



AMERICAN UNIVERSITY
OF PHNOM PENH

STUDY LOCALLY. LIVE GLOBALLY.

INF 653 – Back-end Web Development

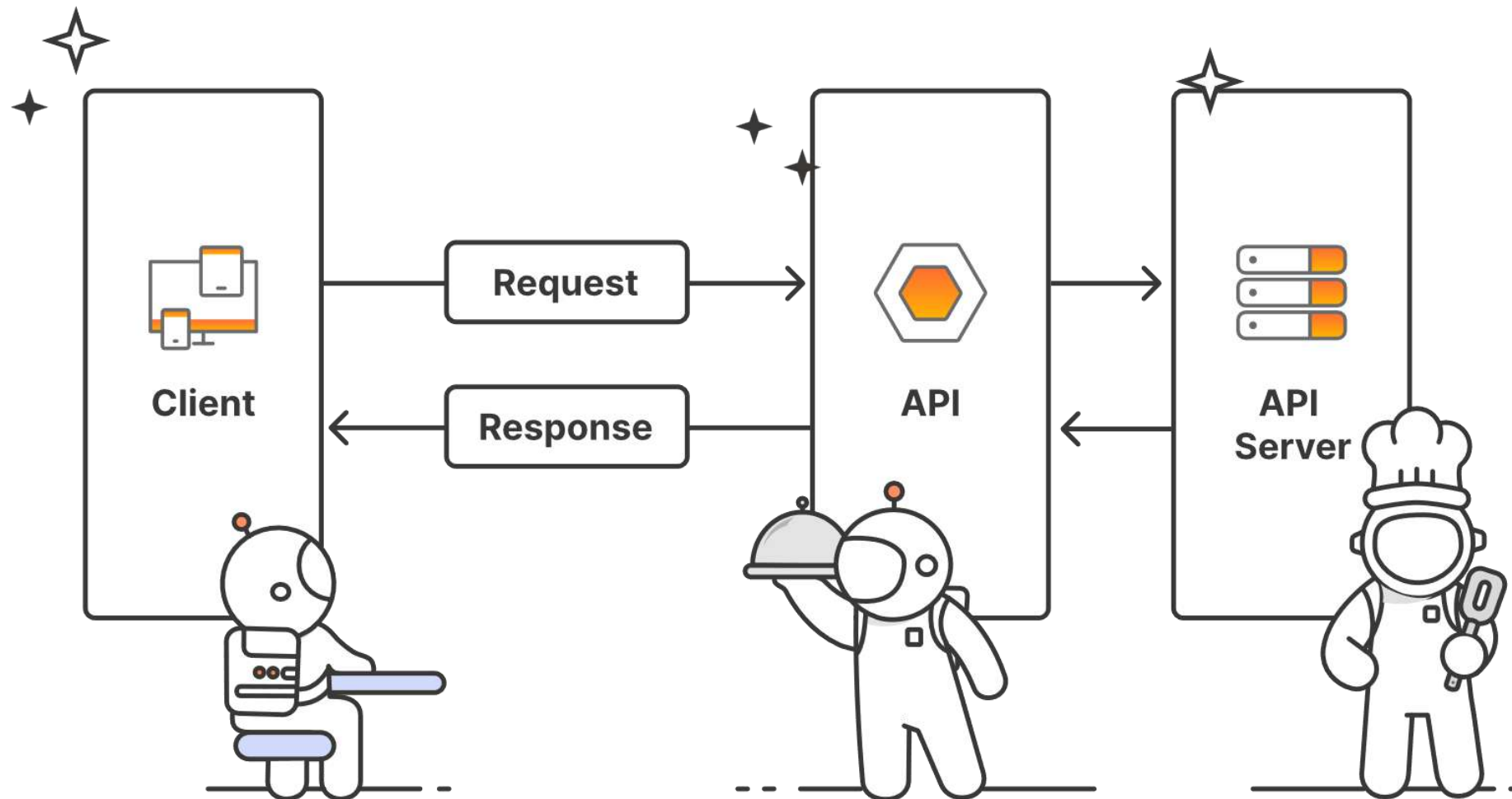
HANDLING HTTP REQUESTS AND FORMS WITH NODE.JS

Instructor: Monyrath Buntoun (Ms.)

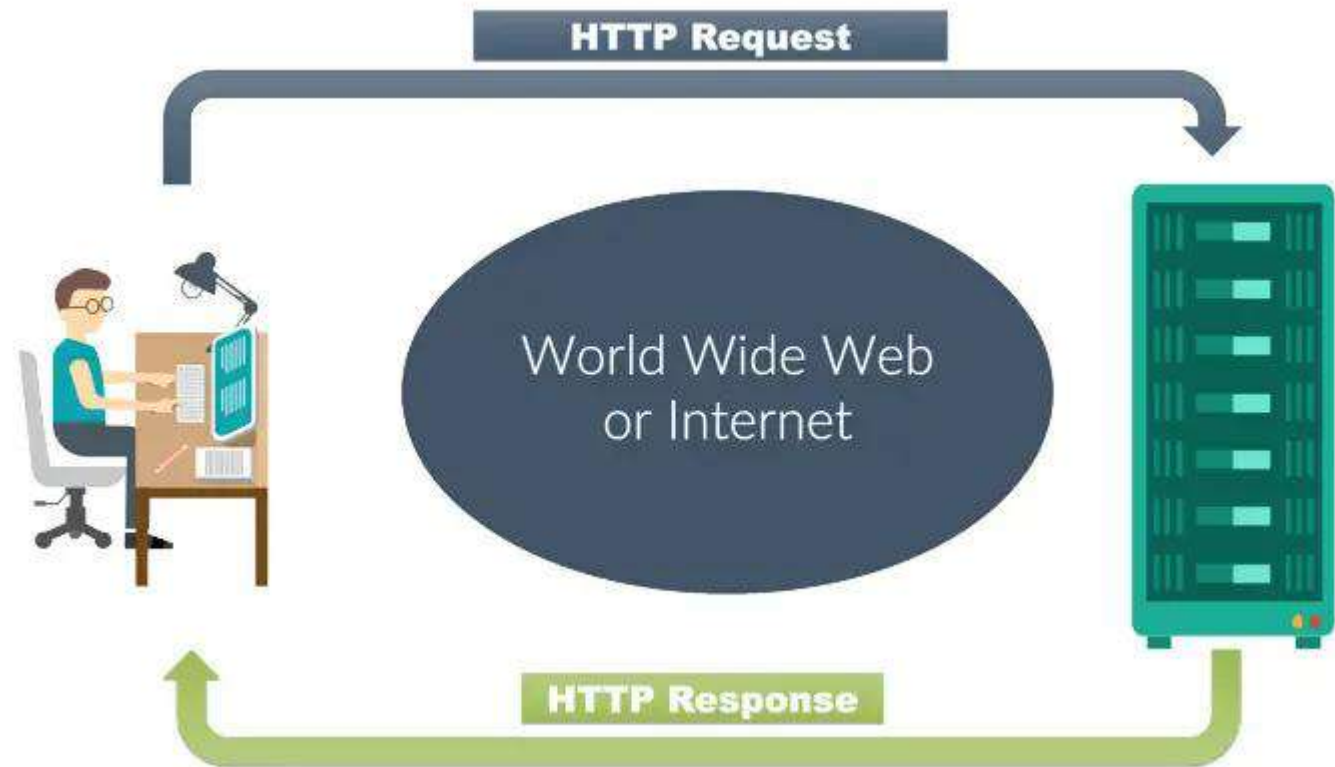
CONTENT

- I. Introduction
- II. Breaking down the URL
- III. HTTP Request Methods
- IV. Introduction to Express
- V. Request & Response Header
- VI. Request & Response Objects
- VII. Form Handling

INTRODUCTION



What are some examples of a client request?

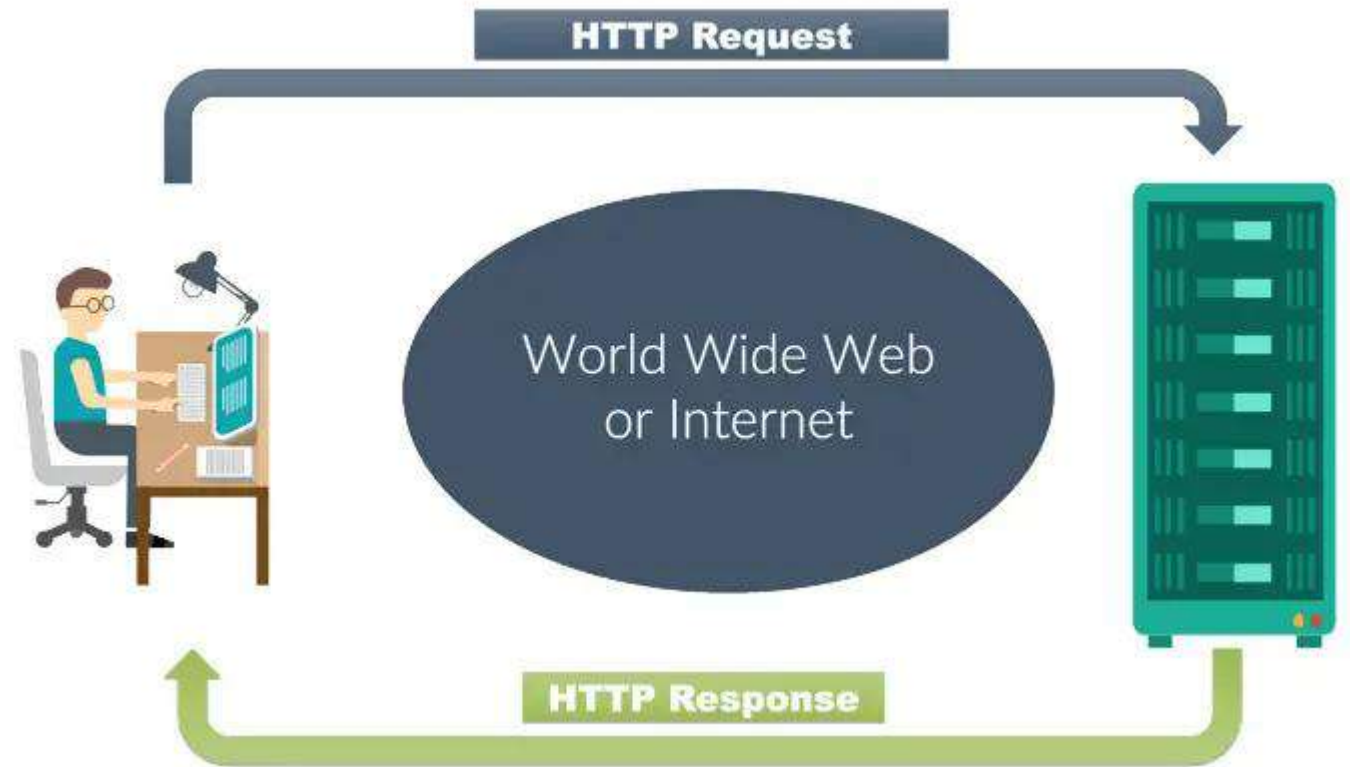


Img src: <https://www.ryadel.com/en/http-request-response-what-how-guide/>

INTRODUCTION

Client requests include:

- Simply visiting the websites
- Trying to access information
- Submitting forms
- Authentication
- Posting information, and more.



*On the **server side**, we will be focusing on **responding/handling the requests** made by clients.*

BREAKING DOWN THE URL

https://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express
<i>protocol</i>	<i>hostname</i>	<i>port</i>	<i>path</i>	<i>querystring</i>	<i>fragment</i>

A typical URL can be broken down to:

- **protocol, hostname, port, path, querystring, and fragment.**

Each has its own meaning, and purposes.

BREAKING DOWN THE URL

https://	google.com				#q=express
http://	www.bing.com		/search	?q=grunt&first=9	
http://	localhost	:3000	/about	?test=1	#history
<i>protocol</i>	<i>hostname</i>	<i>port</i>	<i>path</i>	<i>querystring</i>	<i>fragment</i>

Protocol – determines how the request will be transmitted.

- We will be focusing on **http** & **https** (secured) protocol which is for **web browsing**.
- Aside from http(s), there are also other protocols such as ftp(s) for file transfer, smtp(s) for email, and more.

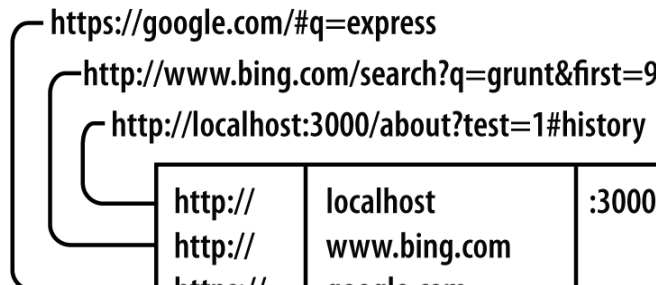
BREAKING DOWN THE URL

https://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express
<i>protocol</i>	<i>hostname</i>	<i>port</i>	<i>path</i>	<i>querystring</i>	<i>fragment</i>

hostname – identifies the server where the resource is hosted.

- localhost would be your local network.
- hostname could also be domain name like google.com, www.bing.com etc.
- We can identify the server hosts and the type of website based on domain name.

BREAKING DOWN THE URL



https://google.com/#q=express

http://www.bing.com/search?q=grunt&first=9

http://localhost:3000/about?test=1#history

http://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express
<i>protocol</i>	<i>hostname</i>	<i>port</i>	<i>path</i>	<i>querystring</i>	<i>fragment</i>

port – a single machine has a collection of numbered ports. The port number specifies the entry point for the server.

Eg. **:3000** means the server is hosted on port 3000.

- Some port numbers are special, like 80 (HTTP) and 443 (HTTPS).
- In general, if you aren't using port 80 or 443, you should use a port number greater than 1023.
- For **development**, easy-to-remember ports like **3000**, **8080** are commonly used.

Only one server can be associated with a given port.

BREAKING DOWN THE URL

https://google.com/#q=express

http://www.bing.com/search?q=grunt&first=9

http://localhost:3000/about?test=1#history

http://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express
<i>protocol</i>	<i>hostname</i>	<i>port</i>	<i>path</i>	<i>querystring</i>	<i>fragment</i>

path – the **first part of the URL that your web app actually cares about.**

- Should be used to uniquely identify pages or other resources in your app.

Eg. /search would mean accessing the “search” page or endpoint.

BREAKING DOWN THE URL


https://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express
<i>protocol</i>	<i>hostname</i>	<i>port</i>	<i>path</i>	<i>querystring</i>	<i>fragment</i>

Querystring – an **optional** collection of **name/value pairs** for sending data to the server.

- The querystring starts with a question mark (?), and name/value pairs are separated by ampersands (&).
- Both names and values should be URL encoded.

Eg. spaces will be replaced with plus signs (+). Other special characters will be replaced with numeric character references.

BREAKING DOWN THE URL



http://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express
<i>protocol</i>	<i>hostname</i>	<i>port</i>	<i>path</i>	<i>querystring</i>	<i>fragment</i>

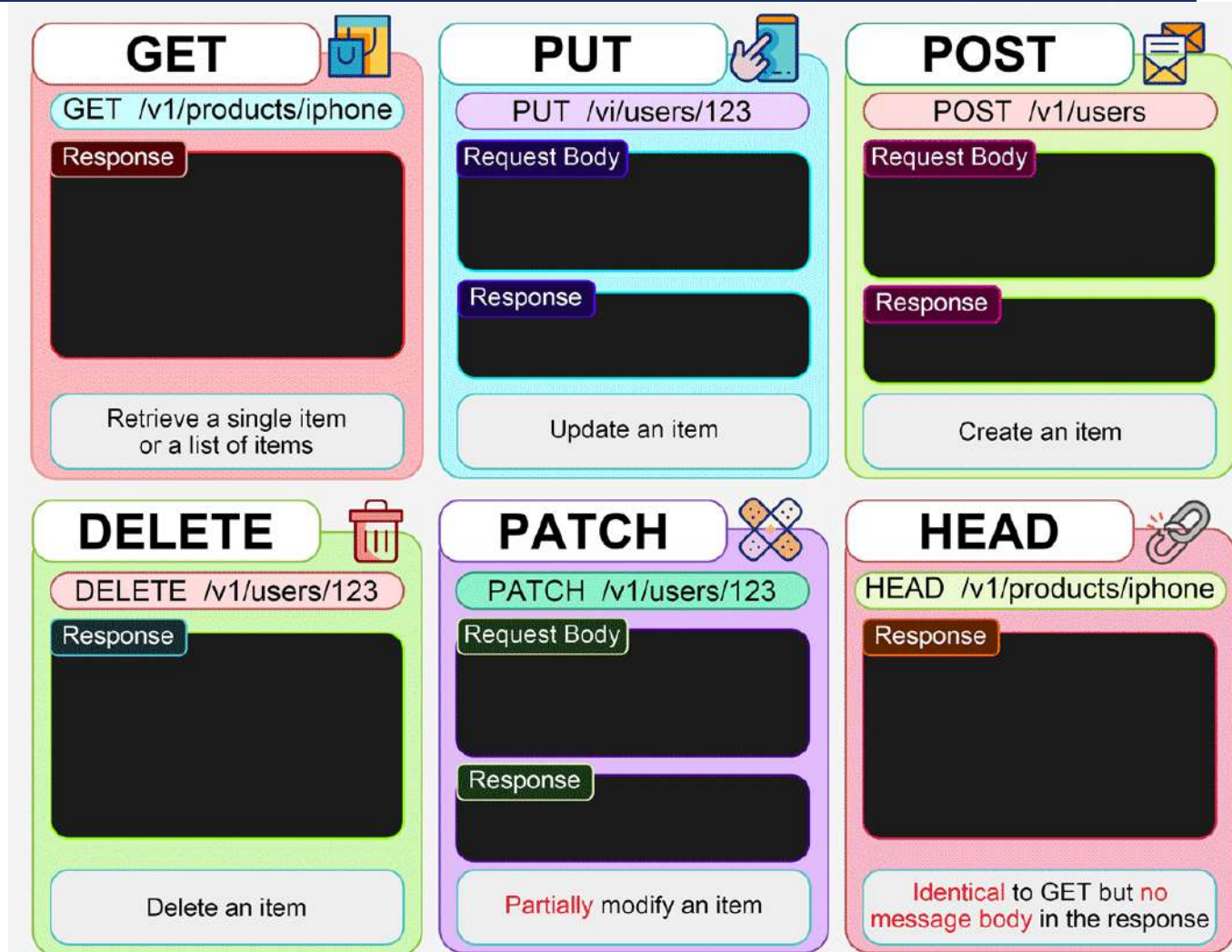
Fragment – The fragment (or hash) **is not passed to the server** at all; it is **strictly for use by the browser**.

- It can be used to jump over to a specific part of the webpage.
- Some **single-page applications** use the fragment to control application navigation.

HTTP REQUEST METHODS

Common HTTP request methods for backend development include:

- GET request
- POST request
- PUT request
- PATCH request
- DELETE request



HTTP REQUEST METHODS

- **GET Request** – for retrieving data from the server. Can be used to:
 - Fetch a list of resources (<http://www.websitename.com/products>).
 - Fetch a specific resource by its identifier (<http://www.websitename.com/products/123>).
 - **POST Request** – for sending data to the server. Commonly use with:
 - Form submission.
 - Adding new items, users, etc.
- **POST Request is Not Idempotent.*** Sending multiple of the same POST requests may create duplicate resources.

HTTP REQUEST METHODS

- **PUT Request** – Update all information of the specified resource.
 - **Idempotent:** Sending the same PUT request multiple times results in the same state.
 - **Requires all fields for the update.**
- **PATCH Request** – Update partial information of the specified resource.
 - **Not Necessarily Idempotent:** depends on implementation.
 - **Only the fields provided in the request body are updated.**
- **Delete Request** – Delete a specified resource.

INTRODUCTION TO EXPRESS

Before showing examples on the HTTP request methods, let's get to know Express!

- **Express** is a **minimal and flexible web application framework** for Node.js.
- It simplifies the creation of web servers and APIs.

To be able to use Express for our project, we will need to install it first!


1. Create a new folder called "http-request".
2. Open your terminal (or from Visual Studio Code Terminal), type:

```
npm install express
```


INTRODUCTION TO EXPRESS

JS app.js ×

JS app.js > [🔗] items

```
1  const express = require('express')
2  const app = express()
3  
4  let items = ['Item1', 'Item2', 'Item3']
5
6  // Retrieve all items
7  app.get('/items', (req, res) => {
8    res.json({ items })
9  })
10
11 // Retrieve a specific item by ID
12 app.get('/items/:id', (req, res) => {
13   const itemId = req.params.id
14   res.json({ id: itemId, name: `Item ${itemId}` })
15 })
16
17 app.listen(3000, () => console.log('Server is running on port 3000'))
```

Create **app.js**
with a simple
GET Requests.

On your browser, try:

- localhost:3000/items
- localhost:3000/items/5

INTRODUCTION TO EXPRESS

Adding POST Request

```
// For parsing JSON request bodies
app.use(express.json())

app.post('/items', (req, res) => {
  const {item} = req.body

  // Response with a status 201 and save items
  res.status(201).json({ message: 'Item created', item: item })
  items.push(item)
  console.log(items)
})
```

INTRODUCTION TO EXPRESS

Adding PUT Request

```
app.put('/items/:id', (req, res) => {  
  const {id} = req.params  
  const {item} = req.body  
  
  // Replace the existing item based on id (index)  
  if (id > items.length - 1) {  
    res.status(404).json({ message: `Item ${id} not found` })  
  } else {  
    res.json({ message: `Item ${id} fully updated`, item: item })  
    items[id] = item  
    console.log(items)  
  }  
})
```

QUICK RECAP

- When visiting a website, client is making requests to the server.
- That includes **simply browsing**, as well as **adding or modifying information on the website**.
- For the most common HTTP request, we have:
 - **GET** – for getting a list of resources or a specified resource by id.
 - **POST** – for adding new item to existing resources.
 - **PUT** – for updating all fields/attributes of a specified resource by id.
 - **PATCH** – for updating some fields/attributes of a specified resource by id.
 - **DELETE** – for deleting a specified resource by id.

QUICK RECAP

- We installed express for our project.
- Express will have to be installed per project (not globally).
- Express provide the boilerplate function for GET, POST, PUT, PATCH, and DELETE that can we can straightly use to handle the HTTP requests.
- We can use Postman for HTTP requests, specifically POST, PUT, PATCH, and DELETE.

INTRODUCTION TO EXPRESS

In class Exercise:

Work in pairs:

1. Try adding DELETE request
2. Try to modify the array into an array of object with id, name, and price.
3. Modify the previous GET, POST, and PUT requests.
4. Add PATCH requests.

Submit your code file (.js) to Canvas.

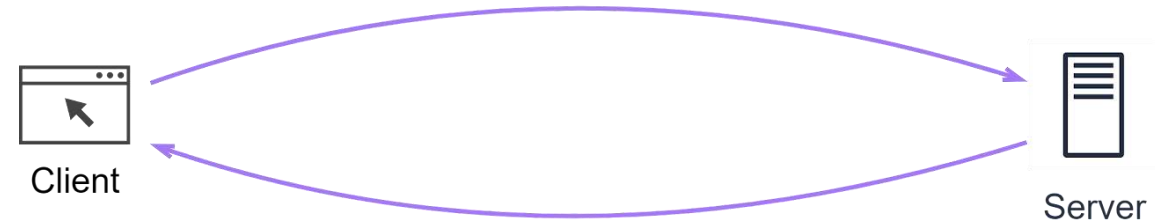
REQUEST & RESPONSE HEADER

- Beside sending and receiving the request and response data between clients and servers, **additional hidden information are also sent along with the response and request.**
- These are called the
 - **Request header** – sent along with the request made on client side.
 - **Response header** – sent along with the response from server side.
- They contain metadata on critical information such as type of data being sent, its length, how it's compressed, and more.

REQUEST & RESPONSE HEADER

- **User-agent** (Browser), **Server** – the types of server (Apache/Nginx), **Host**
- **Content-Length**: indicates the size of the body of the request or response in bytes.
 - This helps the receiver understand when the current message ends and potentially prepare for the next one, especially in cases where multiple HTTP messages are being sent over the same connection.
- **Content-type**: the format of the data it's receiving.

Request Line	GET /product/iphone HTTP/1.1
Request Header	User-agent: Chrome xxx Host: www.mywebsite.com Accept: text/html Accept-language: en Connection: Keep-Alive Keep-Alive: timeout=10, max=500 Accept-Encoding: gzip
Body	N/A



Response Line	HTTP/1.1 200 OK
Response Header	Date: xxx Server: Apache xxx Last-modified: xxx Content-length: 321 Content-type: text/html Content-Encoding: gzip
Body	<HTML> <HEAD>iphone</HEAD> <BODY> <H1>iPhone 14</H1> <P>This is an iPhone 14.</P> </BODY> </HTML>

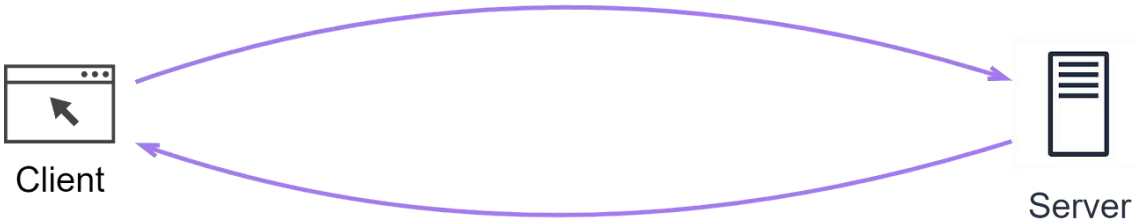
REQUEST & RESPONSE HEADER

- **Content-encoding:** indicates the compression format used for the data.

Eg. seeing 'gzip' encoding, means it needs to decompress the data.

- **Connection:** specify the types of connection. Whether it's:
 - Non-persistent (close) – close connection after a single request. Typically used in an older, legacy server.
 - Persistent (keep-alive) – keep the connection on for multiple requests.

Request Line	GET /product/iphone HTTP/1.1
Request Header	User-agent: Chrome xxx Host: www.mywebsite.com Accept: text/html Accept-language: en Connection: Keep-Alive Keep-Alive: timeout=10, max=500 Accept-Encoding: gzip
Body	N/A



Response Line	HTTP/1.1 200 OK
Response Header	Date: xxx Server: Apache xxx Last-modified: xxx Content-length: 321 Content-type: text/html Content-Encoding: gzip
Body	<HTML> <HEAD>iphone</HEAD> <BODY> <H1>iPhone 14</H1> <P>This is an iPhone 14.</P> </BODY> </HTML>

REQUEST & RESPONSE OBJECT

- **Request & Response Object** – Passed as an argument for the callback function when handling HTTP request.

```
app.get('/items', (req, res) => {  
  res.json(items)  
})
```

- **req** – the request object
- **res** – the response object

REQUEST & RESPONSE OBJECT

The Request Object

- **req.params**: An array containing the named route parameters. More on that in later chapters.
- **req.query**: An object containing querystring parameters (sometimes called GET parameters) as name/value pairs.
- **req.body**: An object containing POST parameters. It is so named because POST parameters are passed in the body of the request, not in the URL as querystring parameters are.

REQUEST & RESPONSE OBJECT

The Request Object

- `req.cookies/req.signedCookies`: Objects containing cookie values passed from the client.
- `req.headers`: The request headers received from the client.
- `req.accepts(types)`: A convenience method to determine whether the client accepts a given type or types (optional types can be a single MIME type, such as `application/json`, a comma-delimited list, or an array).

REQUEST & RESPONSE OBJECT

The Response Object

- `res.status(code)`: Sets the HTTP status code. Express defaults to 200 (OK), so you will have to use this method to customize the return status.
- `res.send(body)`: Sends a response to the client. Express defaults to a content type of text/html, so if you want to change it to text/plain, you'll have to call `res.type('text/plain')` before calling `res.send`.
- `res.json(json)`: Sends JSON to the client.
- `res.end()`: Ends the connection without sending a response.



Form Handling

FORM HANDLING

Sending Client Data to the Server:

So far, we know that sending data from client to the server can be done through:

- **QueryString** in GET request (in the URL)
- **Request body** (in POST, PUT, PATCH request)

Generally, it is possible to do everything with GET request.

BUT it is NOT recommended to use GET request to Create or Update data as the URL could get messy with large data, and it is less secured.

Eg. It is very challenging to perform encryption on GET request as it is located in the URL.

FORM HANDLING

- Previously, we've used Postman to test our basic POST, PUT, PATCH, and DELETE requests.
- From Client side, we can do that using Forms.
- Forms are commonly used for:
 - User registration/login
 - Data submission
 - File uploads
 - Search interfaces
 - Survey/feedback collection

FORM HANDLING

Form Processing Flow:

1. User fills out form in browser
2. Browser sends data to server
3. Server processes the data
4. Server sends response
5. Browser handles the response

FORM HANDLING – KEY ATTRIBUTES IN FORMS

```
<form action="/submit-endpoint" method="POST" enctype="application/x-www-form-urlencoded">
|   <!-- Form controls go here -->
| </form>
```

- **action:** Server endpoint that receives the data
- **method:** HTTP method used for submission. (Only support GET and POST)
 - GET: Data in URL – QueryString (searchable, bookmarkable)
 - POST: Data in request body (secure, larger data)
- **enctype:** How form data is encoded
 - application/x-www-form-urlencoded (default)
 - multipart/form-data (for file uploads)
 - text/plain (rarely used)

FORM HANDLING

Example:

1. First, let's create a new project folder called "**form-handling**".
2. Inside the folder, create **app.js**
3. Install express (npm install express)
4. Create another folder called "**public**"
5. Inside public, create **register.html** (We will create our HTML form there).

Project Structure:

```
▼ FORM-HANDLING
  > node_modules
  ▼ public
    | <> register.html
  JS app.js
  {} package-lock.json
  {} package.json
```

register.html has been uploaded to Canvas. You may use the HTML code for this example.

FORM HANDLING

```
<form action="/register" method="POST">
  <div>
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required>
  </div>

  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
  </div>
</form>
```

- `action="/register"` means the endpoint is `/register`. Our server will handle the POST request from that end point.
- `name="username"` and `name="email"` means we can access the data through `req.body.username` and `req.body.email`

FORM HANDLING

Inside our app.js

- We're adding two middlewares for **handling form data** (`express.urlencoded()`) and **serving static file** (`express.static()`).
- In our GET request, when visiting the homepage (localhost:3000), we will server the register.html page.
- In our POST request, we simply logged the received data and send an HTML response back.

```
const express = require('express')
const app = express()
const path = require('path')

// Middleware for handling form data and file path
app.use(express.urlencoded({ extended: true }))
app.use(express.static('public'))

// Serve the HTML form
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'register.html'))
})

// Handle form submission
app.post('/register', (req, res) => {
  const { username, email, password, age, interests } = req.body

  // Log the received data
  console.log('New Registration:', {
    username,
    email,
    age,
    interests: Array.isArray(interests) ? interests : [interests]
  })

  // Send response
  res.send(`
    <h2>Registration Successful!</h2>
    <p>Welcome, ${username}!</p>
    <p>Confirmation email sent to: ${email}</p>
  `)
})

app.listen(3000, () => console.log('Server is running on port 3000'))
```

FORM HANDLING

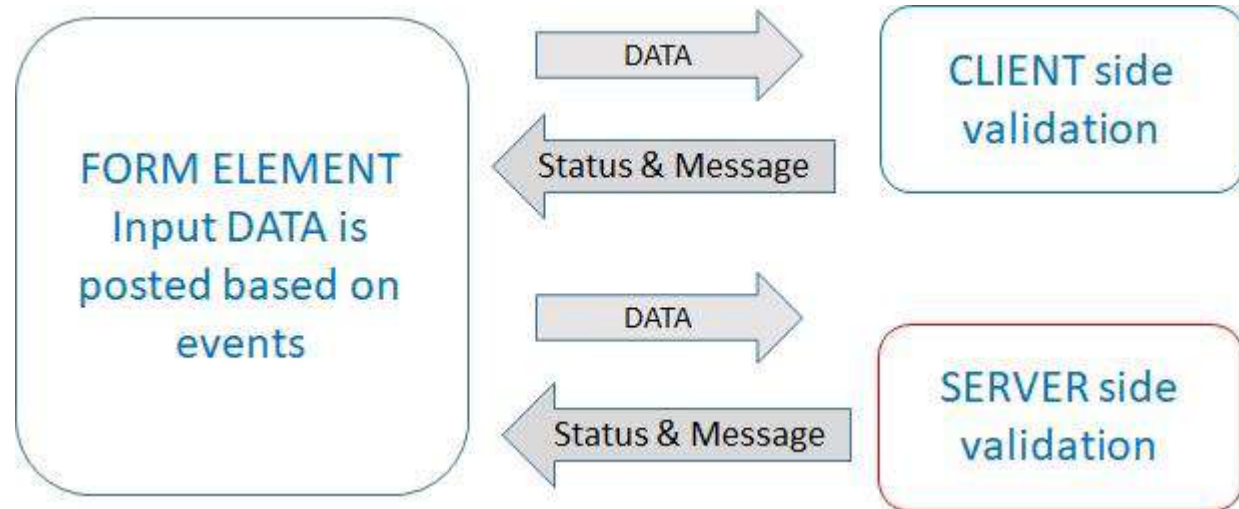
Today, we will talk about **Form Validation**.

What is Form Validation?

Why is it important?

FORM HANDLING – VALIDATION

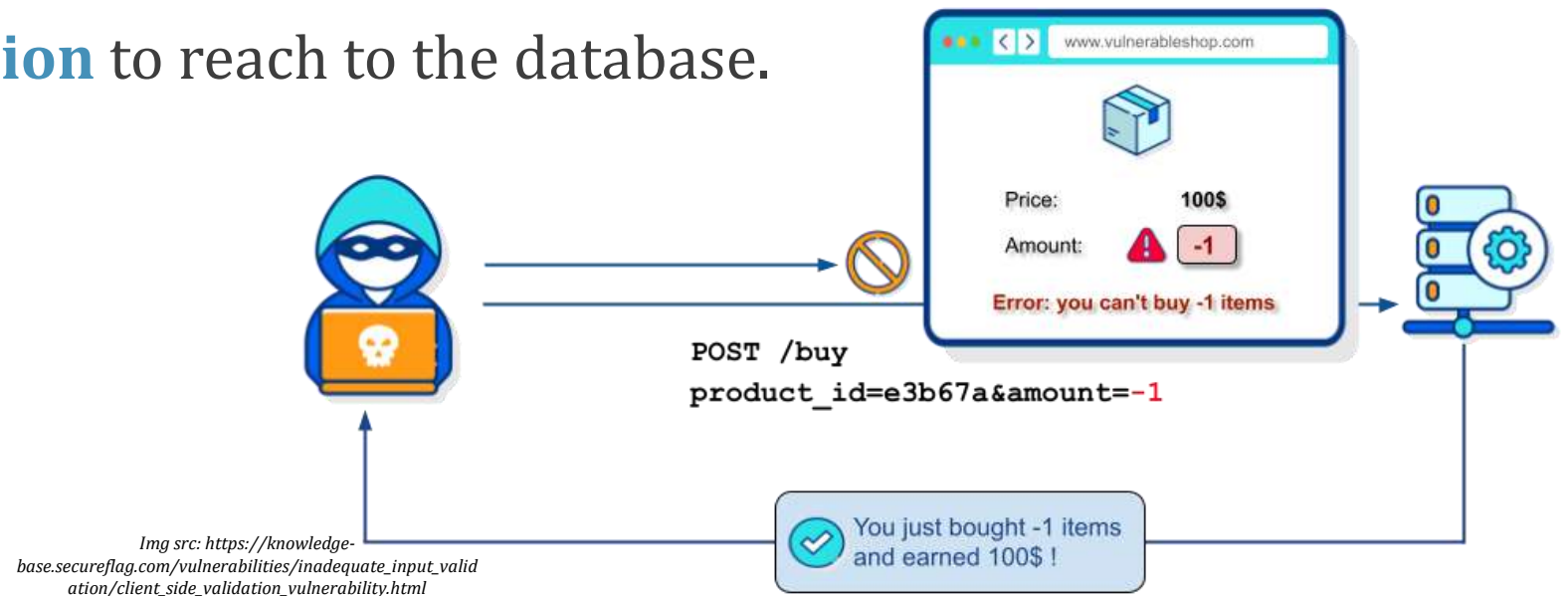
- **Form Validation** refers to the process of checking whether the **user inputs** to the form are **complete, accurate, and fulfils certain criteria** before processing them.
- Even though Form Validation should occur on client-side, we, on **the server-side must also perform validation of the submitted data as well.**



FORM HANDLING – VALIDATION

Benefits of form validation:

- **Ensure data integrity** – valid and real email addresses, phone number, etc.
- **Prevents security risk** – make user choose secured password, prevent attacks like Cross-Site Scripting (XSS) by rejecting scripts embedded in form inputs.
- **Prevent invalid submission** to reach to the database.



FORM HANDLING – VALIDATION

Adding Validations to our previous code:

```
// Validation functions
const validateUsername = (username) => {
  if (!username) return 'Username is required'
  if (username.length < 3) return 'Username must be at least 3 characters'
  if (username.length > 50) return 'Username must be less than 50 characters'
  return null
}

const validateEmail = (email) => {
  if (!email) return 'Email is required'
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/
  if (!emailRegex.test(email)) return 'Invalid email format'
  return null
}
```

```
app.post('/register', (req, res) => {  
  const { username, email, password, age, interests } = req.body
```

In our POST request

```
  // Collect all validation errors
```

```
  const errors = {  
    username: validateUsername(username),  
    email: validateEmail(email)  
  }
```

```
  // Filter out null errors
```

```
  const actualErrors = Object.entries(errors)  
    .filter(([_, value]) => value !== null)  
    .reduce((acc, [key, value]) => ({ ...acc, [key]: value }), {})
```

```
  // If there are errors, send them back
```

```
  if (Object.keys(actualErrors).length > 0) {  
    return res.status(400).send(`  
      <h2>Registration Failed</h2>  
      <ul>  
        ${Object.values(actualErrors).map(error => `<li>${error}</li>`).join('')}  
      </ul>  
      <p><a href="/">Back to Registration</a></p>  
    `)  
  }
```

**Add these to your POST request.
The rest are the same.**

FORM HANDLING – VALIDATION

Practice

- Try adding validation for Password and Age.
- Submit it to Canvas for [In-class Practice] Form Validation

***[Optional] You can also explore **express-validator** package for standard form validation.*

FORM HANDLING – FILE UPLOAD

- In general, the type of data we accept from users are not only text, but could also be files.

Example: Uploading profile picture, Uploading CV, etc.

- As mentioned previously, we have the encoding multipart/form-data for file uploads using form.
- On Express side, we need a middleware called **multer** to handle file uploads.

FORM HANDLING – FILE UPLOAD

Let's get started with the examples.

1. We can still use our same project file.
2. In the folder “public”, add another file call **fileUpload.html**.
3. Install multer: `npm install multer`
4. Create a new JavaScript file for this: **appFileUpload.js**

FORM HANDLING – FILE UPLOAD

***fileUpload.html has been uploaded to Canvas.*

- For our html code, as we're sending data, the method is still POST.
- However, the encoding type is now changed to “multipart/form-data”

```
<form action="/upload" method="POST" enctype="multipart/form-data">  
  <label for="file">Choose a file:</label>  
  <input type="file" name="file" id="file" required>  
  <button type="submit">Upload</button>  
</form>
```

```
const express = require('express')
const multer = require('multer')
const app = express()
const path = require('path')

app.use(express.static('public'))

// Configure Storage for Uploaded Files
const storage = multer.diskStorage({
  destination: './uploads/', // Directory to store files
  filename: (req, file, cb) => {
    cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname))
  }
})

const upload = multer({ storage: storage })

app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'fileUpload.html'))
})

// File Upload
app.post('/upload', upload.single('file'), (req, res) => {
  if (!req.file) return res.status(400).send('No file uploaded.')
  res.send(`File uploaded: ${req.file.filename}`)
})

app.listen(3000, () => console.log('Server running on port 3000'))
```

Multer's storage configuration:

- **destination:** Where files will be stored
- **filename:** Generates a unique name for the file using `Date.now().upload`
- **single('file'):** Handles a single file with the field name "file". The file details are accessible via `req.file`.

EXERCISE

- Figure out how to upload multiple files
- And restricting file types (for example: only jpg, jpeg, and png for images)

REFERENCES

1. Brown, E. (2020). *Web Development with Node and Express: Leveraging the JavaScript Stack*. 2nd Edition “O’Reilly Media, Inc.” – Chapter 6 & 8.
2. ByteByteGo. (2023, July 13). *The foundation of REST API: HTTP*. ByteByteGo Newsletter. <https://blog.bytebytego.com/p/the-foundation-of-rest-api-http>

ANNOUNCEMENT

Midterm Exam next week (**27th Feb**)

- Paper-based. There will be a short coding part.
- More details in the Revision session next Monday (24th Feb).

Final Project Announcement