing.	It does not block the execution of the program.
3.	These functions take File Descriptor as the last argument.	These functions take a callback function as the last argument.
4.	Examples: fs.readFileSync(), fs.appendFileSync(), fs.writeFileSync() etc.	Examples: fs.readFile(), fs.appendFile(), fs.writeFile(), fs.stat() etc.

Integration of notification Service

In previous session you have already know how to create a notification service
Here we will be using Post call to the below api and request mentioned below to call the notification service from CRM app

```
POST /notifiServ/api/v1/notifications
```

```
Headers :
 Content-Type:application/json
Sample request body :
{
    "subject" : "Ticket with id [ id ] has been created",
    "content" : "Ticket has been created. User is having some issues",
    "recepientEmails" : "xyz@gmail.com, pqr@linkeddin.com",
    "requester" : "Vishwa Mohan",
    "ticketId" : "qqwe2314355521"
}

Sample response body :
{
    "requestId": "qqwe2314355521",
    "status": "Accepted Request"
}
```

Here there are two thing.

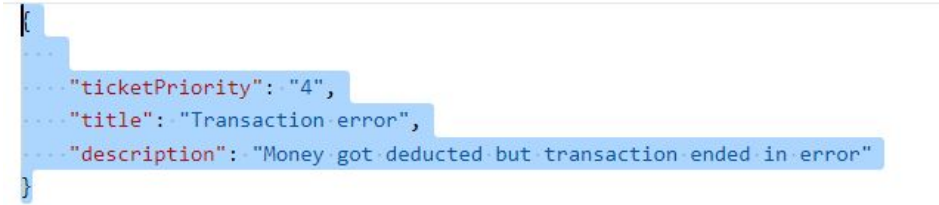
1. Calling notification Service when Ticket is created
2. Calling notification Service when Ticket is updated

1. ticket.controller.js

```
/**
 * This is the controller for the ticket resource
 */
const User = require("../models/user.model");
const Ticket = require("../models/ticket.model");
const constants = require("../utils/constants");
const objectConvertor = require("../utils/objectConcverter");
const sendEmail = require("../utils/NotificationClient");


/**
 * Create a ticket :
 *   As soon as ticket is created, it will be assigned an Engineer if present
 */
exports.createTicket = async (req, res) => {

    const ticketObject = {
        title: req.body.title,
        ticketPriority: req.body.ticketPriority,
        description: req.body.description,
        status: req.body.status,
        reporter: req.userId //this will be retrieved from the middleware
    }

    /**
     * Logic to find an Engineer in the Approved state
     */
    const engineer = await User.findOne({
        userType: constants.userTypes.engineer,
        userStatus: constants.userStatus.approved
    });
    ticketObject.assignee = engineer.userId;

    try {
        const ticket = await Ticket.create(ticketObject);
        console.log(req.userId);

        if (ticket) {
            //Updating the customer
            const user = await User.findOne({
                userId: req.userId
            });
            user.ticketsCreated.push(ticket._id);
            await user.save();

            //Updating the Engineer
            engineer.ticketsAssigned.push(ticket._id);
            await engineer.save();

            /**
             * Sending the notification to the assigned Engineer in asynchronous manner
             */
            sendEmail(ticket._id,"Ticket with id: " +ticket._id +" created",ticket.description, user.email + "," + engineer.email,user.email);


            res.status(201).send(objectConvertor.ticketResponse(ticket));
        }
    } catch (err) {
        console.log("Some error happened while creating ticket", err.message);
        res.status(500).send({
            message: "Some internal server error"
        })
    }

};
```

1. ticket.controller.js

- Here we are importing ticket.model as we are using it for creating an entry in the database and fetching/getting entries from the database.
- We are importing NotificationClient to do post call to the notification service with all the required details to send mail to the assignee
- A notificationClientis created that has all the required email details and is used with post call to create notification when created
- First will create ticket Object
- In the class you can see the create ticket async function which gets the required ticket details such as



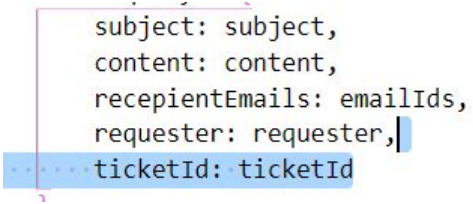
1. ticket.controller.js

- First we will check whether he is an engineer userType. If so req.userId is got from the middleware by decoding the token
- Then assign the particular user with ticket
- Call the send mail which is in turn calling the Notification service with below required details for example

```
sendEmail(ticket._id,"Ticket with id: " +ticket._id +" created",ticket.description, user.email + "," + engineer.email,user.email);
```

2. NotificationClient.js

- Below is the request body to create post request for creation of notification service

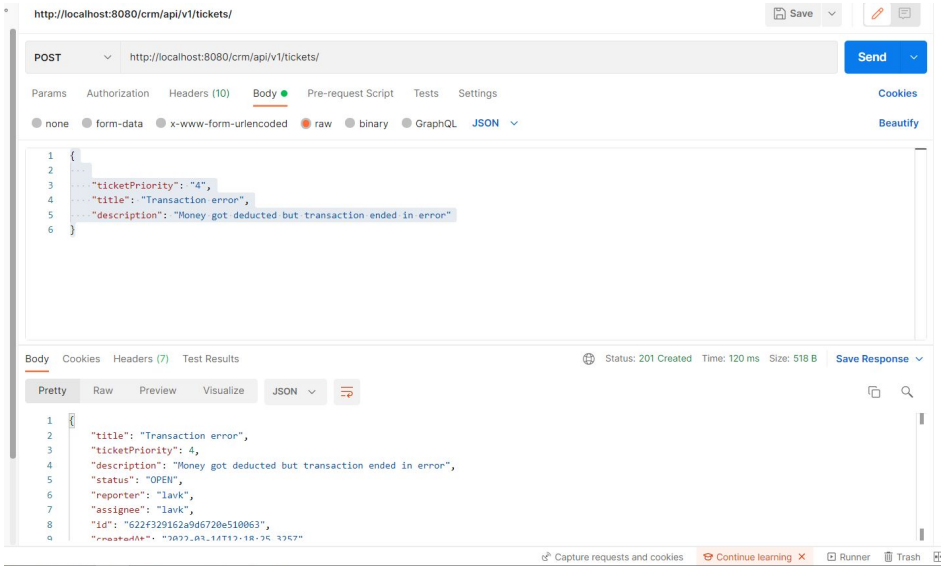


2. NotificationClient.js

Here an Example of how it works

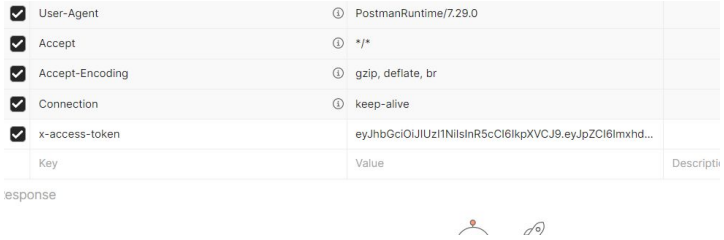
I created a user with lavk and set the password/

After that you can send the



2. NotificationClient.js

Header Details



2. NotificationClient.js

Simultaneously notification service is called in ticket to create notification . Parallel a notification service is running on the another port


```
{
  from: 'crm-notification-service@gmail.com',
  to: [ 'techdevatha@gmail.com,techdevatha@gmail.com' ],
  subject: 'Ticket with id: 622f329162a9d6720e510063 created',
  text: 'Money got deducted but transaction ended in error'
}
{
  accepted: [ 'techdevatha@gmail.com' ],
  rejected: [],
  envelopeTime: 1119,
  messageTime: 854,
  messageSize: 401,
  response: '250 2.0.0 OK  1647260314 j7-20020a056a00130700b004b9f7cd94a4sm20013884pfu.56 - gsmtp',
  envelope: {
    from: 'crm-notification-service@gmail.com',
    to: [ 'techdevatha@gmail.com' ]
  },
  messageId: '<54d6712d-ed0f-3e96-bf4f-5f687ff9208f@gmail.com>'
}
```


You can see That notification service is called when ticket is created

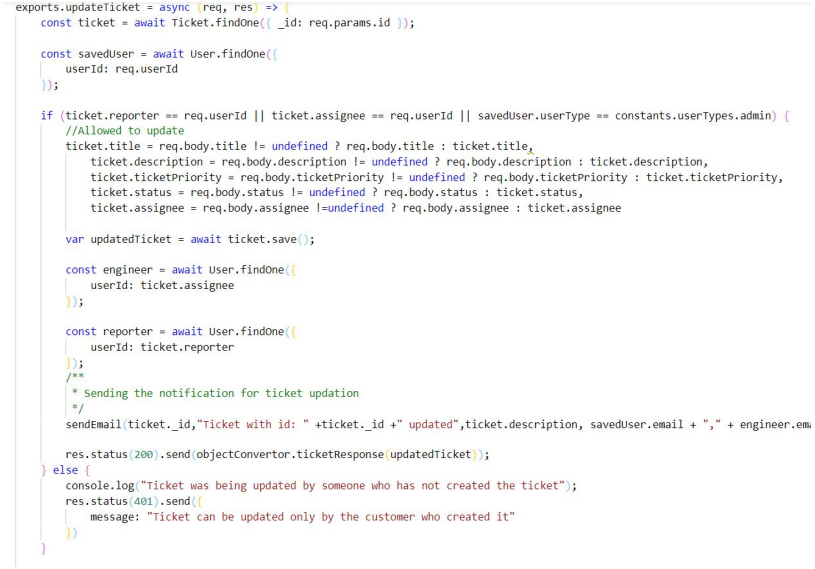
Call the Notification service every time a ticket is updated, to send email to the reporter and assignee

- The same notification client object is also called in case of ticket update as well

```
exports.updateTicket = async (req, res) => {
    const ticket = await Ticket.findOne({ _id: req.params.id });

    const savedUser = await User.findOne({
        userId: req.userId
    });

    if (ticket.reporter == req.userId || ticket.assignee == req.userId || savedUser.userType == constants.userTypes.admin) {
        //Allowed to update
        ticket.title = req.body.title != undefined ? req.body.title : ticket.title,
            ticket.description = req.body.description != undefined ? req.body.description : ticket.description,
            ticket.ticketPriority = req.body.ticketPriority != undefined ? req.body.ticketPriority : ticket.ticketPriority,
            ticket.status = req.body.status != undefined ? req.body.status : ticket.status,
            ticket.assignee = req.body.assignee !=undefined ? req.body.assignee : ticket.assignee

        var updatedTicket = await ticket.save();

        const engineer = await User.findOne({
            userId: ticket.assignee
        });

        const reporter = await User.findOne({
            userId: ticket.reporter
        });
        /**
         * Sending the notification for ticket updation
         */
        sendEmail(ticket._id,"Ticket with id: " +ticket._id +" updated",ticket.description, savedUser.email + "," + engineer.em

        res.status(200).send(objectConvertor.ticketResponse(updatedTicket));
    } else {
        console.log("Ticket was being updated by someone who has not created the ticket");
        res.status(401).send({
            message: "Ticket can be updated only by the customer who created it"
        })
    }
```

Call the Notification service every time a ticket is updated, to send email to the reporter and assignee

- As soon as update ticket put service is called Notification service is triggered



```
exports.updateTicket = async (req, res) => {
    const ticket = await Ticket.findOne({ _id: req.params.id });

    const savedUser = await User.findOne({
        userId: req.userId
    });

    if (ticket.reporter == req.userId || ticket.assignee == req.userId || savedUser.userType == constants.userTypes.admin) {
        //Allowed to update
        ticket.title = req.body.title != undefined ? req.body.title : ticket.title,
            ticket.description = req.body.description != undefined ? req.body.description : ticket.description,
            ticket.ticketPriority = req.body.ticketPriority != undefined ? req.body.ticketPriority : ticket.ticketPriority,
            ticket.status = req.body.status != undefined ? req.body.status : ticket.status,
            ticket.assignee = req.body.assignee !=undefined ? req.body.assignee : ticket.assignee

        var updatedTicket = await ticket.save();

        const engineer = await User.findOne({
            userId: ticket.assignee
        });

        const reporter = await User.findOne({
            userId: ticket.reporter
        });
        /**
         * Sending the notification for ticket updation
         */
        sendEmail(ticket._id,"Ticket with id: " +ticket._id +" updated",ticket.description, savedUser.email + "," + engineer.em

        res.status(200).send(objectConvertor.ticketResponse(updatedTicket));
    } else {
        console.log("Ticket was being updated by someone who has not created the ticket");
        res.status(401).send({
            message: "Ticket can be updated only by the customer who created it"
        })
    }
```

Notification Service output

```
{
  from: 'crm-notification-service@gmail.com',
  to: [
    'techdevatha@gmail.com,techdevatha@gmail.com,techdevatha@gmail.com'
  ],
  subject: 'Ticket with id: 622f329162a9d6720e510063 updated',
  text: 'Money got deducted but transaction ended in error'
}
{
  from: 'crm-notification-service@gmail.com',
  to: [
    'techdevatha@gmail.com,techdevatha@gmail.com,techdevatha@gmail.com'
  ],
  subject: 'Ticket with id: 622f329162a9d6720e510063 updated',
  text: 'Money got deducted but transaction ended in error'
}
{
  accepted: [ 'techdevatha@gmail.com' ],
  rejected: [],
  envelopeTime: 1337,
  messageTime: 907,
  messageSize: 424,
  response: '250 2.0.0 OK  1647285663 p17-20020a639511000000b0038108d69e8fsm11350061pgd.53 - gsmtp',
```

Challenges of Hard Coded URL

- It poses a security risk because any developer with code access has the ability to view secrets such as production database credentials and API keys
- Difficult to maintain, if multiple calls is there

Solution: This is can be avoided by placing domain url in environment based on environment and relative path inside the code

For Example

```
let con = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
});
```

Scalability in Node JS

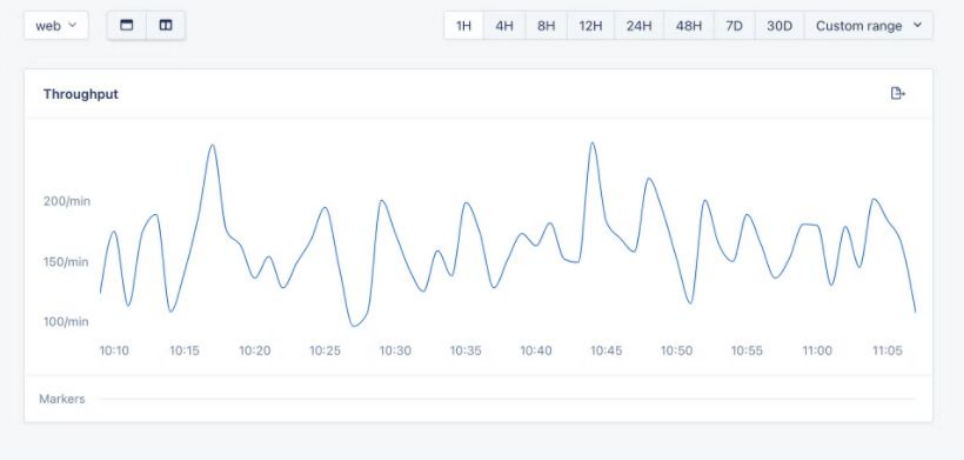
- **Caching**

One of the most common strategies for improving the performance of a web application is server-side caching. Its primary goal is to speed up data retrieval by either spending less time computing such data or performing I/O.

A relatively simple method for implementing caching in a Node. The js application is cached in-process using a solution such as node-cache. It entails storing frequently used data in memory so that it can be retrieved more quickly

Scalability in Node JS

- Use Clustering to Improve Throughput



Scalability in Node JS

Clustering is a technique that allows a Node.js server to be horizontally scaled on a single machine by spawning child processes (workers) that run concurrently and share a single port. It is a common strategy for reducing downtime, slowdowns, and outages by distributing incoming connections across all available worker processes so that available CPU cores are fully utilised.

```
const cluster = require("cluster");
const http = require("http");
const process = require("process");
const os = require("os");

const cpus = os.cpus;

const numCPUs = cpus().length;

if (cluster.isPrimary) {
  console.log(`Primary ${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`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http
    .createServer((req, res) => {
      res.writeHead(200);
      res.end("hello world\n");
    })
    .listen(8000);

  console.log(`Worker ${process.pid} started`);
```

Scalability in Node JS

When you run this programme, connections to port 8000 are shared among the worker processes. As a result, the application's request management will be more efficient:

```
$ node server.js
Primary 15990 is running
Worker 15997 started
Worker 15998 started
Worker 16010 started
Worker 16004 started
```

- **Scale across Multiple Machines with a Load Balancer**

The main requirement is that incoming traffic be distributed to the servers via a load balancer.
To avoid a single point of failure, you can even have multiple load balancers pointing to the same set of servers.

MCQs

1. **Why is the Node so reliant on event handlers?**
   A. Reusable
   B. Modular
   C. Asynchronous
   D. synchronous

Answer: C

MCQs

2. **What is the code to print hello in one second??**
    A. setTimeout(function() { console.log(“Hello”); }, 1000);
    B. setTimeout(function() { 1000, console.log(“Hello”); });
    C. setTimeout(function(1000) { console.log(“Hello”); });
    D. setTimeout(function() { console.log(“Hello”); });

Answer: D

MCQs

3. **Why does Node not block while awaiting the completion of operations??**
    A. synchronous
    B. Asynchronous
    C. Static
    D. Recursive

Answer: B

MCQs

4. What is the below URL performing in our code when requested by the Customer?

GET /crm/api/v1/tickets/

A. Fetching all the tickets raised by the Customer [Correct Answer]

B. Fetching all the tickets assigned to Engineer

C. A and B

D. None

Answer: A

MCQs

5. **How to avoid hardcoded URL**
    A. Using ENV
    B. Specifying as constant in node js class

Answer: A

MCQs

6. **How can you improve the performance in case load increase**
   A. clustering
   B. caching
   C. both a & b
   D. None of the above

Answer: C

Practice Problems

- Modify the get api of tickets to fetch the response of notifications

Thank You!