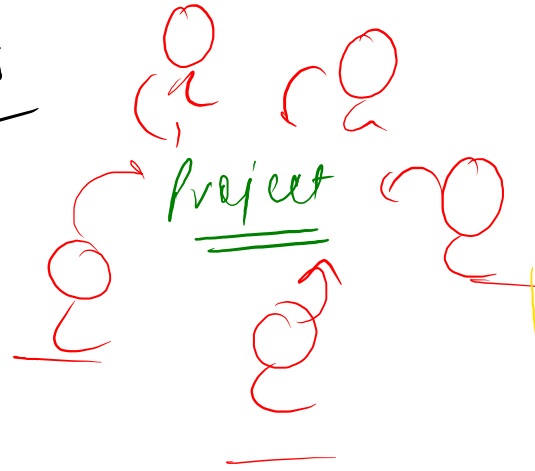
Express → Framework

Java → spring

Python → Django

Industries



① easily collaborate

② Avoid boiler plate code

Holi

Gujiya

GM

GM

GM

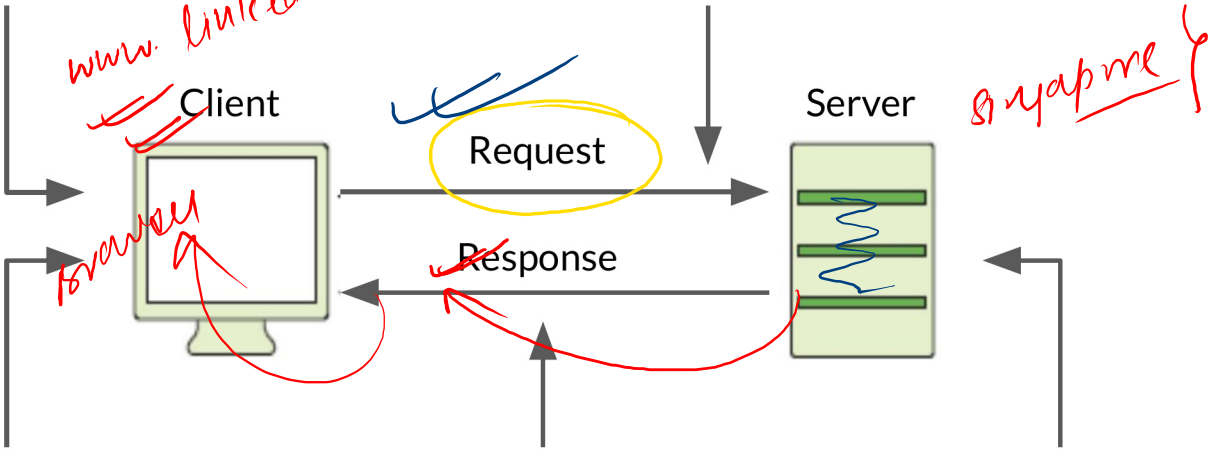
Gujiya Makers

INTRODUCTION TO EXPRESS

Let's understand the request-response cycle

1. User enters a URL in their browser

2. An HTTP request is submitted to the URL's corresponding server.



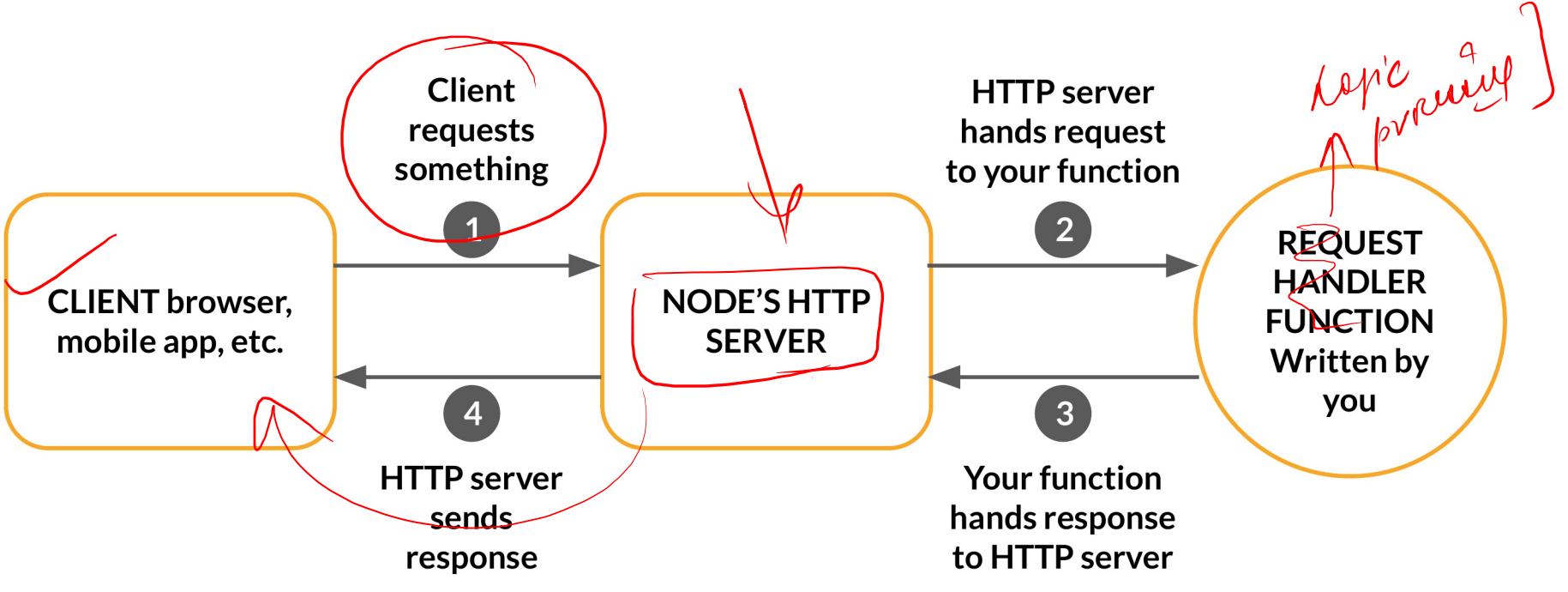
3. A physical machine operates server logic on the user's request to determine what to send back

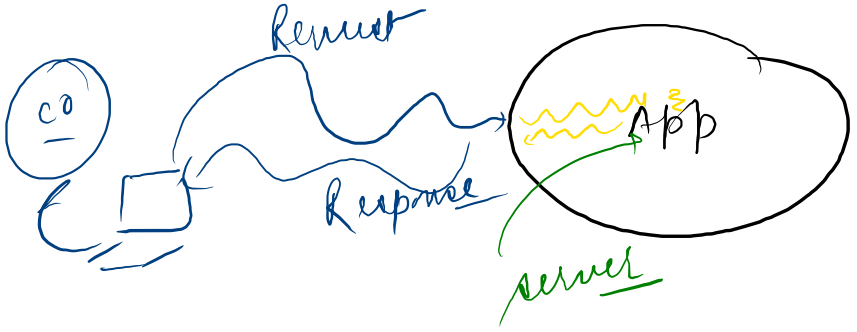
4. An HTTP response is sent to the user (client) in the form of HTML, JSON, plain text, or other formats.

5. User receives a response, usually in the form of an HTML page, which renders on their browser window.

INTRODUCTION TO EXPRESS

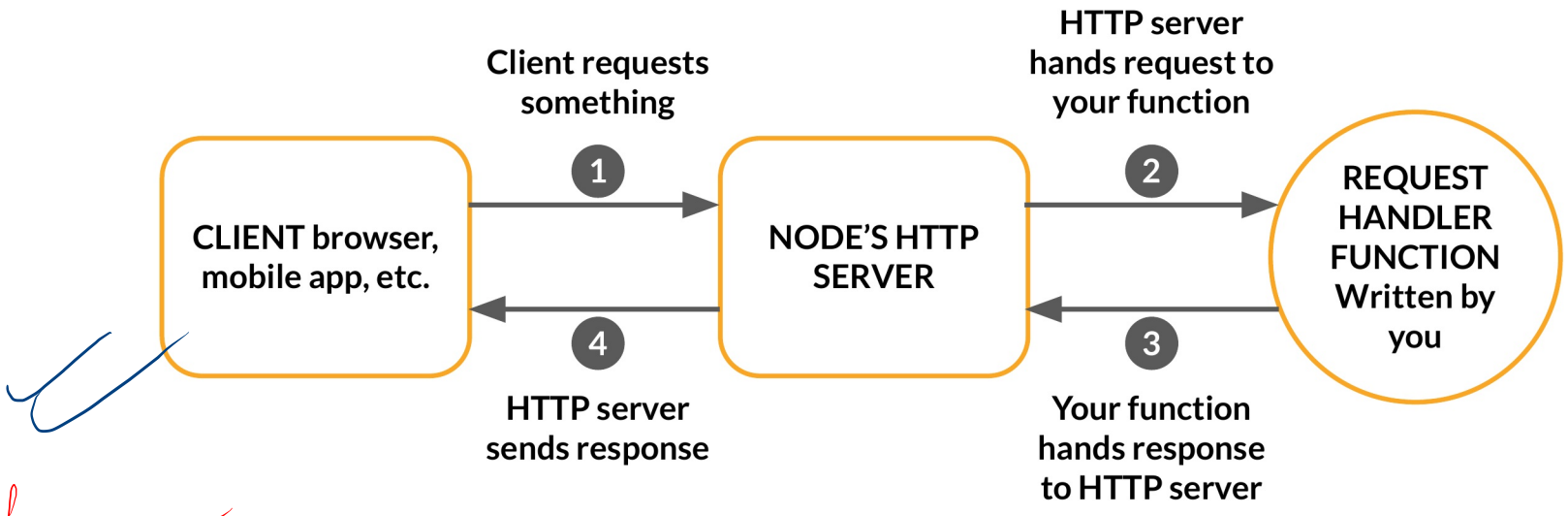
Request - response cycle handled in the plain Node.js application




Request
Response
App
server

INTRODUCTION TO EXPRESS

Request - response cycle handled in the plain Node.js application



Challenges in using only Node.js:

- As the number of types of client requests increases, handling those gets complex
- Many of the commonly used features such as reusable HTML templates require a lot of code manipulations
- There is lots of boilerplate code

WHY USE EXPRESS?

Express is the framework that can improve the experience of creating any web application/Restful APIs

Advantages of using Express in your application:

- Faster server-side development
- Pluggable middlewares
- Support for the templating engines
- Clean routing supports
- Debugging support

GETTING STARTED WITH EXPRESS

Setup

Express, like any other dependency, has to be installed first to be used in your project

The steps involved in installing Express are as follows:

- Create a Node.js project inside a designated folder

```
npm init
```

- Run the following command under the root folder

```
npm install express
```

GETTING STARTED WITH EXPRESS

What happens when we trigger npm install express?

- Changes the package.json
- Downloads Express and all its transitive dependencies and puts it inside the **node_modules** folder under the root directly

```
{
  "name": "express_demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.1"
  }
}
```

GETTING STARTED WITH EXPRESS

Using Express in the application

One of the distinctive feature of the Express framework is that it is a non-opinionated way of building applications. That way, Express turns out to be highly flexible and pluggable

Let's see how we can uncover the complete potential of the Express framework

To use Express, we first need to '**require express**' in the concerned file:

```
const express = require("express")
```

GETTING STARTED WITH EXPRESS

Creating a small Express application

- Create a file app.js inside the node project we created

```
var express = require('express')
var app = express()
app.get('/', function (req, res) {
 res.send('Express.js simple demo example');
})
app.listen(8000)
```

- Execute this file

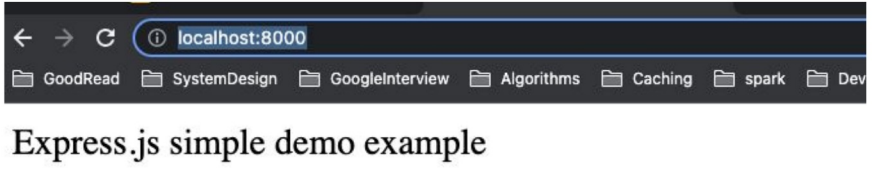
```
 node app.js
```

GETTING STARTED WITH EXPRESS

Creating a small Express application

Check the output:

Open the browser and type in http://localhost:8000/



You would be able to see the response of this application over the browser

GETTING STARTED WITH EXPRESS

Let's see what happened internally

```
var express = require('express')
var app = express()
app.get('/', function (req, res) {
 res.send('Express.js simple demo example');
})
app.listen(8000)
```

- We are importing the Express module to the app.js file so that it can be used for creating our simple web application

GETTING STARTED WITH EXPRESS

Let's see what happened internally

```
var express = require('express')
var app = express()
app.get('/', function (req, res) {
 res.send('Express.js simple demo example');
})
app.listen(8000)
```

- **express** returned from **require('express')** is of the type **function**. So, we need to execute it and store the output of the **express() as app.**
- This **app** will be used to perform all the operations in the Express application

GETTING STARTED WITH EXPRESS

Let's see what happened internally

```
var express = require('express')
var app = express()
app.get('/', function (req, res) {
 res.send('Express.js simple demo example');
})
app.listen(8000)
```

Defining the routes:

- If someone hits the endpoint /, the callback function (which is the second argument) will get triggered
- **res.send**
  - It will send the response back to the client
  - It checks the data that you send and sets the correct response headers

GETTING STARTED WITH EXPRESS

Let's see what happened internally

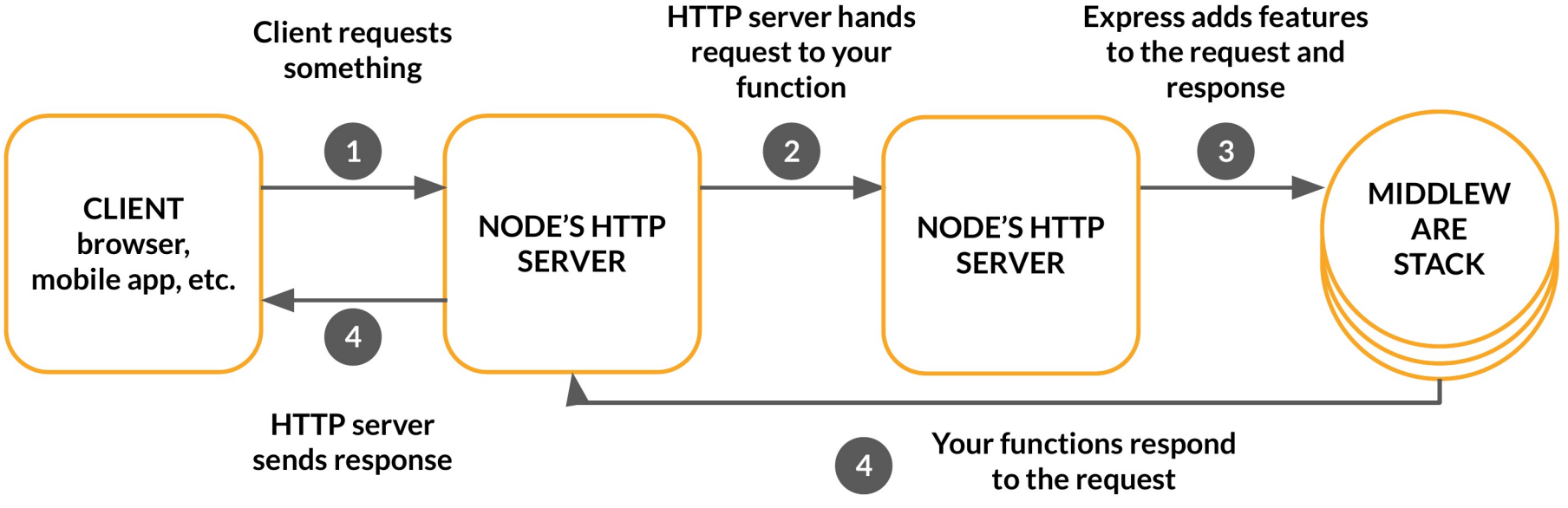
```
var express = require('express')
var app = express()
app.get('/', function (req, res) {
 res.send('Express.js simple demo example');
})
app.listen(8000)
```

Starting the http server

- Starts the http server that is listening on the port number 8000

GETTING STARTED WITH EXPRESS

On a high level, a generic Express application internally works like this:



GETTING STARTED WITH EXPRESS

The core components of any express application are middleware, routes and sub-applications

Middleware

- These are functions that have a hold on ‘Request’ and ‘Response Object’
- Express can easily plug in middleware to enhance its capabilities
- We can either create a middleware or use a third-party handy middleware

CORE COMPONENTS OF ANY EXPRESS APPLICATION: MIDDLEWARE

Let's see what happened internally

```
var express = require('express')
var app = express()

app.use(middleWare);
app.get('/', function (req, res) {
 console.log(req.serverName);
 res.send('Express.js simple demo example from
' + req.serverName );
})
app.listen(8000)

function middleWare(req, res, next){
   console.log("I will modify req object");
   req.serverName = 'upgrad demo server';
   next();
}

app.listen(8000)
```

For example:

- We have created the middleware highlighted in yellow, which intercepts the req object and adds a field serverName to it

CORE COMPONENTS OF ANY EXPRESS APPLICATION: MIDDLEWARE

Important points about the middleware:

- Has access to **req and res** objects
- Multiple middlewares can be present
- **The order of the execution of middlewares is the one in which they are included in the file**

CORE COMPONENTS OF ANY EXPRESS APPLICATION: ROUTES

Routes

- It is a way for servers to provide different types of resources (such as HTML, scripts and images) to the client via different routes
- It a way of intercepting different types of HTTP verbs calls (for example, POST, PUT, GET, DELETE)

CORE COMPONENTS OF ANY EXPRESS APPLICATION: ROUTES

```
var express = require('express')
var app = express()
app.get('/', function (req, res) {
   res.send('Express.js simple demo example');
})
app.listen(8000)
```

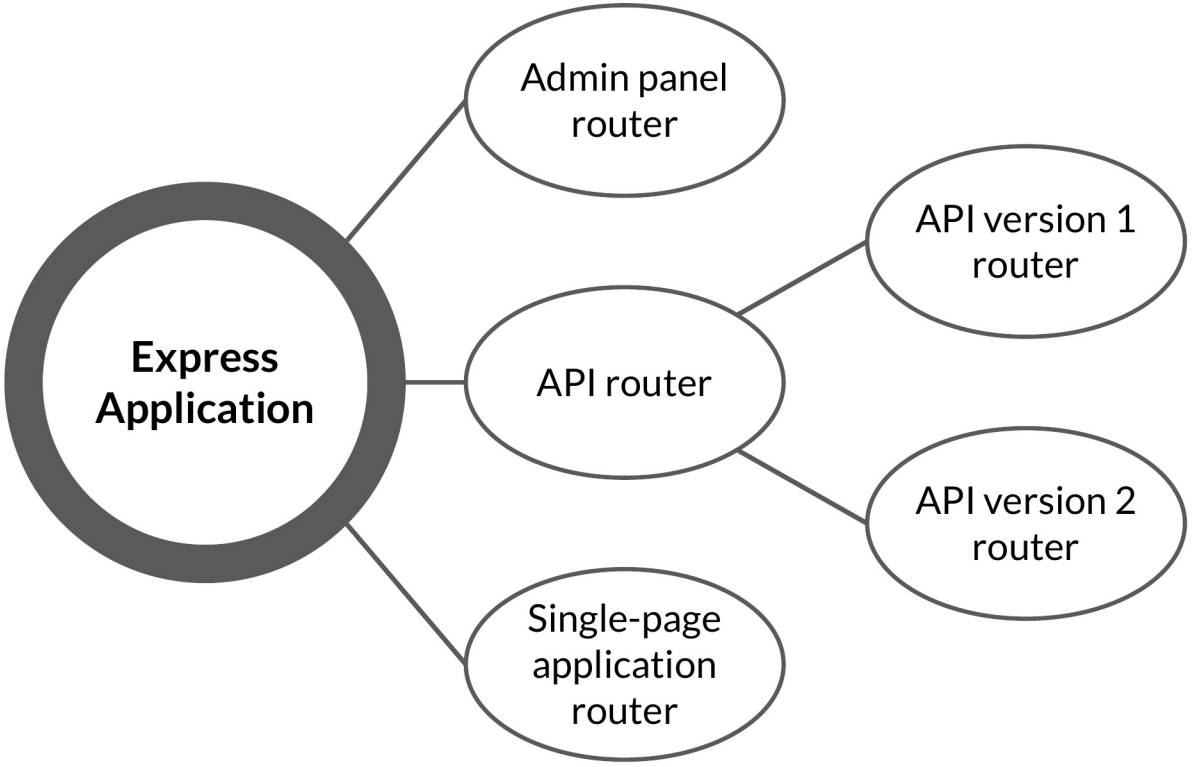
- For example, the highlighted portion in the code snippet on the left is a very simple example of the routes
- So, any client making a GET for the endpoint “/” will be handled by the callback function that we have provided as the second argument of the app.get() method

CORE COMPONENTS OF ANY EXPRESS APPLICATION: SUB-APPLICATIONS

Sub-applications

- Structure the whole project
- Break the routes into multiple logical routes
- Help in a better understanding and maintenance of the application

CORE COMPONENTS OF ANY EXPRESS APPLICATION: SUB APPLICATIONS



For example, we can have separate routes files for handling admin panel and the single page application router.

GETTING STARTED WITH EXPRESS

Other node.js frameworks available in the market: