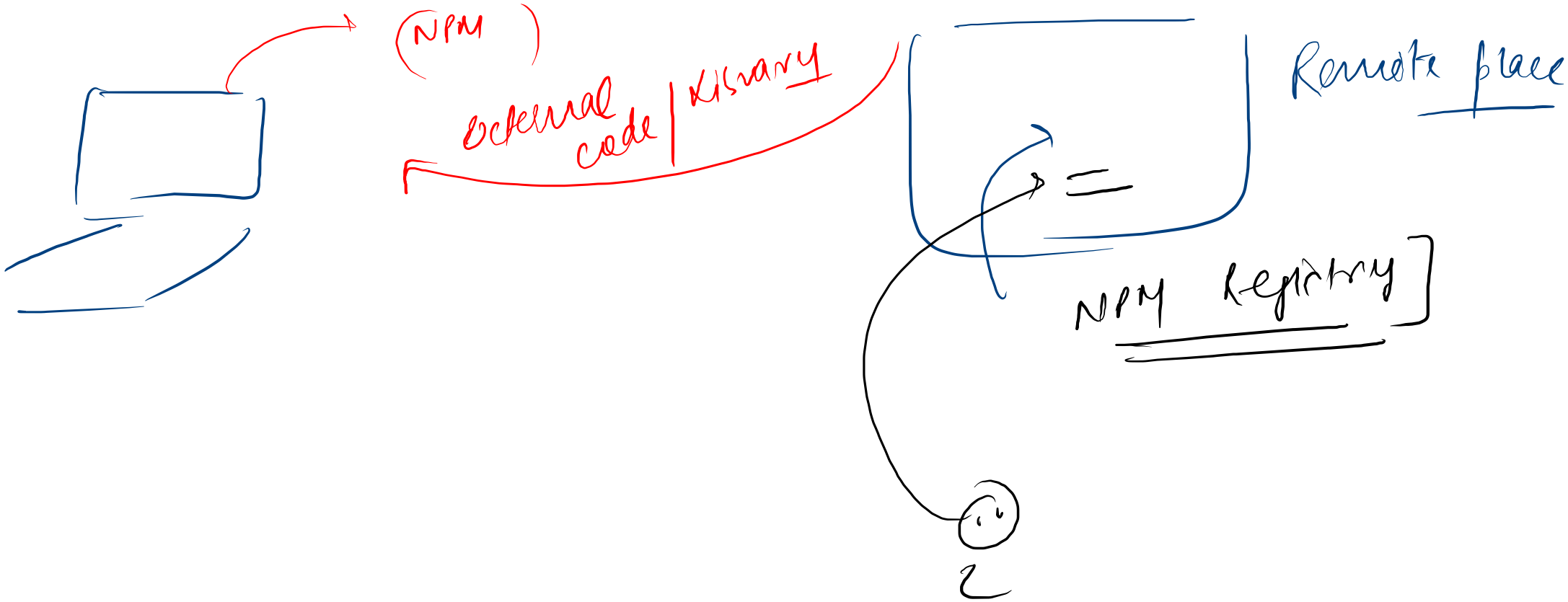
INTRODUCTION TO NPM

- NPM is a **Node Package Manager** for JavaScript programming language. You may have heard of tools such as Maven and Gradle for installing third-party libraries in Java
- Similarly, npm is a package manager that can be used to **install third-party/external node library**
- It is the **default package manager** for the JavaScript runtime environment Node.js

NPM is the world's largest software library


(NPM)
external code | Library
Remote place
NPM Repository

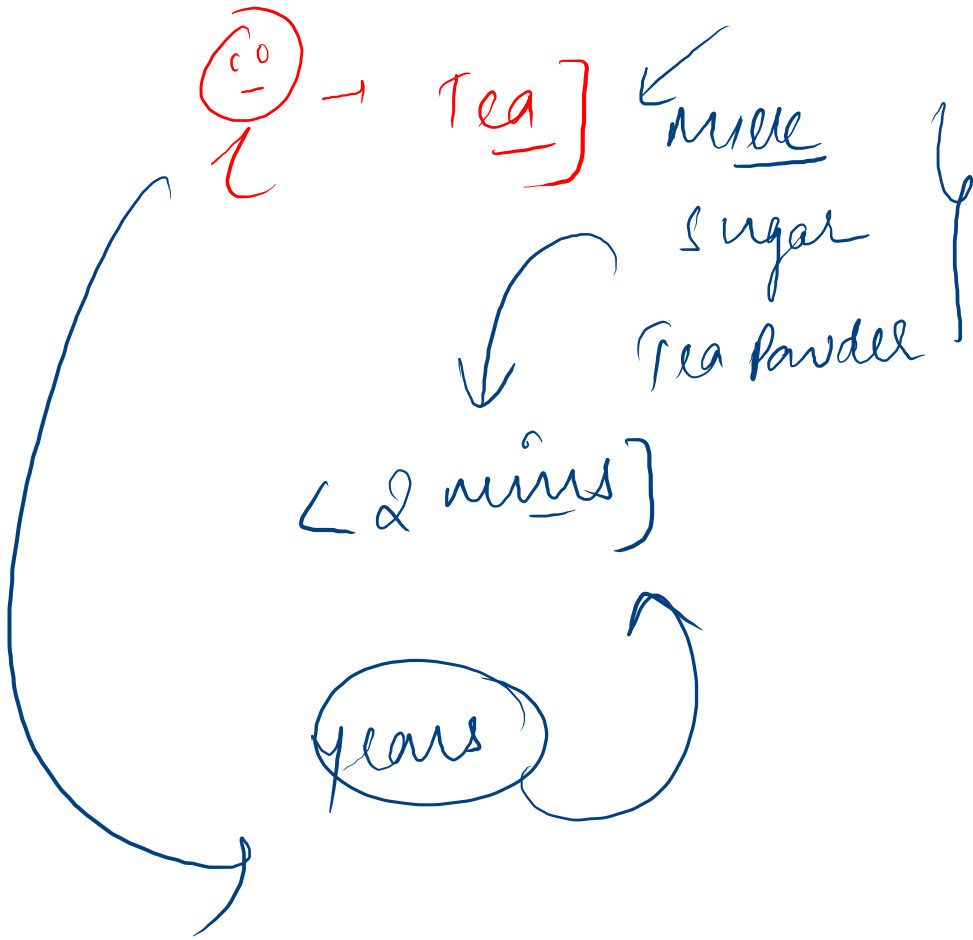
(App)

External codes / Libraries / Modules

Dependency Management

Files / structure

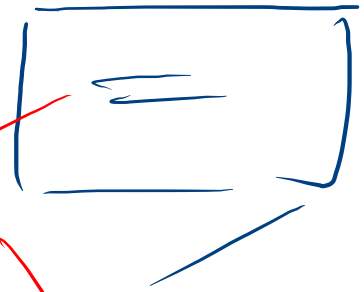
contributors



Java → Maven

Python → PIP

(Node.js) → (JS)



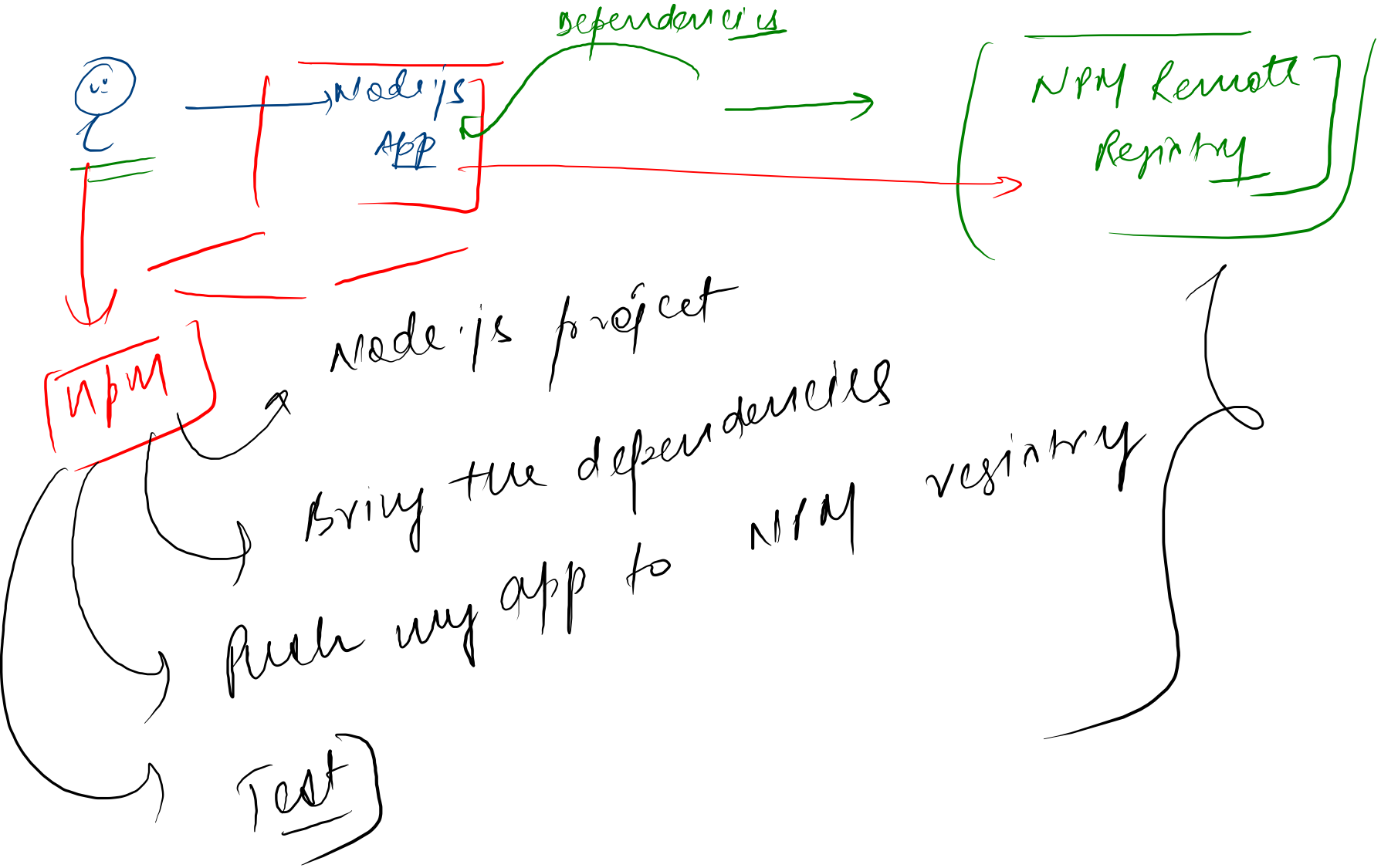
(Package Manager)

NPM → Node Package Manager

INTRODUCTION TO NPM

- The registry is accessed via the client, and the available packages can be browsed and searched via the npm website. The package manager and registry are managed by npm, Inc
- NPM is included as a recommended feature in Node.js installer
- Whenever you install Node.js, npm is included in it by default. You can verify it by running the command mentioned below

npm -v Here v stands for version


Dependencies
Node.js App
NPM Remote Repository
NPM
Node.js project
Bring the dependencies
Push my app to NPM registry
Test

(NPM) → Create a Node.js project

WHAT IS NPM?

Default package Manager

yes

- Many companies use npm to manage private development
- Another package manager, like npm, that we can use to install third-party modules and external libraries is called **YARN (Yet Another Resource Navigator)** package manager

YARN

- NPM is a prerequisite for installing YARN, and you need to use this command to install YARN

install yarn

```
npm install yarn –global
```

- YARN generates a ‘yarn.lock’ file.
- YARN add command logs are brief and much cleaner. They are ordered in the form of a tree. An image of it is attached in the next slide
- To install a package in YARN, we use the command given below

```
yarn add package_name@
version_number
```

YARN → More secure
→ Parallel loading → Lot of time

NPM

FB

① Security threats

② Sequential works

Project

sequentially

D_1

D_2

D_2

D_1 no

Lot of time

NPM VS YARN

NPM	YARN
It is the **default package manager**	Unlike npm, it is **not the default package manager**
You need not install npm; it is included in the Node.js installer	To install YARN, npm needs to be installed
The npm generates a **package-json.lock** file	YARN generates a **yarn.lock** file

NPM VS YARN

NPM	YARN
The npm creates massive output logs whenever you install a dependency	YARN output logs are much cleaner and easier to understand and are in the form of tree
The npm does not have a **'why'** command yet	YARN has a **'why'** command; this command tells why a package has been installed
The npm does not have a license checker that can give an idea of all the licenses used in the project	YARN has a neat license checker. To see them, run **yarn licenses list**
The npm fetches packages from the npm registry during every npm install	YARN stores dependencies locally and fetches the dependencies from the disk during a **yarn add** command

Hands on

NPM VS YARN

Type of Command	NPM Command	YARN Command
Install dependencies	npm install or npm i	yarn
Install package	npm install package_name or npm install package_name@version_number	yarn add package_name or yarn add package_name@version_number
Uninstall package	npm uninstall package_name	yarn remove package_name
Install package as dev dependency	npm install package_name –save-dev	yarn add package_name -dev
Update dev package	npm update package_name or npm update package_name@version_number	Yarn upgrade package_name or yarn upgrade version_number
Install package globally	npm install -g package_name	yarn global add package_name

NPM

module

INSTALLING/UNINSTALLING MODULES IN Node.js

install → uninstall

Module ?

- Refer back to the earlier slides to identify which package manager you need to use for your specific use case. Both Yarn and NPM are package managers that can be used to install modules and dependencies in Node.js

NPM install command	npm install package_name
NPM uninstall command	npm uninstall package_name
Yarn install/uninstall command	yarn add package_name yarn remove package_name

MODULES, DEPENDENCIES AND PACKAGES

Terms commonly used in Node.js:

- **Modules** are single or multiple files that contain code for a single concept, functionality or library
- **Dependencies** are requirements that you need to use an API or a function that is part of an external or internal module then for that you need that module as a dependency
- **Packages** are single modules or collections of multiple modules

Module → file single or multiple file → contains some logic program

could be called and used from anywhere in the project

1. User defined
2. In-built / Natively ~~proj~~ present — os, http, fs
3. External Modules — npm

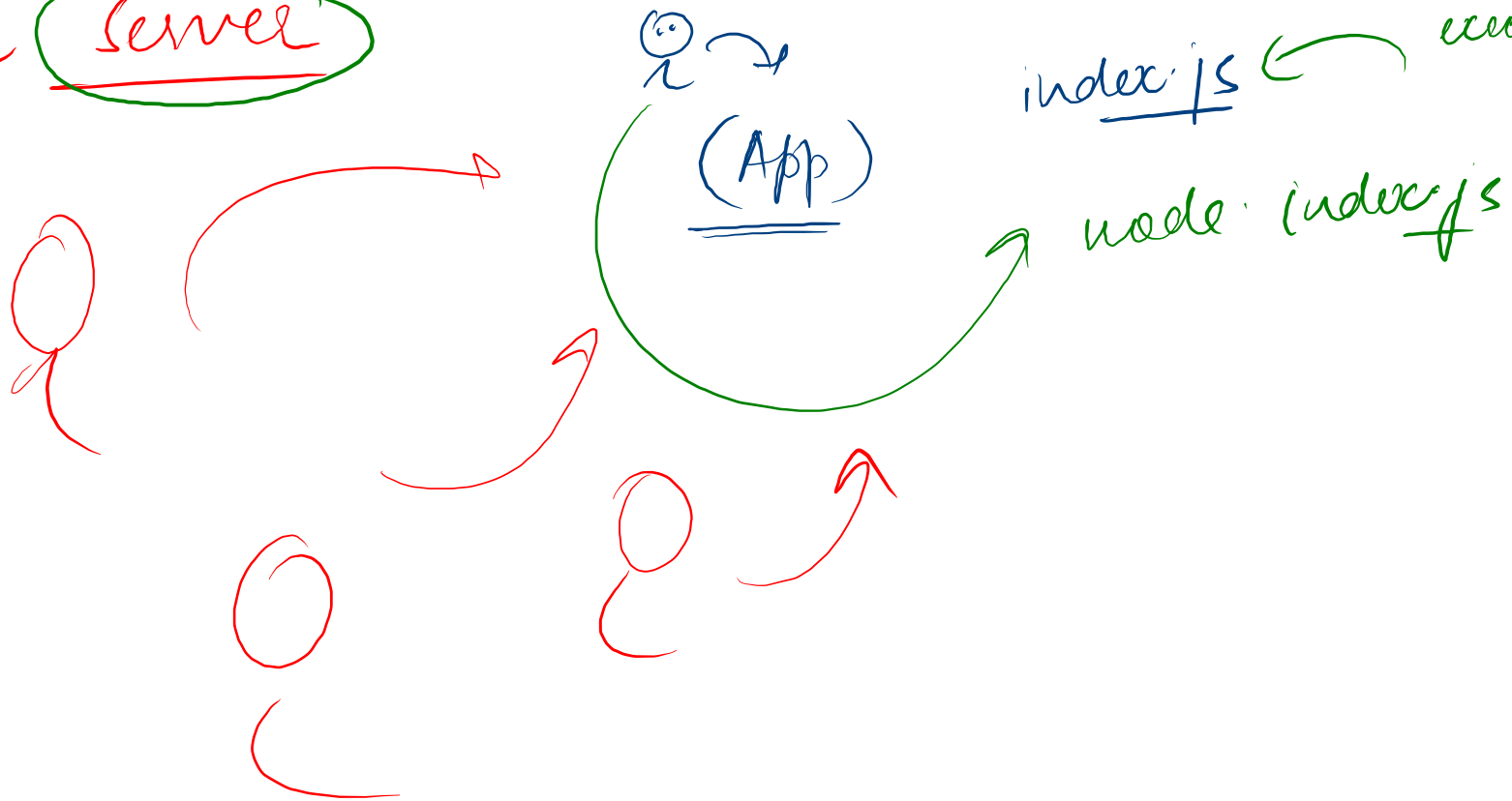
→ create our own custom module]

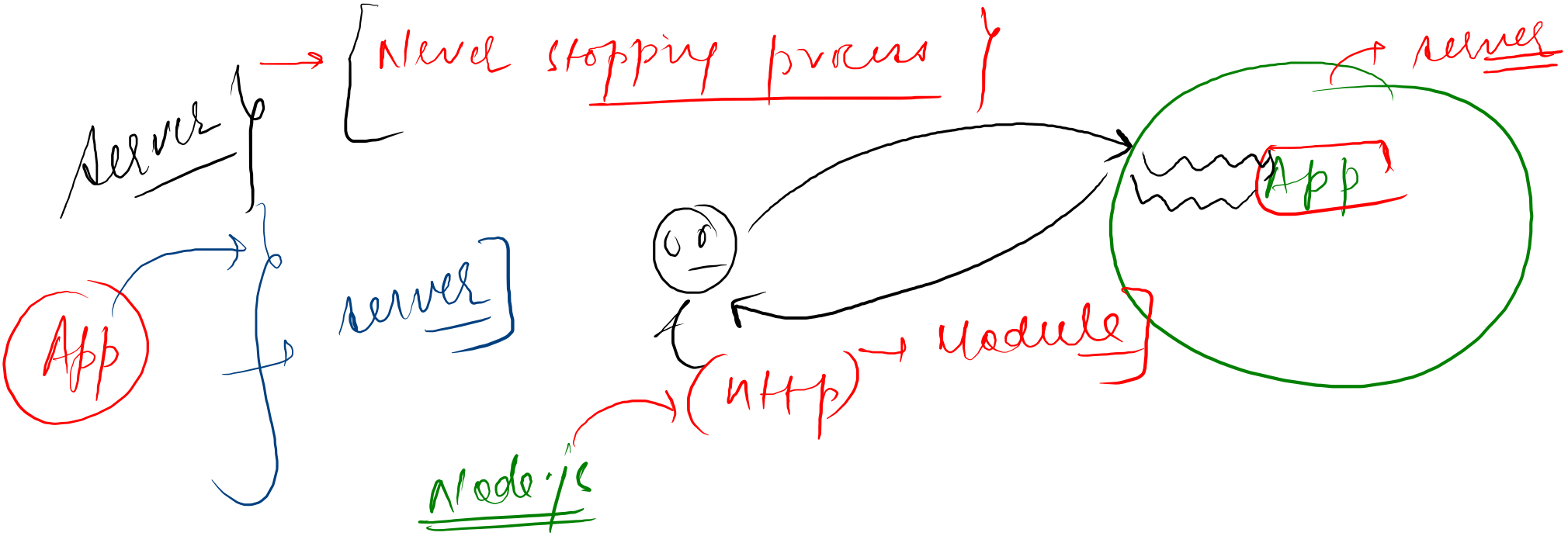
(fs) ← Module

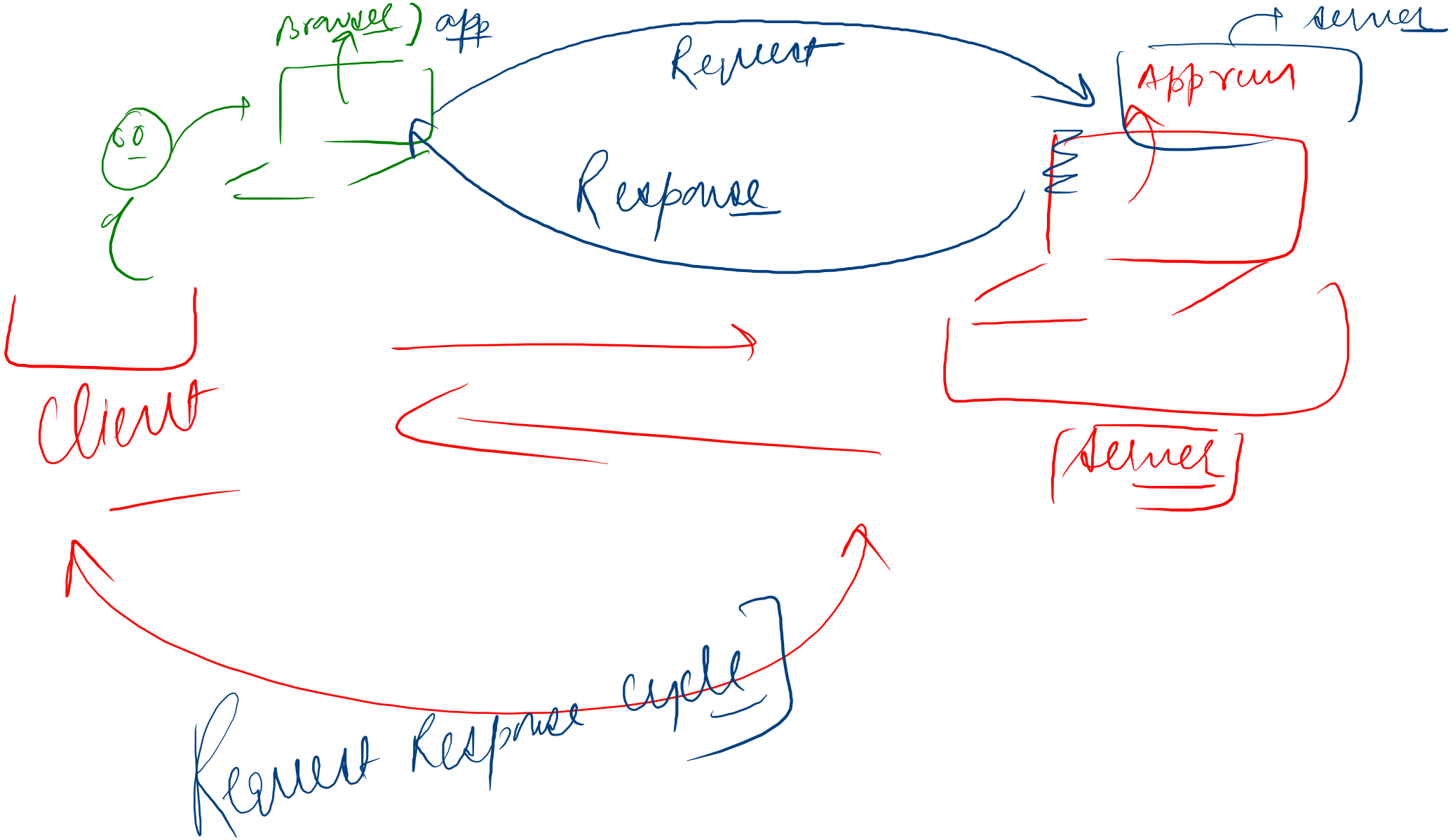
① Read → sync, Async

② write → sync, Async

Client and Server




Server
Never stopping process
server
App
server
App
(http)
module
Node.js


browser) app
Request
Response
server
Application
Client
Server
Request Response cycle

Module

① User defined

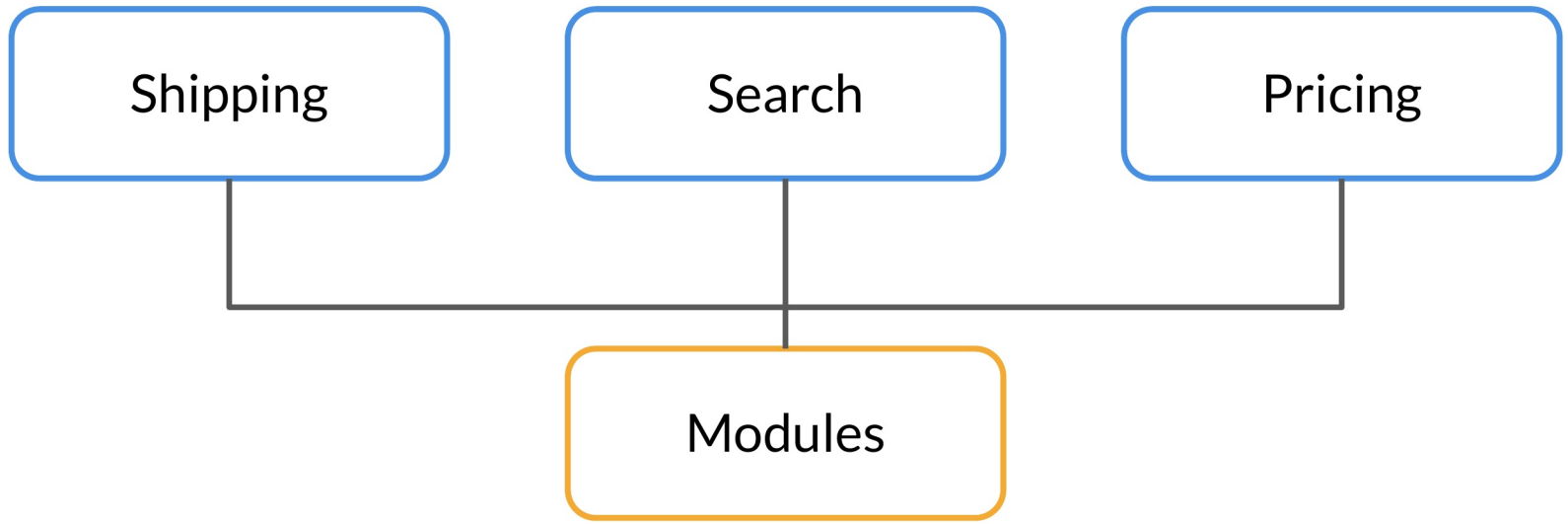
② in-built

npm active/ set

MODULES

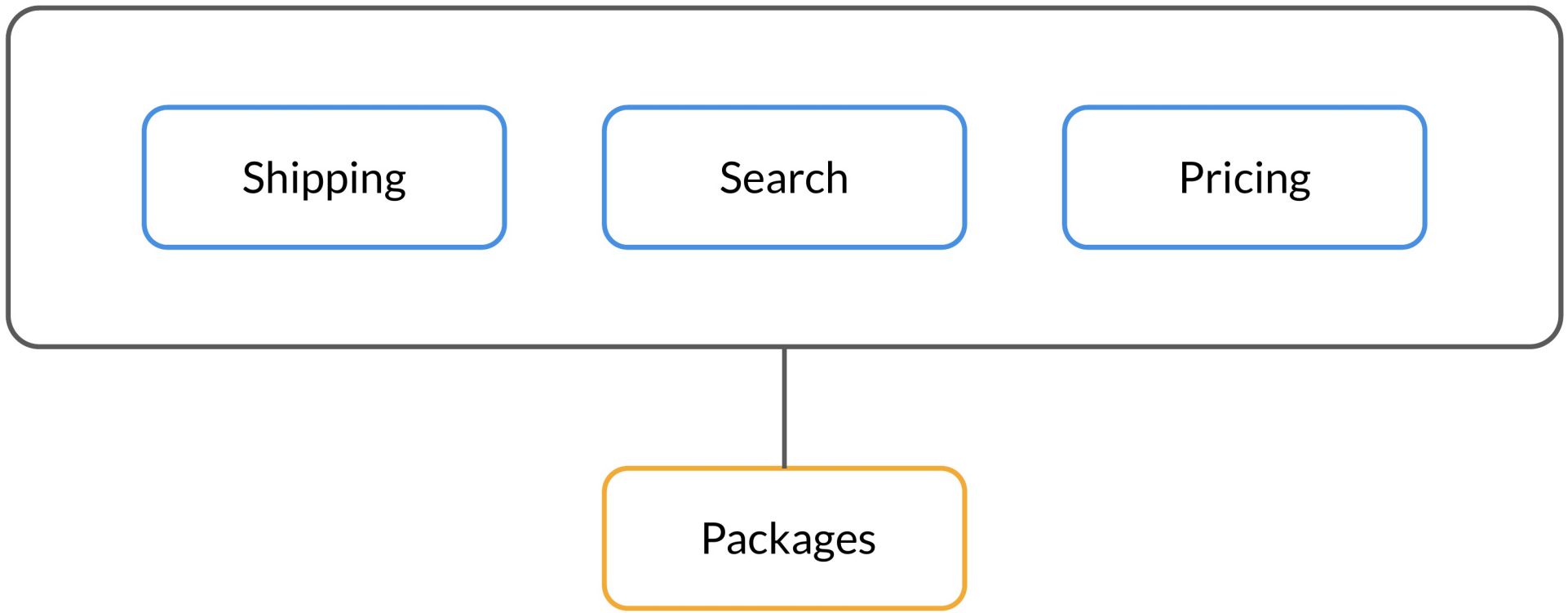
- Modules are nothing but the code you write in a single file and export for use at multiple places

  Example of e-commerce



- Shipping, search and pricing can be exported to be consumed elsewhere in the code

PACKAGES



- **Packages**: A combination of multiple modules is called a package

MODULES AND PACKAGES

- Suppose you want to compress a user's image to upload it on a blob storage such as AWS S3 service
- For this image compression there are two choices as a developer which you can take
  - You write your own image compression module and export that module and import it as a dependency
  - You import a third-party dependency from npm registry, which has over 80K total npm modules
- Go to the official website of npm and search for image compress modules. You will find plenty of options shared publicly by developers across the globe. You can read the documentation and use any one of them depending upon your requirements

  Example: https://www.npmjs.com/package/image-compressor-lib

Modular Programming in Node.js

EXPORTING AND IMPORTING A MODULE

- So far we have talked about modules, packages and dependencies. Let's see how you can write a module and export it so that it can used across a server-side application

- One can also use **exports.sumOfNumbers** to export the module across the application
- **exports** is a shorthand for module.exports

```
Const sumOfNumbers=  (args) => {

let sum=0;
for (let i=0; i<args.length; i++){
 sum+=args[i]
 }
return sum;
}
module.exports=sumOfNumbers;
```

IMPORTING A MODULE

- To import a module, we use '**require**'. Take a look at the example given below:

```
const sumOfNumberFunction = require("./sumOfNumbers");

console.log(sumOfNumbers([1,2,3,4,5]))
```

File Systems and Streams

INBUILT MODULES IN Node.js

- There are lots of **inbuilt modules** in node.js for which you **do not need to install** anything. They are automatically available for your use
- We will be talking about the main modules that you would be using quite frequently in your server-side application development
- **A file system**, also referred to as an **'fs'** module, is used for doing file manipulation operations. With 'fs' module, you can read from already created file synchronously and asynchronously and also write and append into the file synchronously and asynchronously

Reading

```
fs.readFile('file.txt', function(err, data) { .. });
fs.readFile('file.txt', {encoding: 'utf-8'}, function(err, data) { .. });
```

INBUILT MODULES IN Node.js

File System (FS)

Writing

```
fs.writeFile('output.txt', function(err) { .. });
fs.appendFile('output.txt', function(err) { .. });
```

Watch

```
fs.watch('dir OR file.txt', { persistent: true }, function(event, file) {
  event; /* rename | change */
});
```

FILE SYSTEM (FS) MODULE

Getting info

```
fs.exists('file.txt', function(exists /*bool*/) { ... });

fs.stat('file.txt', function(stats) {
  stats.isFile();
  stats.isDirectory();
  stats.isSymbolicLink();
});
```

File operations

```
fs.rename('old.txt', 'new.txt', function(){});
fs.chown('file.txt', uid, gid, function(){});
fs.symlink('src', 'dest', function(){});
fs.unlink('path', function(){});
fs.rmdir('path', function(){});

fs.readdir('path', function(err, files) { .. }); /* `files` = array of names */
```

FILE SYSTEM MODULE

Path

```
fs.realpath('/etc/passwd', function(err, path) { /* "/private/etc/passwd" */ });
```

Sync

```
data = fs.readFileSync('input.txt');
fs.writeFileSync('output.txt', data);
fs.appendFileSync('output.txt', data);
fs.existsSync('file.txt');
```

INBUILT MODULES

- **Path module**: It provides utilities for working with file and directory paths. It can be accessed using

```
const path = require("path")
```

- **OS module**: It provides operating-system-related utility methods and properties.It can be accessed using

```
const os = require("os")
```

- **Http module:** It is used for creating HTTP servers and then receiving requests from and responding to the server. It can be accessed using

```
const http = require("http")
```

EXTERNAL / THIRD-PARTY MODUELS

npm install <module-name>

express

Api

- There are almost **80k third party modules** in node.js
- Whenever we install any third-party module in node.js, by default, we get an entry in our package.json file. This entry can be **part of devDependencies if installed in dev mode or production dependency if not used in dev mode** while installing that dependency
- We also get a folder named **node_modules**, which contains the dependency modules that we have installed by running the **npm install <package_name>** command
- We also get a **package-lock.json file** or **yarn.lock** file depending upon which package manager we use to install dependencies

OVERVIEW OF Package.json

The **package.json** file roughly is divided into three important sections:

- **Scripts**: This contains your scripts that run your code. It can contain scripts to run the server in dev mode or production mode, or simply run your tests (unit or integration tests)

- **devDependencies**: These are the dependencies that get installed when you install it in dev mode

- **Dependencies**: These are dependencies that get installed when you do not specify **–save-dev flag**, which means that all these dependencies comes under the dependencies tag

package.json > {} dependencies > react-router-dom

```
1   {
2     "name": "gl-front-end-app",
3     "version": "0.1.0",
4     "private": true,
5     "dependencies": {
6       "@testing-library/jest-dom": "^5.11.4",
7       "@testing-library/react": "^11.1.0",
8       "@testing-library/user-event": "^12.1.10",
9       "react": "^17.0.2",
10      "react-bootstrap": "^2.0.3",
11      "react-dom": "^17.0.2",
12      "react-router-dom": "^4.3.1",
13      "react-scripts": "4.0.3",
14      "web-vitals": "^1.0.1"
15    },
        ▷ Debug
16    "scripts": {
17      "start": "react-scripts start",
18      "build": "react-scripts build",
19      "test": "react-scripts test",
20      "eject": "react-scripts eject"
21    },
22    "eslintConfig": {
23      "extends": [
24        "react-app",
25        "react-app/jest"
26      ]
27    },
28    "browserslist": {
29      "production": [
30        ">0.2%",
31        "not dead",
32        "not op_mini all"
33      ],
34      "development": [
35        "last 1 chrome version",
36        "last 1 firefox version",
37        "last 1 safari version"
38      ]
39    }
40  }
```

Modules

custom

inbuilt

external module