Node.js

Instructor: Bui Binh Giang

buibinhgiang@vanlanguni.vn

TABLE OF CONTENTS

Introduction to Node.js

Front-end vs Back-end

Static, Dynamic, API-Powered websites

The Node.js architecture

Blocking and Non-Blocking

Synchronous vs Asynchronous

Node.js Process and the Thread Pool

The Event Loop

Event and Event-Driven

Streams

Node.js Modules

Express.js

Routing

The REST API Architecture

Middleware

MVC Architecture

Error Handling

Introduction to Node.js

What is Node.js?

- Node.js is a JavaScript runtime built on Google's open-source V8 JavaScript engine
- Node.js uses JavaScript on the server
- Node.js: JavaScript outside of the browser

JavaScript on the Server

Perfect conditions for using Node.js as a web server

We can use JavaScript on the server side of web development

Build fast, highly scalable network applications (back-end)

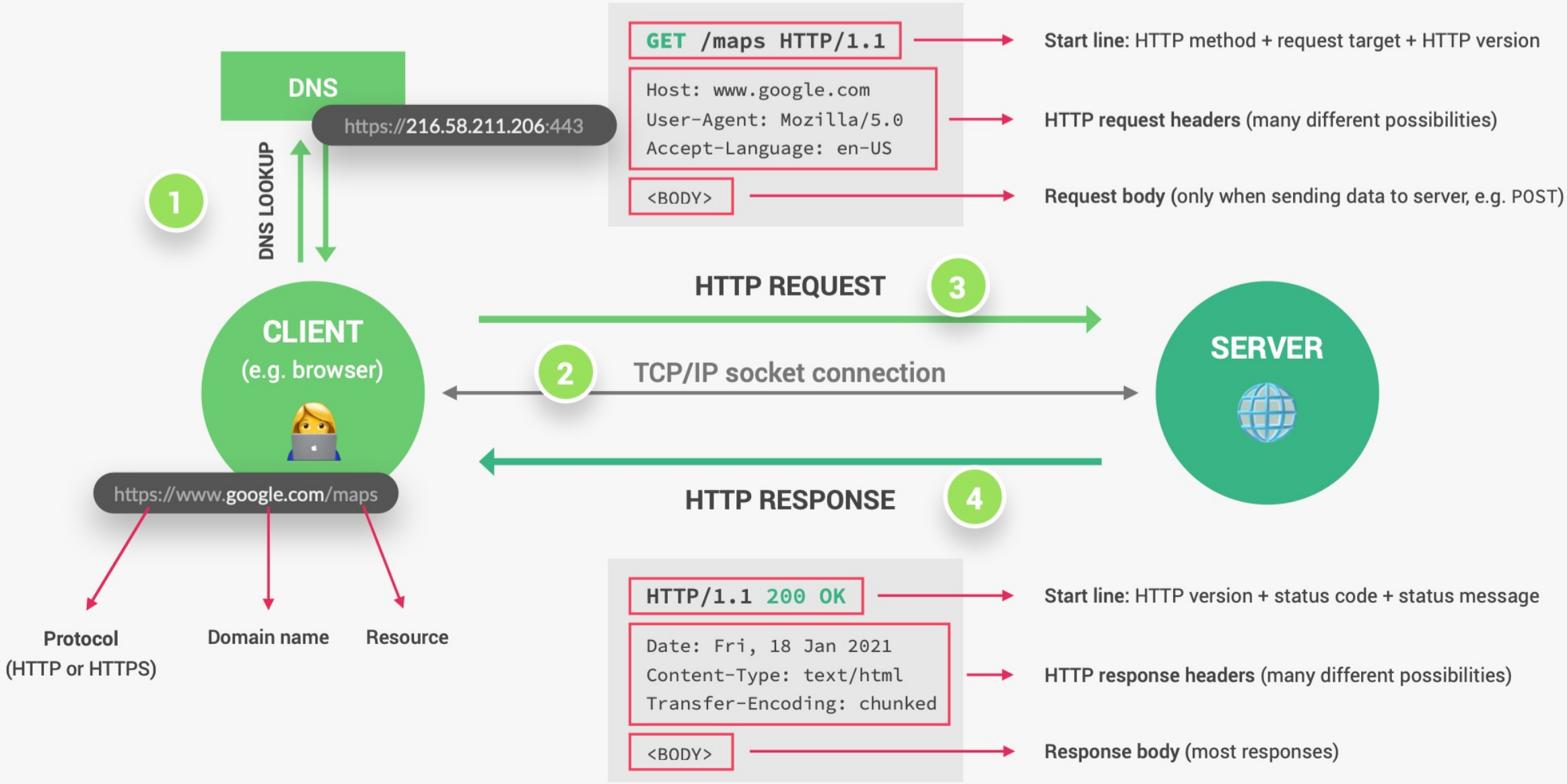
Why and when to use Node.js?

Node.js Pros	Use Node.js
• Single-threaded, based on event driven, non-blocking I/O model • Perfect for building fast and scalable data-intensive apps • JavaScript across the entire stack: faster and more efficient development • NPM: huge library of open-source packages available for everyone for free • Very active developer community	• API with database behind it • Data streaming (think YouTube) • Real-time chat application • Server-side web application **DON'T USE** Applications with heavy server-side processing (CPU-intensive)

How to install Node.js

- Download lasted Node.js from https://nodejs.org/en/download/
- Run file to install

How the web works?



How the web works?

- Request-response model or Client-server architecture
- HTTP - Hyper Text Transfer Protocol: a protocol for transferring data which is understood by Browser and Server
- HTTP Request = Method + Header + Body
- HTTP Response = Status + Header + Body
- HTTPS - Hyper Text Transfer Protocol Secure: HTTP + Data Encryption (during Transmission)

Front-end and Back-end

Front-end (client)	Back-end (server)

• Everything that happens in the web browser
• Designing and building website
• Basic technologies: HTML, CSS, JS
• Modern technologies: React, Angular, Vue, Redux, GraphQL,…
	• Everything that happens in the web server
	• Web server included:
	  ▪ Files (HTML, CSS, images)
	  ▪ HTTP Server (communicates with the client using requests and responses)
	  ▪ Application
	• Application can access the database
	• Technologies: Node.js, Java, Golang,…
	• Databases: MongoDB, MySQL,…

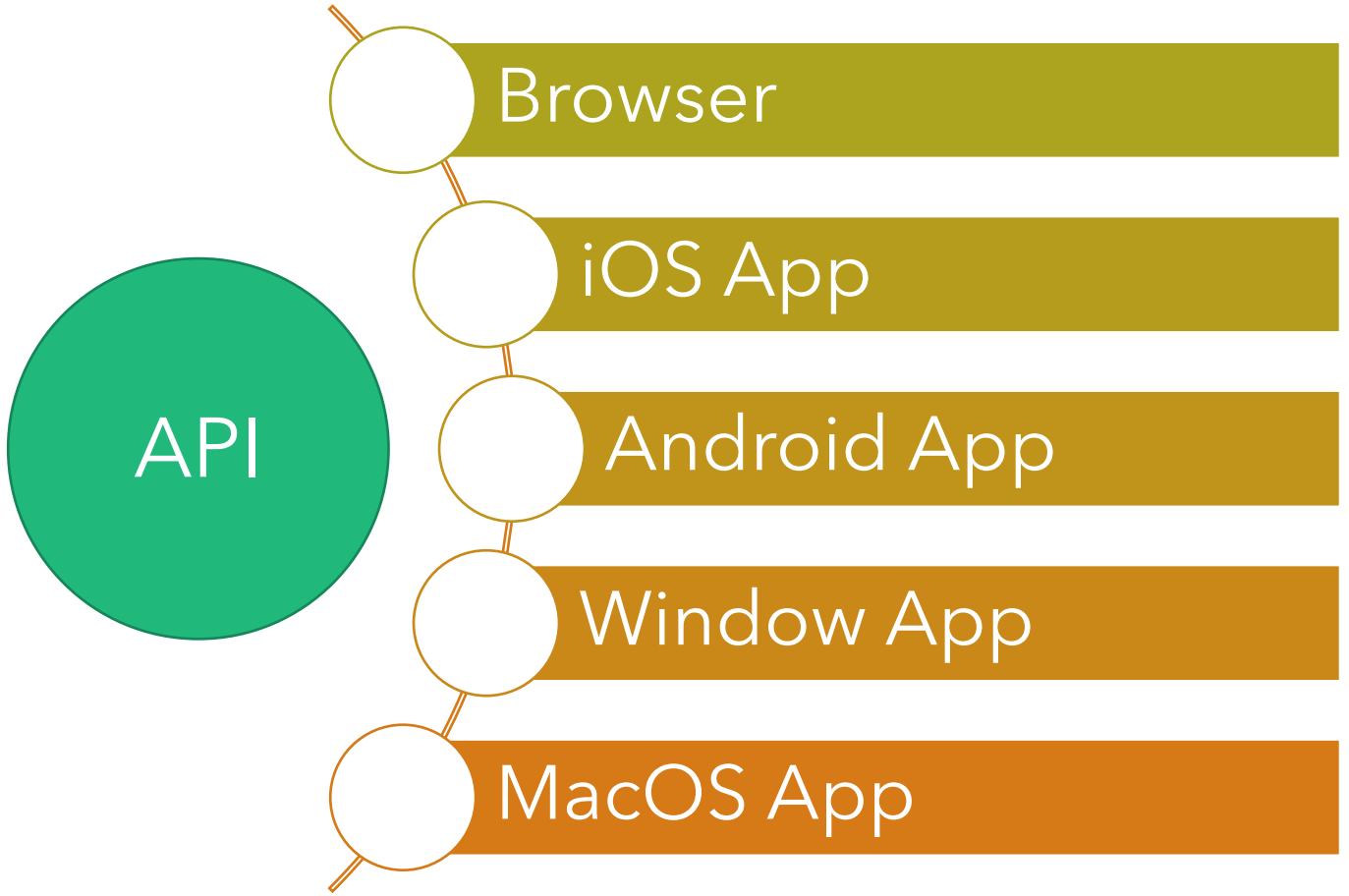
Static websites vs Dynamic websites

Static	Dynamic
• No back-end code • Serving static files (HTML, CSS, JS, images)	• Contain a database, an application back-end running (like Node.js app) • Application fetches data from the database the together with predefined template, builds each page dynamically based on data coming from the database • Pages is built as HTML, CSS, JS files will be sent back to the browser • The whole process is called server-side rendering

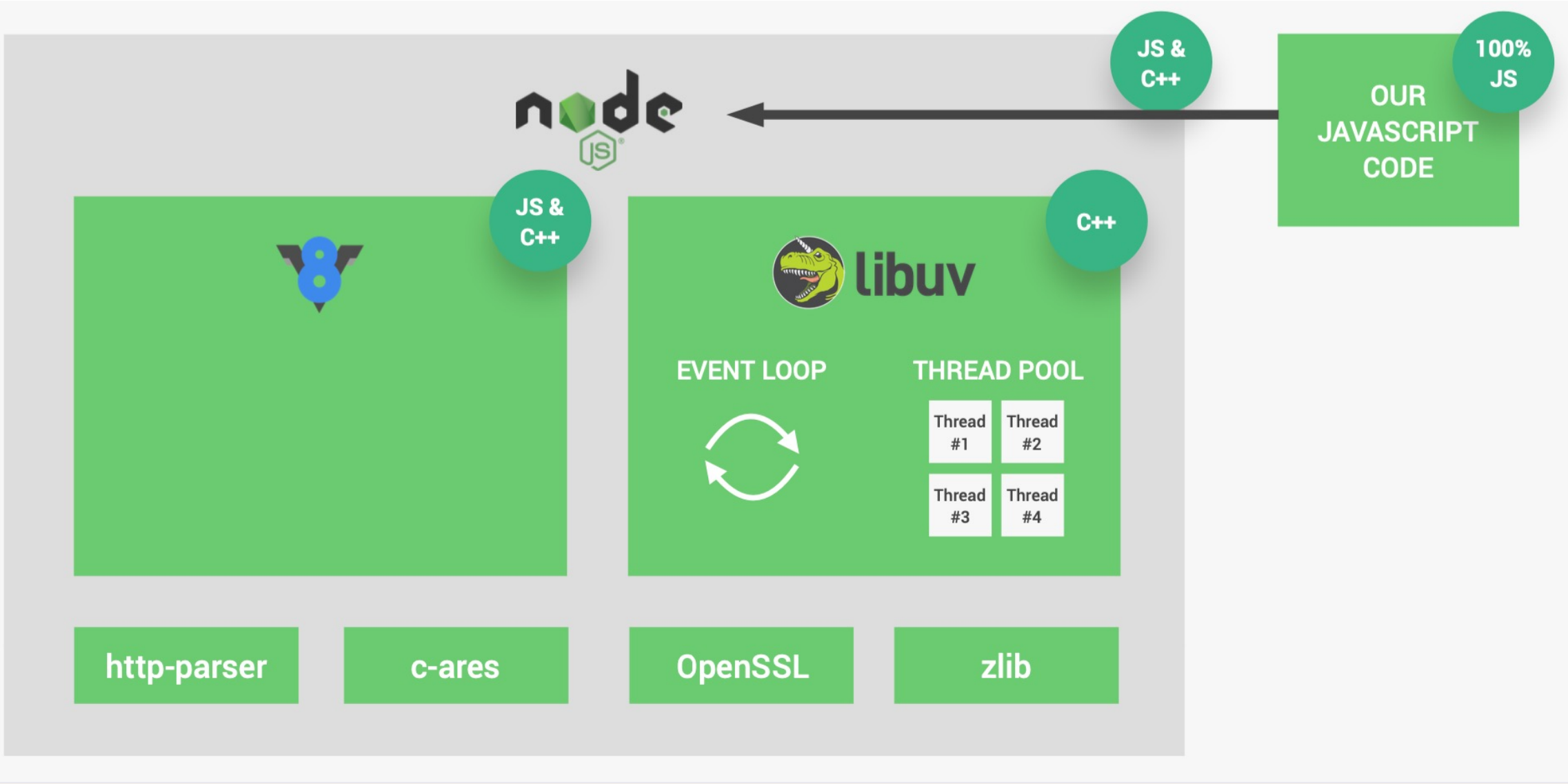
Dynamic websites vs API-Powered websites

Dynamic	API
• Contain a database, an application back-end running (like Node.js app) • Application fetches data from the database then together with predefined template, builds each page dynamically based on data coming from the database • Pages is built as HTML, CSS, JS files will be sent back to the browser • The whole process is called server-side rendering	• Contain a database, an application back-end running (like Node.js app) • Only send the data to the browser, usually in JSON format (no HTML, no CSS , no JS) • API stands for application programing interface, allow application to talk to each other • Server building API, client consuming API • Browser will use received data and template to build a website • The process is called client-side rendering

One API, Many consumer



The Node.js architecture



The Node.js architecture

- Node.js is a collection of dependencies which are responsible for executing our code
- Two of the most important dependencies are V8 and libuv
- V8 engine is responsible for running JavaScript code outside browser
- libuv is responsible for accessing the underlying operating system, file system, networking, and it also solves some extent of concurrency
- Other dependencies: http-parser (parsing http), c-ares (DNS request), openssl (cryptography), zlib (compression)

libuv

- libuv is a library originally written for Node.js to abstract non-blocking I/O operations.
- Included: Event loop, Thread Pool

Blocking I/O vs Non-Blocking I/O

- I/O: input/output: interact with the system's disk, OS and network
- Blocking I/O refers to I/O operations that block further execution until that operation finishes
- Non-Blocking I/O refers to code that doesn't block execution, the I/O operations are delegated to the system, so that the process can execute the next piece of code. It provide a callback function that is called when the operation is completed

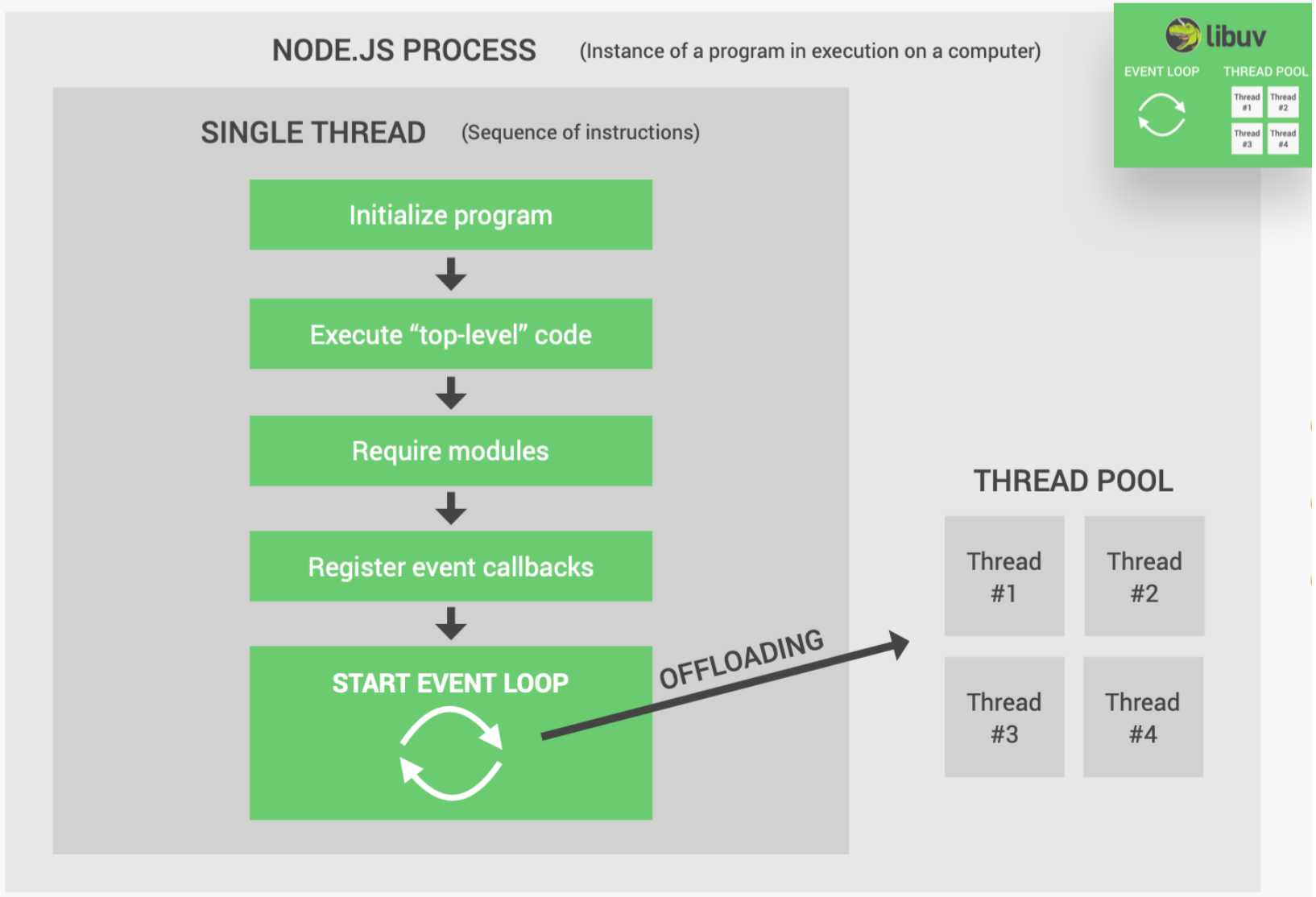
Callbacks

- A callback is a function passed as an argument into another function, which can then be invoked (called back) inside the outer function to complete action at a convenient time

Synchronous vs Asynchronous

- Synchronous (or sync) execution usually refers to code executing in sequence. In sync programming, the program is executed line by line, one line at a time 👉 BLOCKING
- Asynchronous (or async) execution refers to execution that doesn’t run in the sequence it appears in the code. In async programming the program doesn’t wait for the task to complete and can move on to the next task 👉 NON-BLOCKING

Node.js Process

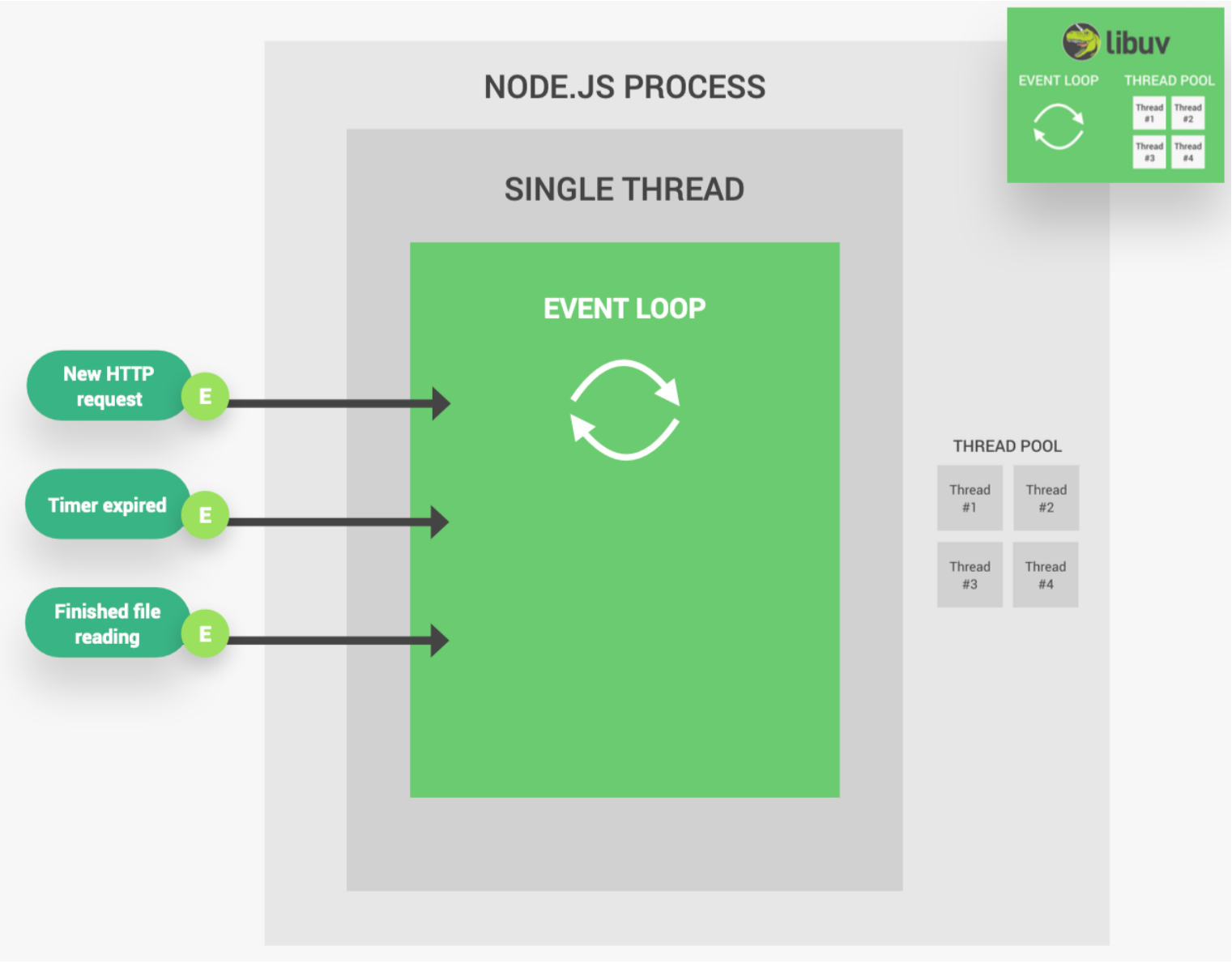


Node.js Process, Threads and the Thread Pool

- Node.js runs in a single thread
- A thread is basically a sequence of instructions
- Heavy tasks will block the single thread => use Thread Pool

Thread Pool

- Additional 4 threads (or more)
- Offload work from the event loop
- Handle heavy (“expensive”) tasks:
  - File system APIs
  - Cryptography
  - Compression
  - DNS lookups

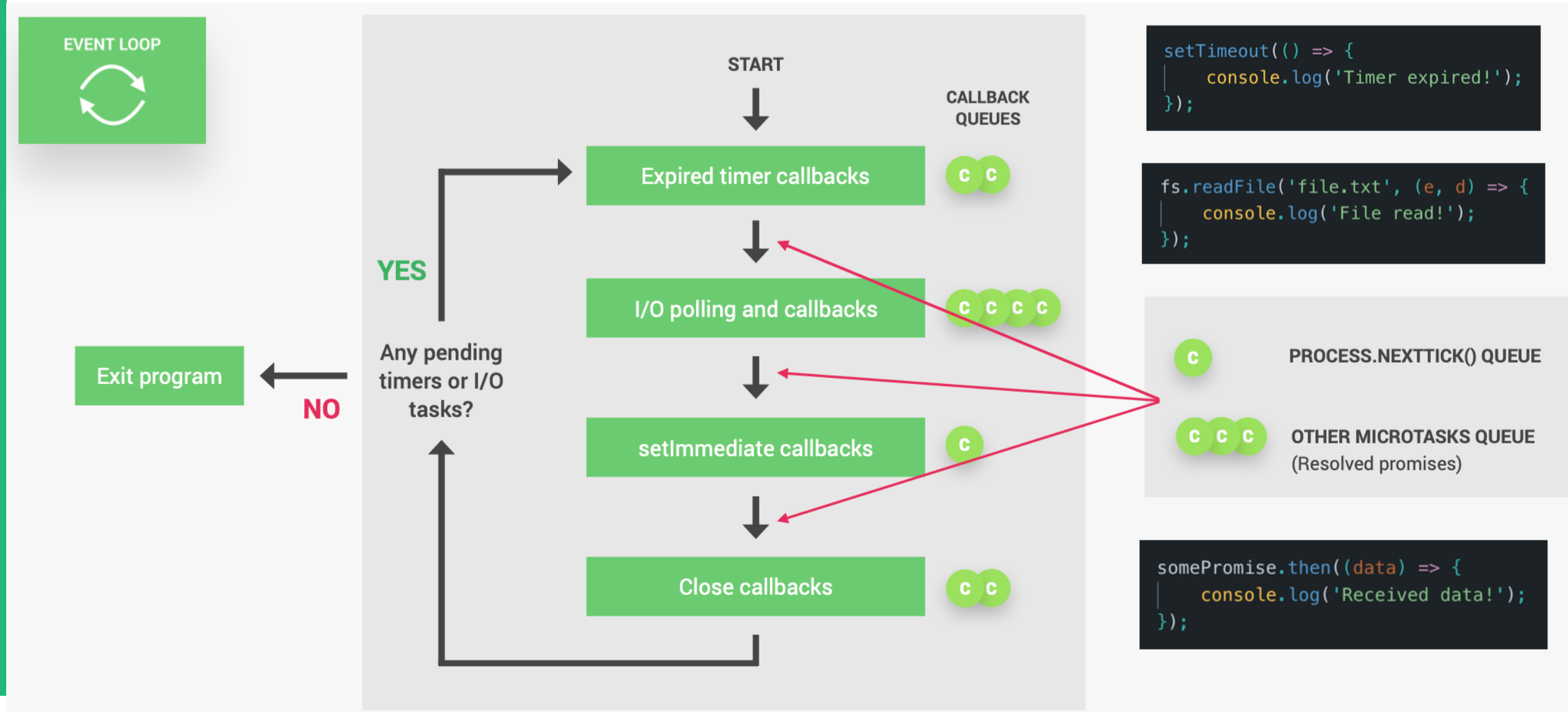


The Event Loop

The Event Loop

- The event loop is what allows Node.js to perform non-blocking I/O operations – even though JavaScript is single-threaded – by offloading operations to the thread pool whenever possible
- Event Loop is where all the application code that is inside callback functions (not top-level code) is executed
- Node.js is build around callback functions (some work is finished some time in future)
- Node.js is event-driven architecture:
  - Events are emitted (new http request, timer expired, finished file reading)
  - Event loops picks them up
  - Callbacks (associated with each event) are called
- Event loop does orchestration

The Event Loop in detail



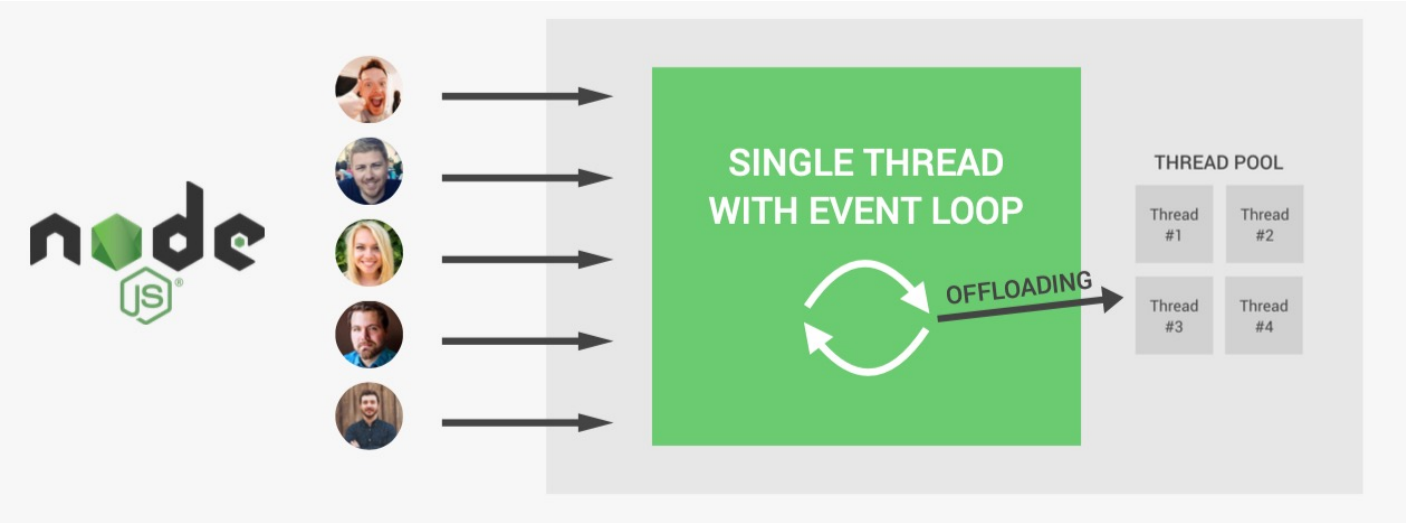
The Event Loop in detail

- 4 Phases of the Event loop:
  - Timers callbacks: scheduled by setTimeout() or setInterval() are executed
  - IO polling and callbacks: retrieves new I/O events and callbacks deferred to the next loop iteration are executed here
  - setImmediate callbacks: process callbacks immediately
  - Close callbacks: handles some close callbacks. Eg: socket.on('close', ...)
- Each phase has a callback queues

The Event Loop in detail

- Beside 4 callback queues, there are also 2 other queues:
  - process.nextTick() queue
  - Other microtask queue (for Resolved Promise)
- 2 queues will be executed right after the current phase finished instead of waiting for entire loop to finish
- A tick is basically just one cycle in this loop
- After finished a tick, node.js checks whether there are any timers or I/O tasks that are still running in the background. If there aren't any ➡️ exit the application. But if there are any pending timers or I/O tasks ➡️ continue running event loop (next tick)

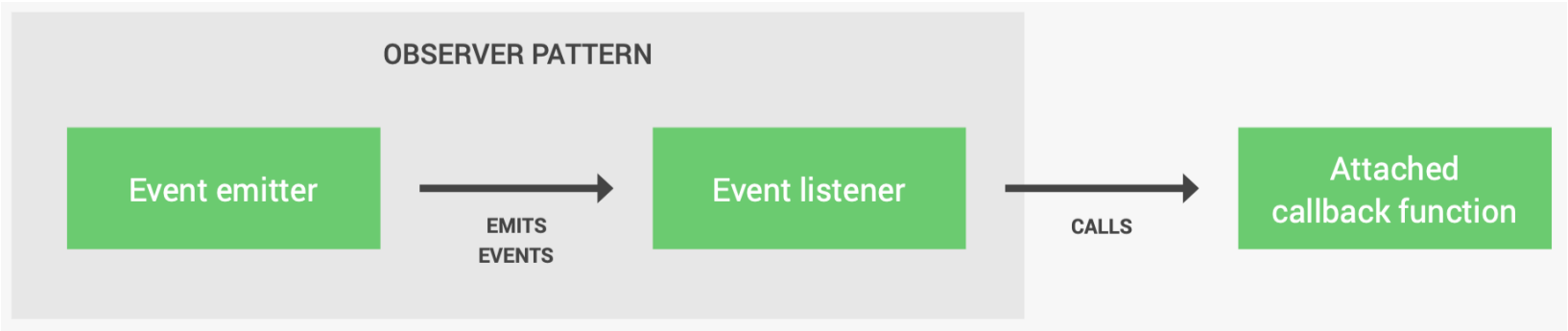
Summary of the Event Loop



- The event loop takes care of all incoming event and perform orchestration by offloading heavier task into the thread pool, and doing more simple work itself
- Don't block the event loop:
  - Don't use `sync` versions of functions in `fs`, `crypto` and `zlib` modules in your callback functions
  - Don't perform complex calculations (e.g. loops inside loops)
  - Be careful with JSON in large objects
  - Don't use too complex regular expressions (e.g. nested quantifiers)

Events and Event-Driven architecture

Events and Event-Driven architecture



- Event emitter: emit events as soon as something happens in the app, like a request hitting server, a timer expiring, or a file finishing to read
- Event listener: picks up event and fire off callback functions that are attached to each listener
- It is called the Observer Pattern in JavaScript

Streams

Introduction to Streams

- Streams: used to process (read and write) data piece by piece (chunks), without completing the whole read or write operation, and therefore without keeping all the data in memory
- Streams are instances of the EventEmitter class
- Pros:
  - Perfect for handling large volumes of data, for example videos
  - More efficient data processing in terms of memory (no need to keep all data in memory) and time (we don't have to wait until all the data is available)

Streams

	DESCRIPTION	EXAMPLE	IMPORTANT EVENTS	IMPORTANT FUNCTIONS
READABLE STREAMS	Streams from which we can read (consume) data	• `http` requests • `fs` read streams	• `data` • `end`	• `pipe()` • `read()`
WRITABLE STREAMS	Streams to which we can write data	• `http` responses • `fs` write streams	• `drain` • `finish`	• `write()` • `end()`
DUPLEX STREAMS	Streams that are both readable and writable	web socket		
TRANSFORM STREAMS	Duplex streams that transform data as it is written or read	`zlib` Gzip creation		

Modules

The CommonJS module system

- Each JavaScript file is treated as a separate module
- Node.js uses the CommonJS module system: `require()`, `exports` or `module.exports`

```
require('http');
```

Node.js Core Modules

Module	Description
http	Launch a server, send requests
https	Launch an SSL server
fs	Work with the file system
path	Handle paths elegantly
os	Get OS information

- Node.js ships with multiple core modules (http, fs, path, ...)
- Core modules can be imported into any file to be used there
- Import via `require('module')`

Resolving and loading a module

PATH RESOLVING: How Node decides which module to load

1. Start with core modules
2. If begins with ‘./‘ or ‘../‘ 👉 Try to load developer module
3. If no file found 👉 Try to find folder with `index.js` in it
4. Else 👉 Go to `node_modules/` and try to find module there

👉 **Core modules**

```
require('http');
```

👉 **Developer modules**

```
require('./lib/controller');
```

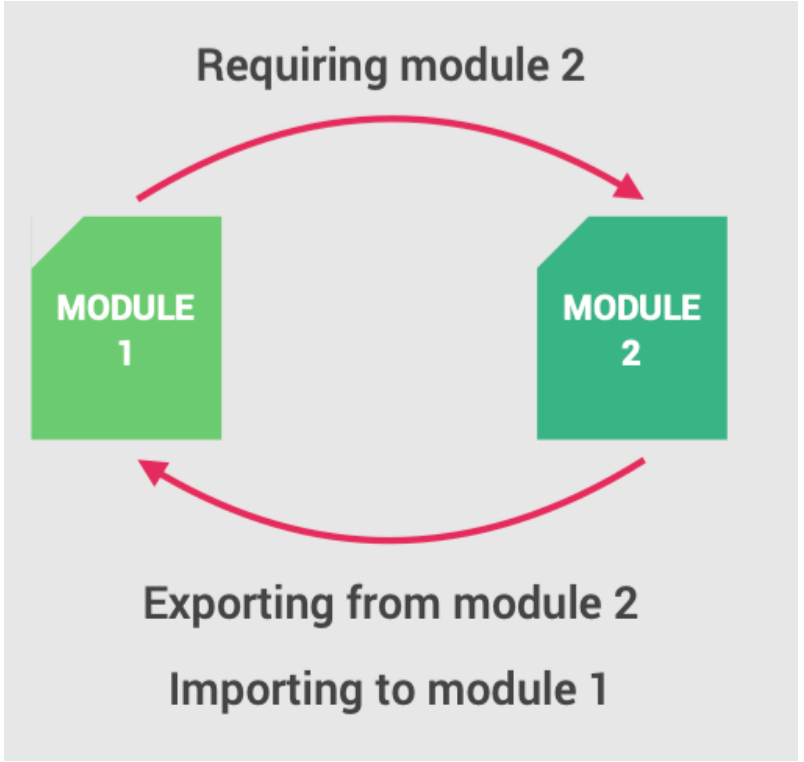
👉 **3rd-party modules (from NPM)**

```
require('express');
```

Export a module

- `require` function returns exports of the required module
- `module.exports` is the returned object (important!)
- Use `module.exports` to export one single variable, e.g. one class or one function (`module.exports = Calculator`)
- Use `exports` to export multiple named variables

```
(exports.add = (a, b) => a + b)
```



NPM

npm

- npm stands for “Node Package Manager” and it allows you to manage your Node project and its dependencies
- You can initialize a project with `npm init`
- npm scripts can be defined in the `package.json` to give you “shortcuts” to common tasks/commands
- You can install third-party packages via npm

Express.js

What is Express.js and Why use it?

- Express is a minimal node.js framework, a higher level of abstraction
- Express contains a very robust set of features: complex routing, easier handling of requests and responses, middleware, server-side rendering,...
- Express allows for rapid development of node.js applications: *we don't have to re-invent the wheel*
- Express makes it easier to organize our application into the MVC architecture

Installing Postman

- Postman is an API platform for building and using APIs
- Download at: https://www.postman.com/downloads/

Setting up Express and Basic Routing

- Install via npm: `npm i express`
- After installed:

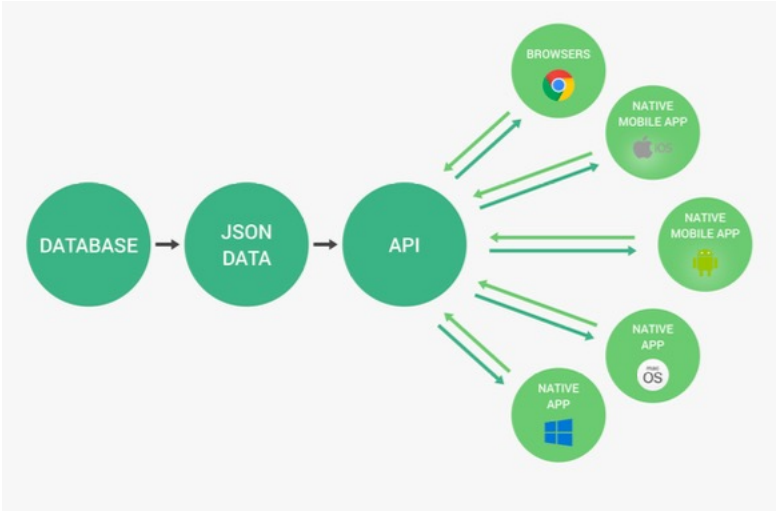
```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
    res.send('Hello World!')
});
const port = 3000;
app.listen(port, () => {
    console.log(`Example app listening on port ${port}!`)
});
```

Routing

- You can filter requests by path and method
- If you filter by method, paths are matched exactly, otherwise, the first segment of a URL is matched
- You can use the express.Router to split your routes across files elegantly
- Route structure: `app.METHOD(PATH, HANDLER)`
  - app is an instance of express
  - METHOD is an HTTP request method, in lowercase
  - PATH is a path on the server
  - HANDLER is the function executed when the route is matched

APIS

- **A**pplication **P**rogramming **I**nterface: a piece of software that can be used by another piece of software, in order to allow applications to talk to each other
- "Application" can be other things:
  - Node.js' `fs` or `http` APIs ("node APIs")
  - Browser's DOM JavaScript API
  - With object-oriented programming, when exposing methods to the public, we're creating an API
  - ....



The REST API Architecture

1. Separate API into logical resources
2. Expose structured, resource-based URLs
3. Use HTTP methods (verbs)
4. Send data as JSON (usually)
5. Be stateless

Resource-based URLs

- Resource: Object or representation of something, which has data associated to it. Any information that can be named can be a resource: `books`, `users`, `views`, `items`
- URL: `https://www.library.com/books`
- End point: `books`

HTTP Method (Verbs)

- GET: get a Resource from the Server
- POST: post a Resource to the Server (i.e. create or append Resource)
- PUT: put a Resource onto the Server (i.e. create or overwrite a Resource)
- PATCH: update parts of an existing Resource on the Server
- DELETE: delete a Resource on the Server

HTTP Method (Verbs)

The endpoints should contain only resources (nouns) and use HTTP method for action!

/addUser

/getUser/:id

/updateUser/:id

/deleteUser/:id

CRUD OPERATIONS →

POST /users

GET /users/:id

PUT/PATCH /users/:id

DELETE /users/:id

JSON


String
Value
Key-value
{
"id": 5,
"tourName": "The Park Camper",
"rating": "4.9",
"guides": [
{
"name": "Steven Miller",
"role": "Lead Guide"
},
{
"name": "Lisa Brown",
"role": "Tour Guide"
}
]
}
Object
Array

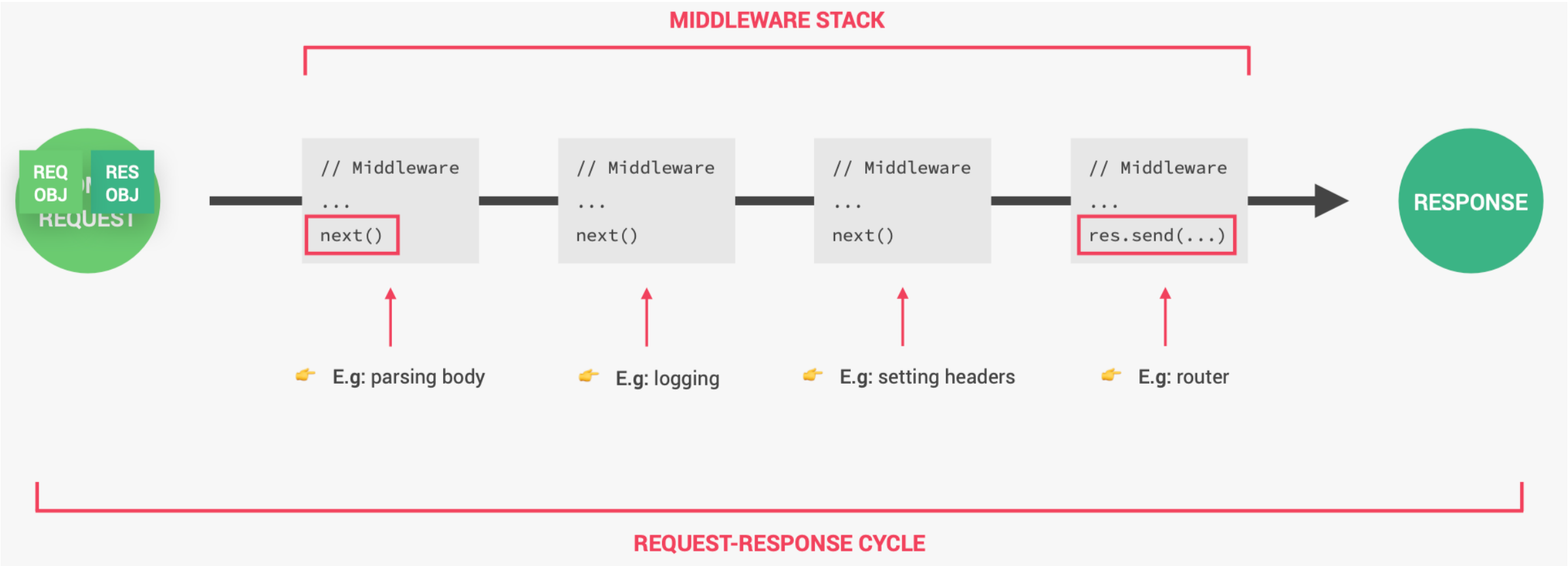
The REST API Architecture

- **CRUD Operation**: Create, Read, Update, Delete
- **Stateless RESTful API**: All state is handled on the client. This means that each request must contain all the information necessary to process a certain request. The server should not have to remember previous requests

Ex: currentPage = 5

Client		**Server**
GET /tours/nextPage	→ BAD	nextPage = currentPage +1 send(nextPage)
GET /tours/page/6	→ OK	send(6)

Middleware and The Request - Response Cycle



Middleware and The Request – Response Cycle

- Express app receives an incoming request when someone hits a server

➡️ It will create a request and response object (OBJ)

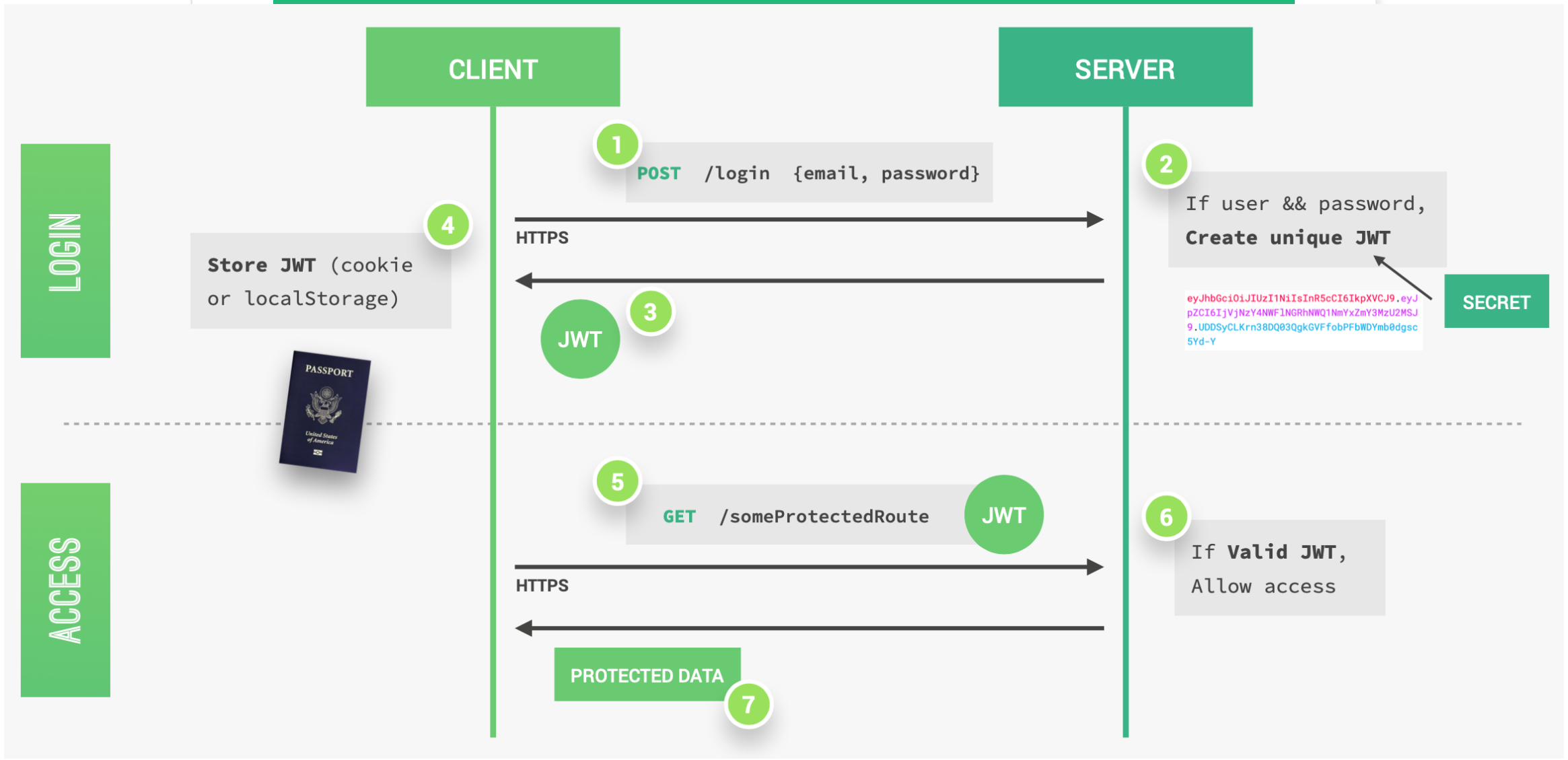
- That data will then be used and processed in order to generate and send back a meaningful response
- In order to process that data, Express use middleware, which can manipulate the request and response object
- In Express, everything is middleware (even routers)
- All the middleware together that we use in the app, is called the middleware stack

Middleware and The Request – Response Cycle

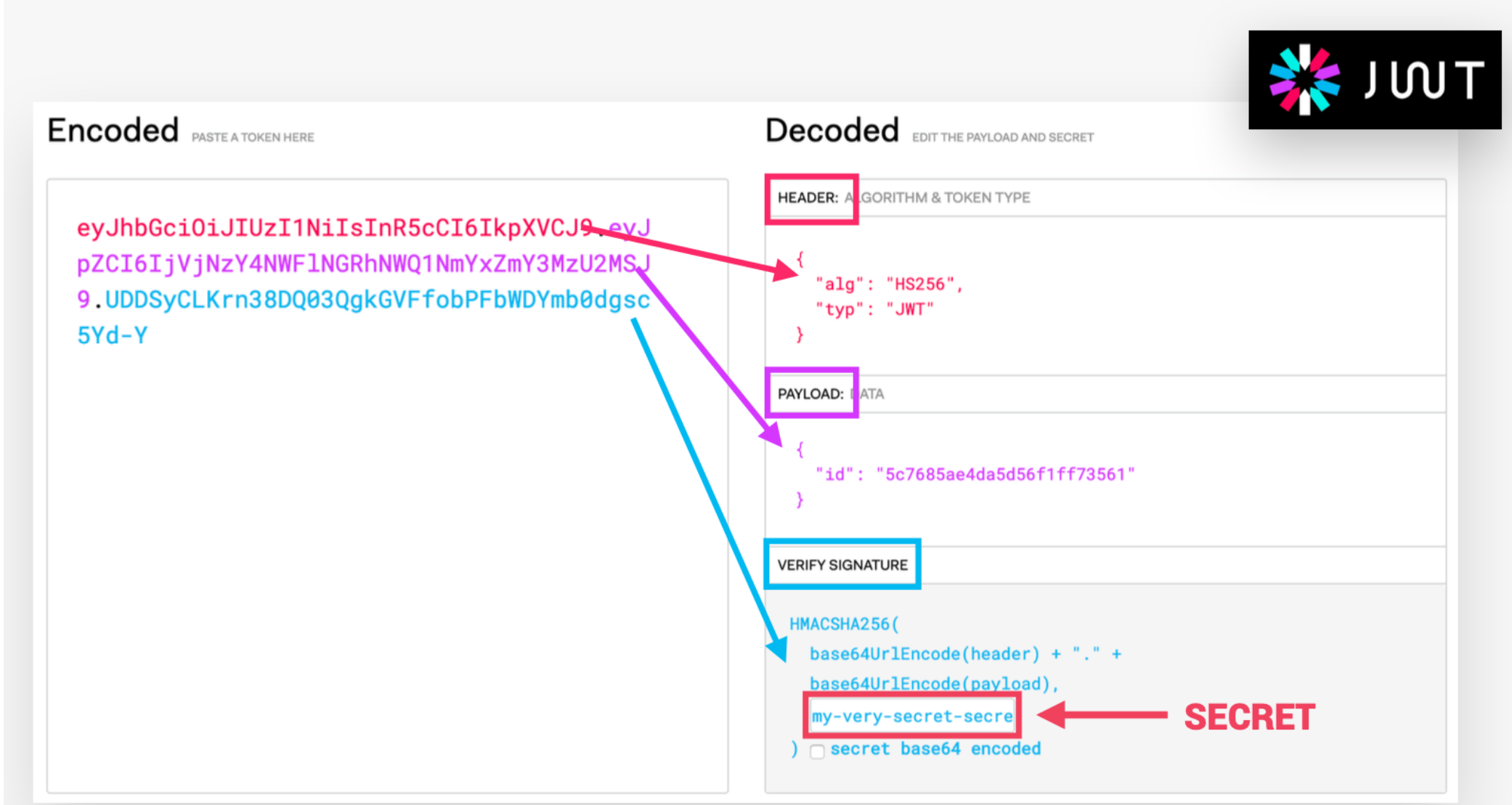
- The order of middleware in stack is defined by the order they are defined in the code
- The request and response object that were created in the beginning, go through each middleware where they are processed. Then at the end of each middleware function, `next()` function is called ➡️ the next middleware in the stack will be executed, until we reach the last one
- This whole process as kind of a pipeline where our data go through
- After last middleware function, we finally send the response data back to the client ➡️ finish the request – response cycle

JSON WEB TOKEN

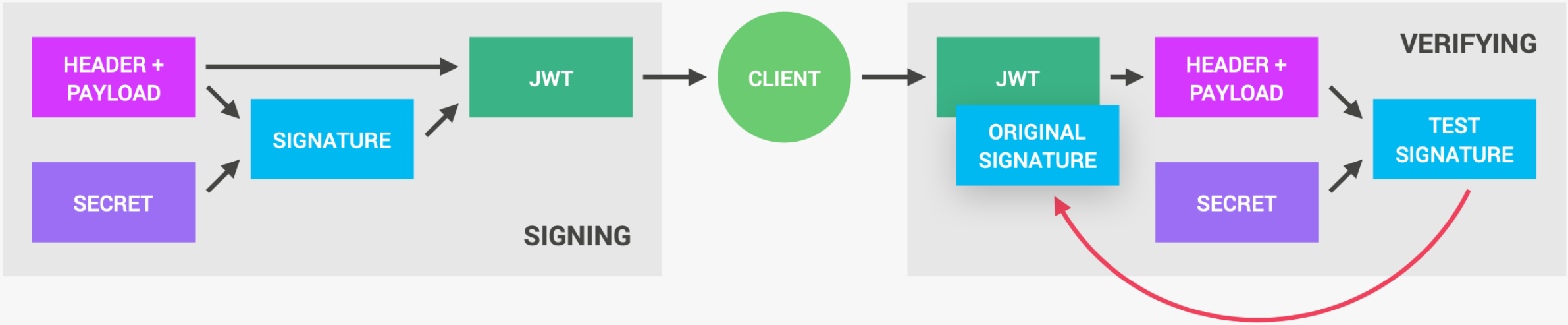
JSON WEB TOKEN - JWT



JSON WEB TOKEN - JWT



JSON WEB TOKEN - JWT



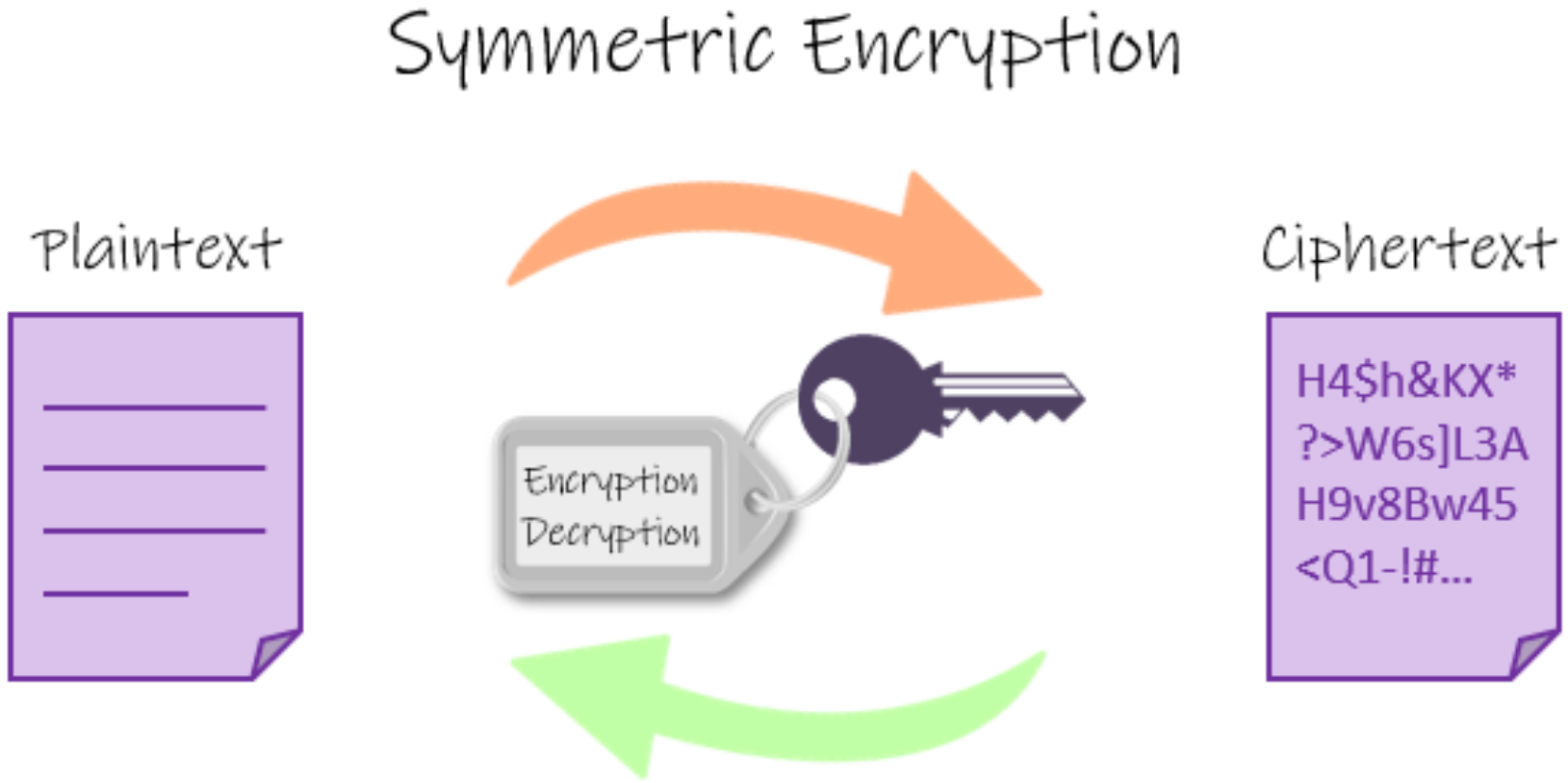
`test signature === signature` 👉 Data has not been modified 👉 **Authenticated**

`test signature !== signature` 👉 Data has been modified 👉 **Not authenticated**

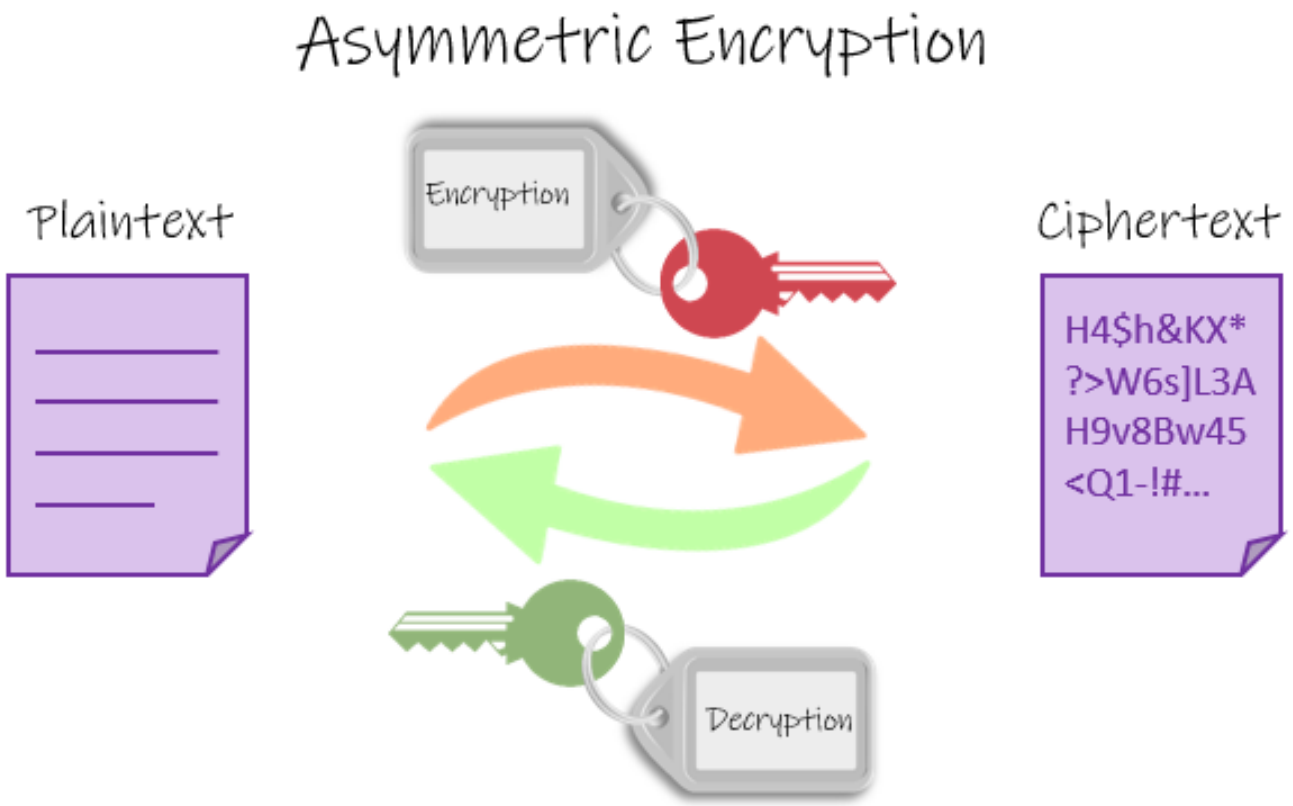
👉 **Without the secret, one will be able to manipulate the JWT data, because they cannot create a valid signature for the new data!**

Hashing algorithm

Symmetric encryption



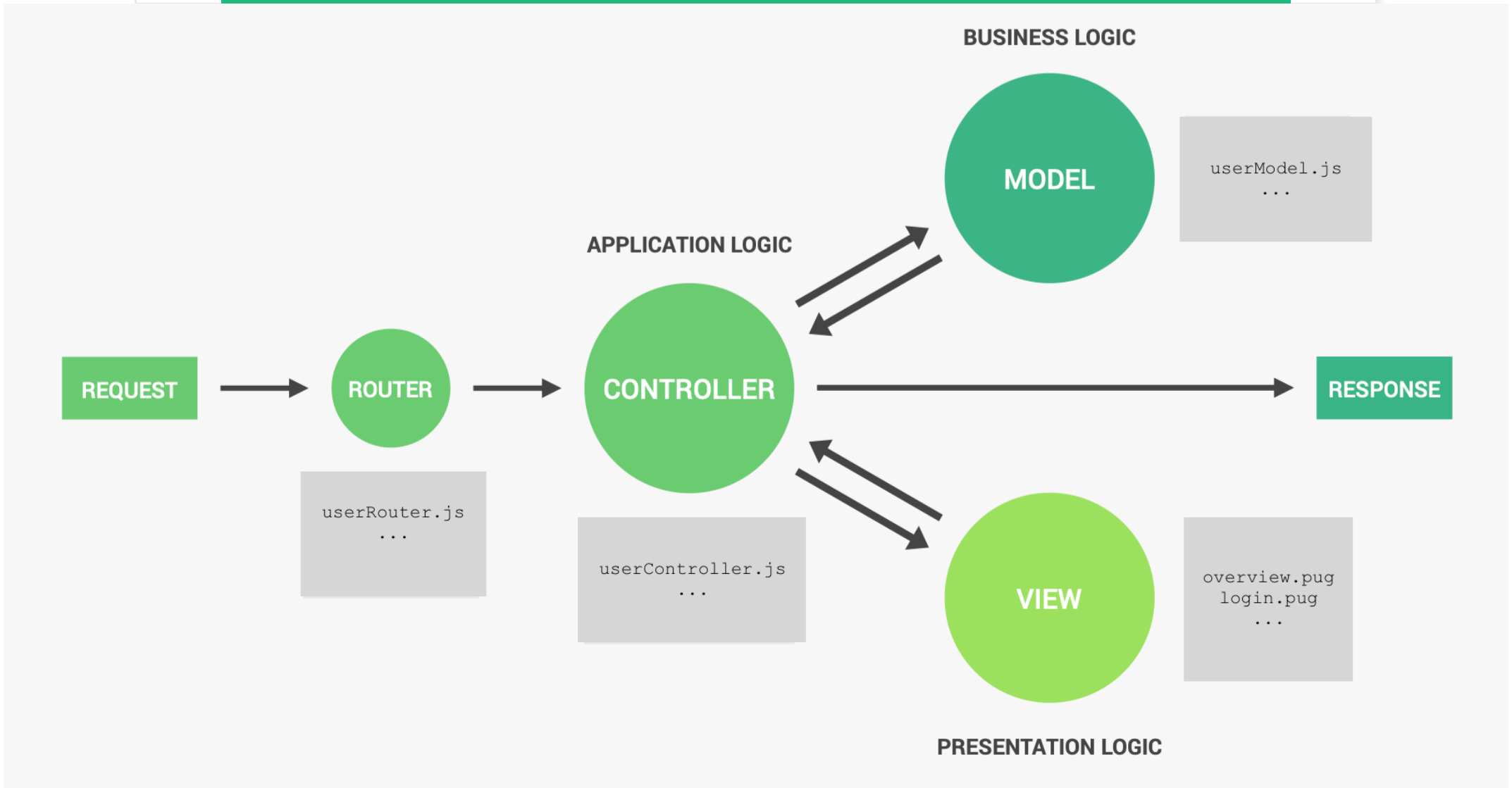
Asymmetric encryption



Serving static files

- You're not limited to serving dummy text as a response
- You can `sendFile()s` to your users – e.g. HTML files
- If a request is directly made for a file (e.g. a .css file is requested), you can enable static serving for such files via `express.static()`
- `express.static(root, [options])` where the `root` argument specifies the root directory from which to serve static assets

MVC Architecture in the Express App



MVC Architecture

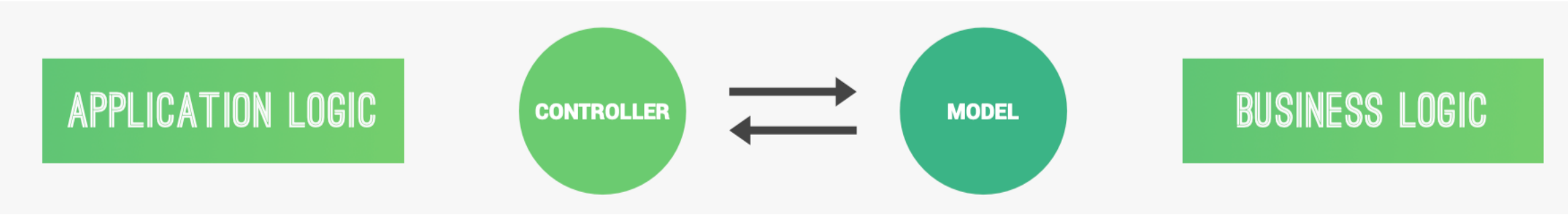
- A request hit one of routers, the router is delegating the request to the correct handler function in one controller
- The controller need to interact with one of the models and getting the data. After this, the controller might then be ready to send back a response to the client
- If we want to render a website (server-side rendering), the controller will then select one of view template and inject data into it. That render website will be sent back as the response

MVC Architecture

Models	Views	Controllers
• Represent your data in your code • Work with your data (e.g. save, fetch) • Doesn't matter if you manage data in memory, files, databases • Contains data-related logic	• What the users sees • Decoupled from your application code	• Connecting your Models and your Views • Contains the "in-between" logic

Application vs Business Logic

Fat models/thin controllers: offload as much logic as possible into the models and keep the controllers as simple and lean as possible



- Code that is only concerned about the application's implementation, not the underlying business problem we're trying to solve (e.g. showing and selling items);
- Concerned about managing requests and responses
- About the app's more technical aspects
- Bridge between model and view layers

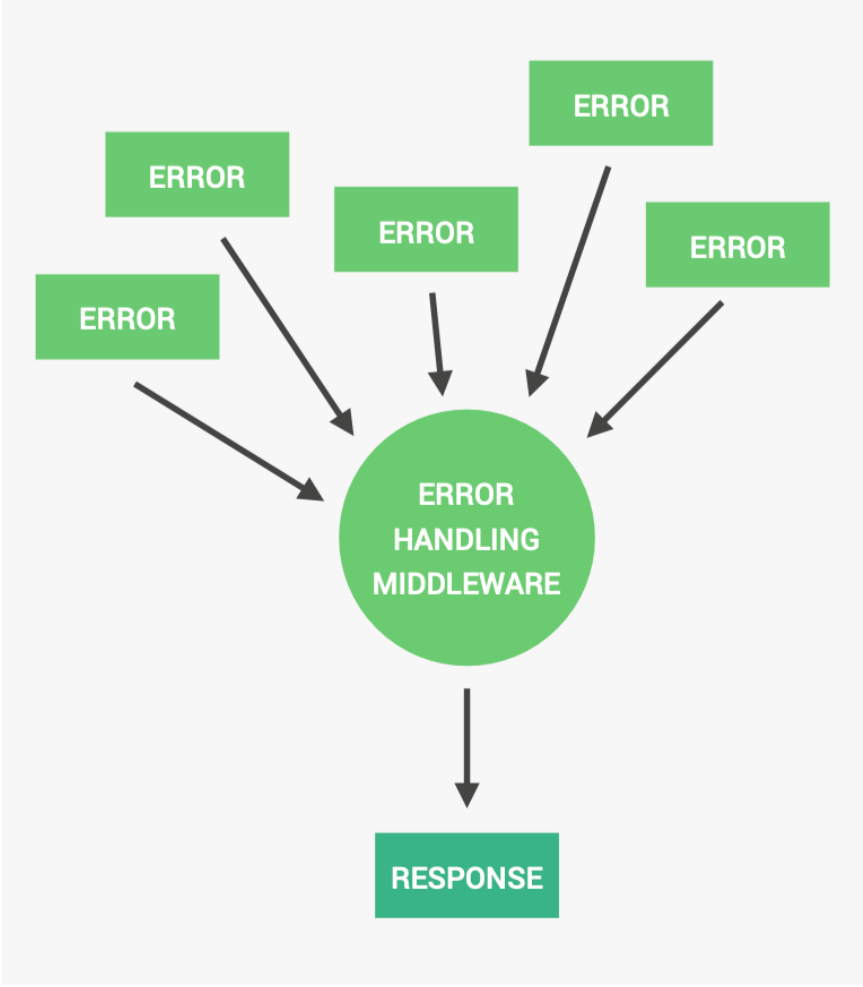
- Code that solves the business problem we set out to solve
- Directly related to business rules, how the business works, and business needs
- Examples:
  - Creating new item in the database
  - Checking if user's password is correct
  - Validating user input data
  - Ensuring only users who bought an item can review it

Error Handling with Express

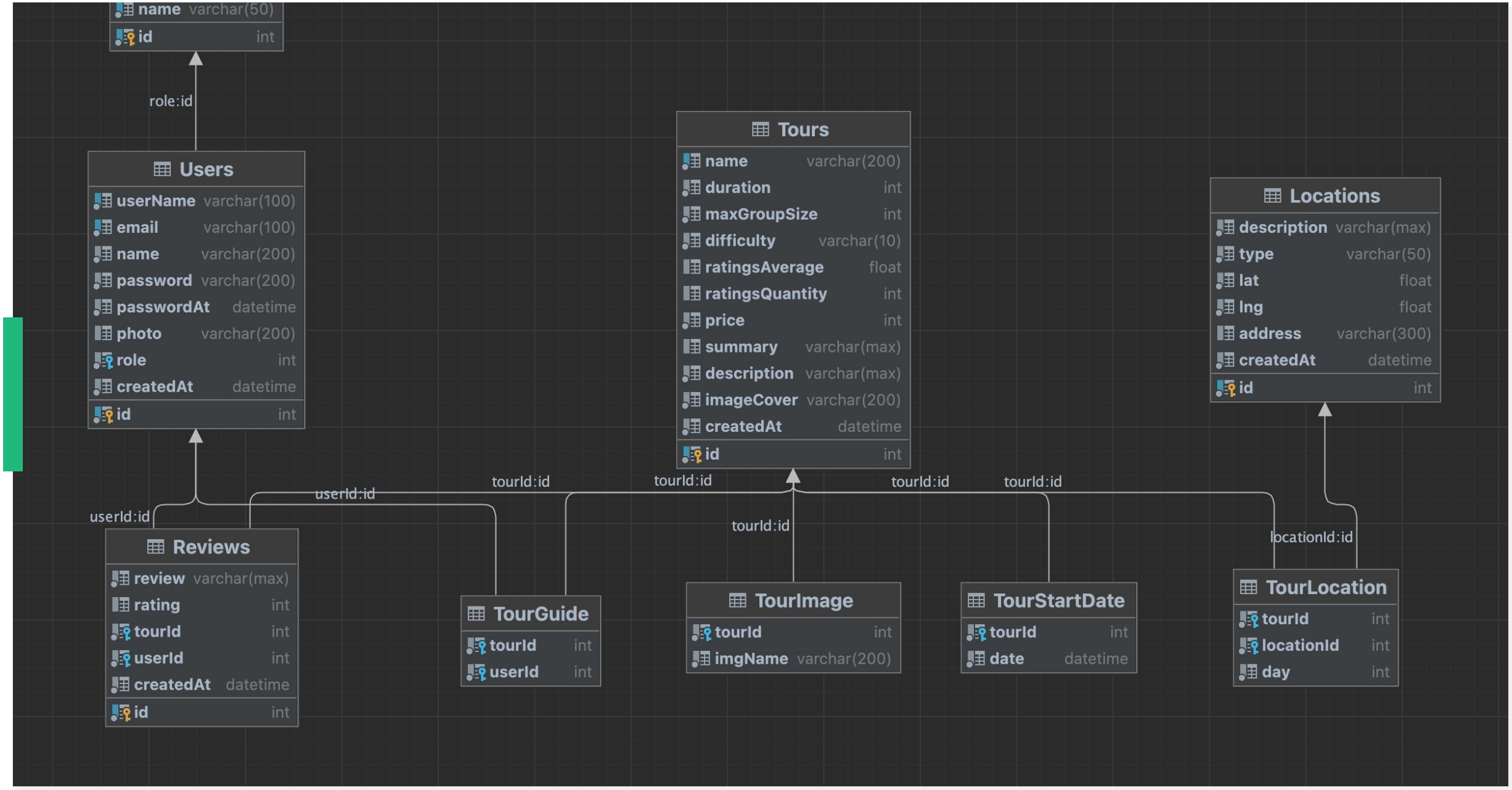
Operational Errors	Programing Errors
Problems that we can predict will happen at some point, so we just need to handle them in advance • Invalid path accessed • Invalid user input • Failed to connect to server • Failed to connect to database • Request timeout • Etc...	Bugs that we developers introduce into our code. Difficult to find and handle • Reading properties on `undefined` • Passing a number where an object is expected • Using `await` without `async` • Using `req.query` instead of `req.body` • Etc...

Catch and handle Operational Errors with Express

- Write a global express error handling middleware with will catch errors coming from all over the application
- We can send nice response back to the client letting the user know what happened
- In other cases, handling can also mean retrying the operation or crashing the server or just ignoring the error




THE END


name varchar(50)
id int
role:id
Users
userName varchar(100)
email varchar(100)
name varchar(200)
password varchar(200)
passwordAt datetime
photo varchar(200)
role int
createdAt datetime
id int
Tours
name varchar(200)
duration int
maxGroupSize int
difficulty varchar(10)
ratingsAverage float
ratingsQuantity int
price int
summary varchar(max)
description varchar(max)
imageCover varchar(200)
createdAt datetime
id int
Locations
description varchar(max)
type varchar(50)
lat float
lng float
address varchar(300)
createdAt datetime
id int
userId:id
userId:id
tourId:id
tourId:id
tourId:id
tourId:id
tourId:id
locationId:id
Reviews
review varchar(max)
rating int
tourId int
userId int
createdAt datetime
id int
TourGuide
tourId int
userId int
TourImage
tourId int
imgName varchar(200)
TourStartDate
tourId int
date datetime
TourLocation
tourId int
locationId int
day int

Query with filter + pagination + sorting

```
const TourSchema = new ModelSchema(
    schema: {
        id: new ModelSchemaValidator( config: {
            name: 'id',
            sqlType: sql.Int,
        }),
        name: new ModelSchemaValidator( config: {
            name: 'name',
            sqlType: sql.VarChar,
            require: true,
        }),
        duration: new ModelSchemaValidator( config: {
            name: 'duration',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
        price: new ModelSchemaValidator( config: {
            name: 'price',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val >= 0;
            },
        }),
        difficulty: new ModelSchemaValidator( config: {
            name: 'difficulty',
            sqlType: sql.VarChar,
            require: true,
            validator:  function (val) {
                const diffArr = ["easy", "medium", "difficult"];
                return (diffArr.indexOf(val) > -1);
            },
        }),
        maxGroupSize: new ModelSchemaValidator( config: {
            name: 'maxGroupSize',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
```

```
{
  duration: { gte: '5', lt: '8' },
  price: { lt: '500' },
  difficulty: 'easy',
  pageSize: '10',
  sort: 'duration,-maxGroupSize',
  page: '1'
}
```

```
for (let criteria in filter){
    if (schema[criteria]){
        const schemaProp = schema[criteria];
        if (i > 0) {
            filterStr += ' AND '
        }else{
            filterStr += 'WHERE '
        }

        if (schemaProp.type === 'number'){
            if (typeof filter[criteria] === 'object'){
                let j = 0;
                for (let criteriaOperator in filter[criteria]){
                    let operator;
                    let criterialVal;
                    if (criteriaOperator === 'gt'){
                        operator = '>'
                        criterialVal = filter[criteria]['gt'];
                    }else if (criteriaOperator === 'gte'){
                        operator = '>='
                        criterialVal = filter[criteria]['gte'];
                    }else if (criteriaOperator === 'lt'){
                        operator = '<'
                        criterialVal = filter[criteria]['lt'];
                    }else if (criteriaOperator === 'lte'){
                        operator = '<='
                        criterialVal = filter[criteria]['lte'];
                    }else if (criteriaOperator === 'eq'){
                        operator = '='
                        criterialVal = filter[criteria]['eq'];
                    }
                    if (operator && criterialVal){
                        if (j > 0) {
                            filterStr += ' AND '
                        }
                        filterStr += criteria + ' ' + operator + ' ' + criterialVal;
                        i++;
                        j++;
                    }
                }
            }else{
                filterStr += criteria + ' = ' + filter[criteria];
                i++;
            }
        }else if (schemaProp.type === 'string'){
            filterStr += criteria + " = '" + filter[criteria] + "'";
            i++;
        }
    }
}
```

```
{
    duration: { gte: '5', lt: '8' },
    price: { lt: '500' },
    difficulty: 'easy',
    pageSize: '10',
    sort: 'duration,-maxGroupSize',
    page: '1'
}
```

```
const TourSchema = new ModelSchema(
    schema: {
        id: new ModelSchemaValidator( config: {
            name: 'id',
            sqlType: sql.Int,
        }),
        name: new ModelSchemaValidator( config: {
            name: 'name',
            sqlType: sql.VarChar,
            require: true,
        }),
        duration: new ModelSchemaValidator( config: {
            name: 'duration',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
        price: new ModelSchemaValidator( config: {
            name: 'price',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val >= 0;
            },
        }),
        difficulty: new ModelSchemaValidator( config: {
            name: 'difficulty',
            sqlType: sql.VarChar,
            require: true,
            validator:  function (val) {
                const diffArr = ["easy", "medium", "difficult"];
                return (diffArr.indexOf(val) > -1);
            },
        }),
        maxGroupSize: new ModelSchemaValidator( config: {
            name: 'maxGroupSize',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
```

```
if (sort){
    let sortCriterias = sort.split(',')
    if (sortCriterias.length > 0){
        // console.log(sortCriterias);
        sortCriterias.forEach(criteria => {
            let sortDirection = 'asc';
            let sortProp = criteria;
            if (criteria.startsWith('-')){
                sortDirection = 'desc';
                sortProp = criteria.replace(/^-+/, '')
            }

            if (schema[sortProp]) {
                sortStr += sortProp + ' ' + sortDirection + ',';
            }
        })
    }
}

if (sortStr){
    sortStr = sortStr.slice(0, -1); //delete last ','
}else{
    sortStr = defaultSortStr;
}
```

```
//offset 0 ROWS FETCH NEXT 10 ROWS ONLY;
paginationStr += ' ' + sortStr + ' OFFSET ' + skip +
  ' ROWS FETCH NEXT ' + pageSize + ' ROWS ONLY'
```

```
{
  duration: { gte: '5', lt: '8' },
  price: { lt: '500' },
  difficulty: 'easy',
  pageSize: '10',
  sort: 'duration,-maxGroupSize',
  page: '1'
}
```

Insert

```
let result = await request
    .input('name', sql.VarChar, tour.name)
    .input('duration', sql.Int, tour.duration)
    .input('maxGroupSize', sql.Int, tour.maxGroupSize)
    .input('difficulty', sql.VarChar, tour.difficulty)
    .input('ratingsAverage', sql.Float, tour.ratingsAverage)
    .input('ratingsQuantity', sql.Int, tour.ratingsQuantity)
    .input('price', sql.Int, tour.price)
    .input('summary', sql.VarChar, tour.summary)
    .input('description', sql.VarChar, tour.description)
    .input('imageCover', sql.VarChar, tour.imageCover)
    .query(
        'insert into Tours ' +
        '(name, duration, maxGroupSize, difficulty, ratingsAverage, ratingsQuantity, price, summary, description, imageCover)' +
        'values (@name, @duration, @maxGroupSize, @difficulty, @ratingsAverage, @ratingsQuantity ,@price, @summary, @description, @imageCover)'
    );
```

```
let result = await dbConfig.db.pool
    .request()
    .input('tourId', sql.Int, tourId)
    .input('imgName', sql.VarChar, imgName)
    .query(
        'insert into TourImage ' +
        '(tourId, imgName)' +
        ' values (@tourId, @imgName)'
    );
```

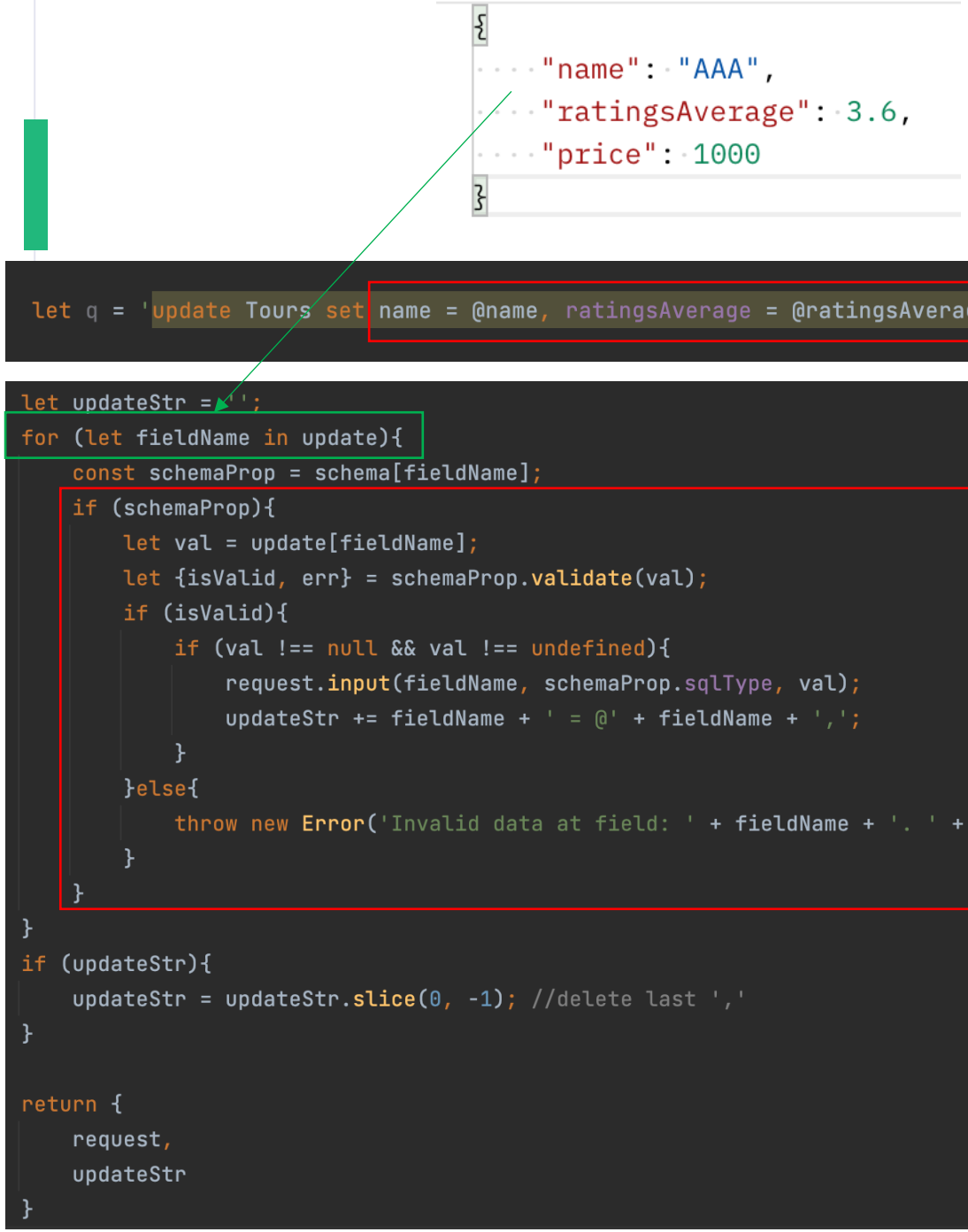
```
let result = await dbConfig.db.pool
    .request()
    .input('tourId', sql.Int, tourId)
    .input('date', sql.DateTime, date)
    .query(
        'insert into TourStartDate ' +
        '(tourId, date)' +
        ' values (@tourId, @date)'
    );
```

```
exports.getInsertQuery = (schema, request, insertData) => {
    if (!insertData){
        throw new Error('Invalid insertData param');
    }

    let insertFieldNamesStr = '';
    let insertValuesStr = '';

    for (let fieldName in schema){
        const schemaProp = schema[fieldName];
        let val = insertData[fieldName];
        let {isValid, err} = schemaProp.validate(val);
        if (isValid){
            if (val !== null && val !== undefined){
                request.input(fieldName, schemaProp.sqlType, val);
                insertFieldNamesStr += fieldName + ',';
                insertValuesStr += '@' + fieldName + ',';
            }
        }else{
            throw new Error('Invalid data at field: ' + fieldName + '. ' + err);
        }

    }

    if (insertFieldNamesStr && insertValuesStr){
        insertFieldNamesStr = insertFieldNamesStr.slice(0, -1); //delete last ','
        insertValuesStr = insertValuesStr.slice(0, -1); //delete last ','
    }


    return {
        request,
        insertFieldNamesStr,
        insertValuesStr,
    }
}
```


```
{
    "name": "The Snow Adventur
    "duration": 4,
    "maxGroupSize": 10,
    "difficulty": "difficult",
    "ratingsAverage": 4.5,
    "ratingsQuantity": 13,
    "price": 997,
    "summary": "Exciting adven
    "description": "Sed do eiu
        nostrud exercitation u
        culpa qui officia dese
        dolore eu fugiat nulla
        consectetur adipisicin
    "imageCover": "tour-3-cove
    "images": ["tour-3-1.jpg",
    "startDates": ["2022-01-05
}
```


```
const TourSchema = new ModelSchema(
    schema: {
        id: new ModelSchemaValidator( config: {
            name: 'id',
            sqlType: sql.Int,
        }),
        name: new ModelSchemaValidator( config: {
            name: 'name',
            sqlType: sql.VarChar,
            require: true,
        }),
        duration: new ModelSchemaValidator( config: {
            name: 'duration',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
        maxGroupSize: new ModelSchemaValidator( config: {
            name: 'maxGroupSize',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
        difficulty: new ModelSchemaValidator( config: {
            name: 'difficulty',
            sqlType: sql.VarChar,
            require: true,
            validator:  function (val) {
                const diffArr = ["easy", "medium", "difficult"];
                return (diffArr.indexOf(val) > -1);
            },
        }),
        ratingsAverage: new ModelSchemaValidator( config: {
            name: 'ratingsAverage',
            sqlType: sql.Float,
            default: 4.5,
            validator:  function (val) {
                return val >= 0 && val <= 5;
            },
        }),
        ratingsQuantity: new ModelSchemaValidator( config: {
```

Update



```
{
    "name": "AAA",
    "ratingsAverage": 3.6,
    "price": 1000
}
```

```
let q = 'update Tours set name = @name, ratingsAverage = @ratingsAverage, price = @price where id = @id';
```

```
let updateStr ='';
for (let fieldName in update){
    const schemaProp = schema[fieldName];
    if (schemaProp){
        let val = update[fieldName];
        let {isValid, err} = schemaProp.validate(val);
        if (isValid){
            if (val !== null && val !== undefined){
                request.input(fieldName, schemaProp.sqlType, val);
                updateStr += fieldName + ' = @' + fieldName + ',';
            }
        }else{
            throw new Error('Invalid data at field: ' + fieldName + '. ' + err);
        }
    }
}
if (updateStr){
    updateStr = updateStr.slice(0, -1); //delete last ','
}

return {
    request,
    updateStr
}
```

```
const TourSchema = new ModelSchema(
    schema: {
        id: new ModelSchemaValidator( config: {
            name: 'id',
            sqlType: sql.Int,
        }),
        name: new ModelSchemaValidator( config: {
            name: 'name',
            sqlType: sql.VarChar,
            require: true,
        }),
        duration: new ModelSchemaValidator( config: {
            name: 'duration',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
        maxGroupSize: new ModelSchemaValidator( config: {
            name: 'maxGroupSize',
            sqlType: sql.Int,
            require: true,
            validator:  function (val) {
                return val > 0;
            },
        }),
        difficulty: new ModelSchemaValidator( config: {
            name: 'difficulty',
            sqlType: sql.VarChar,
            require: true,
            validator:  function (val) {
                const diffArr = ["easy", "medium", "difficult"];
                return (diffArr.indexOf(val) > -1);
            },
        }),
        ratingsAverage: new ModelSchemaValidator( config: {
            name: 'ratingsAverage',
            sqlType: sql.Float,
            default: 4.5,
            validator:  function (val) {
                return val >= 0 && val <= 5;
            },
        }),
        ratingsQuantity: new ModelSchemaValidator( config: {
```