MVC-based JavaScript Web App

Outline

1. Web and HTTP
2. MVC-based Web applications
3. Node.js Express Framework
4. Web API
5. View Template using Handlebars
6. Server-side Rendering of Views
7. Fetch API

Web Client

Request

Response

Web Server

Frontend development

- HTML for page structure
- CSS for styling
- JavaScript for interaction
- AJAX for partial page updates (without reload)

Backend development

- Web API
- Views Template
- Data storage


WWW

Web and HTTP

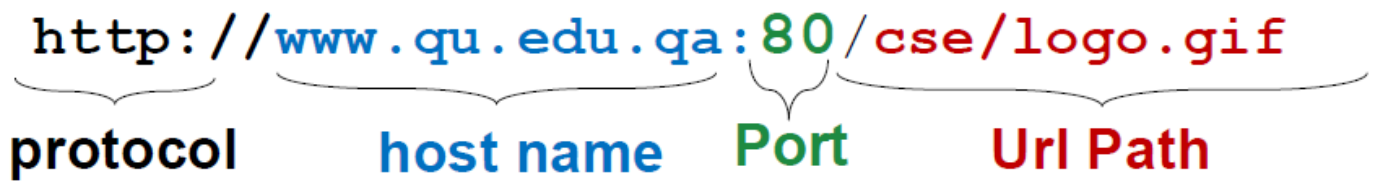

GET /index.html
HTTP/1.1
HTTP/1.1 200 OK
"Welcome to our
Web site!"

What is Web?

- Web = **global distributed system of interlinked hypertext documents accessed over the Internet using the HTTP protocol to serve billions of users worldwide**
  - Consists of set of **resources** located on different servers:

    HTML pages, images, videos and other resources
  - Resources have unique **URL** (Uniform Resource Locator) address
  - Accessed through standard protocols such as HTTP
- The Web has a Client/Server architecture:
  - **Web server** sends resources in response to requests (using HTTP protocol)
  - **Web browser** (client) requests, receives (using HTT protocol) and displays Web resources

Uniform Resource Locator (URL)



- URL is a formatted string, consisting of:

– **Protocol** for communicating with the server (e.g., http, ftp, https, ...)

– **Name of the server or IP** address plus port (e.g. qu.edu.qa:80, localhost:8080)

– **Path of a resource** (e.g. **/directory/index.php**)

– **Parameters** aka **Query String** (optional), e.g.

https://www.google.com/search**?q=qatar%20university**

URL Encoding

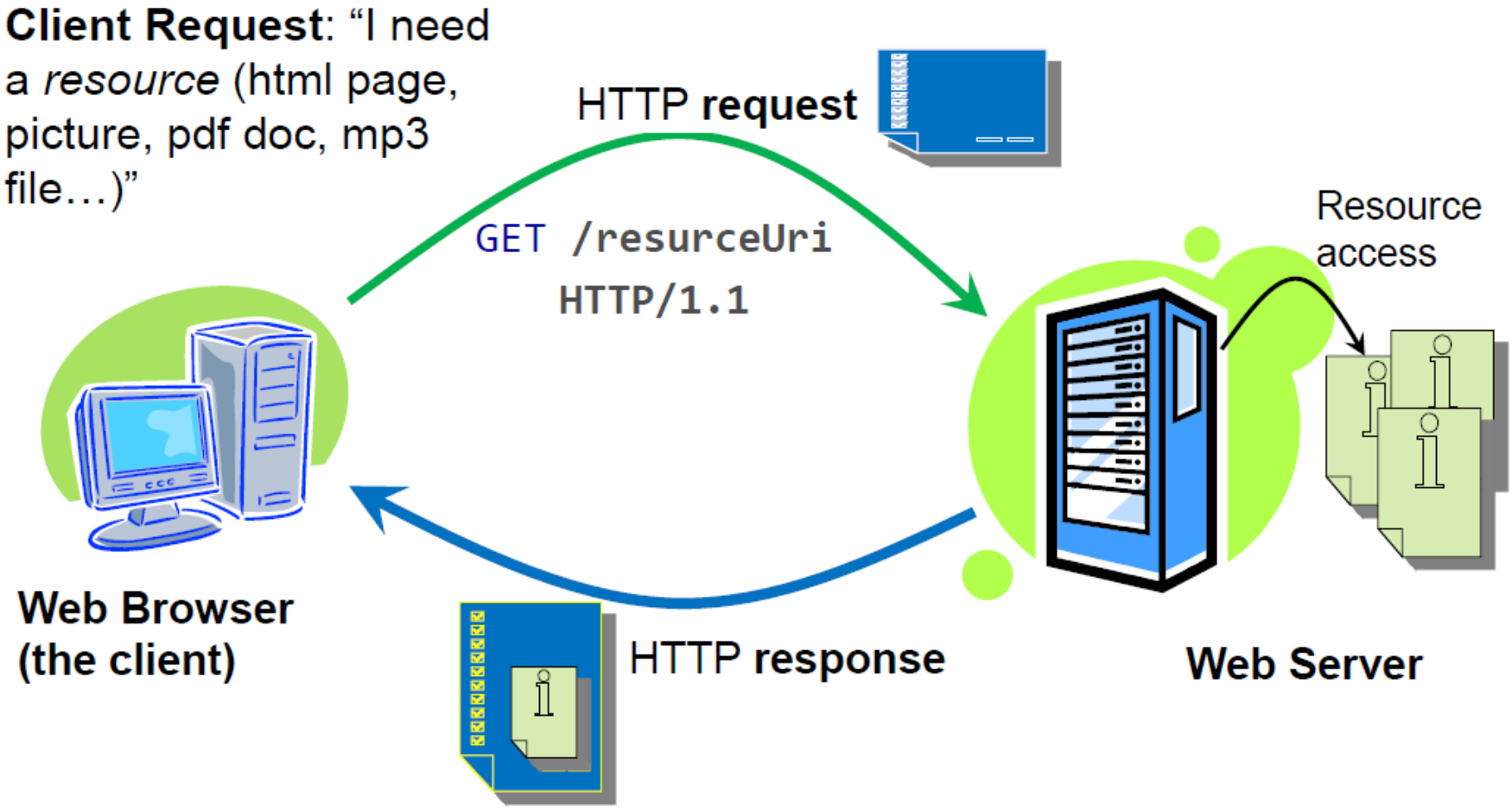
- According RFC 1738, the characters allowed in URL are alphanumeric [0-9a-zA-Z] and the special characters $-_.+!*'()
- Unsafe characters should be encoded, e.g.,

  http://google.com/search?q=qatar%20university

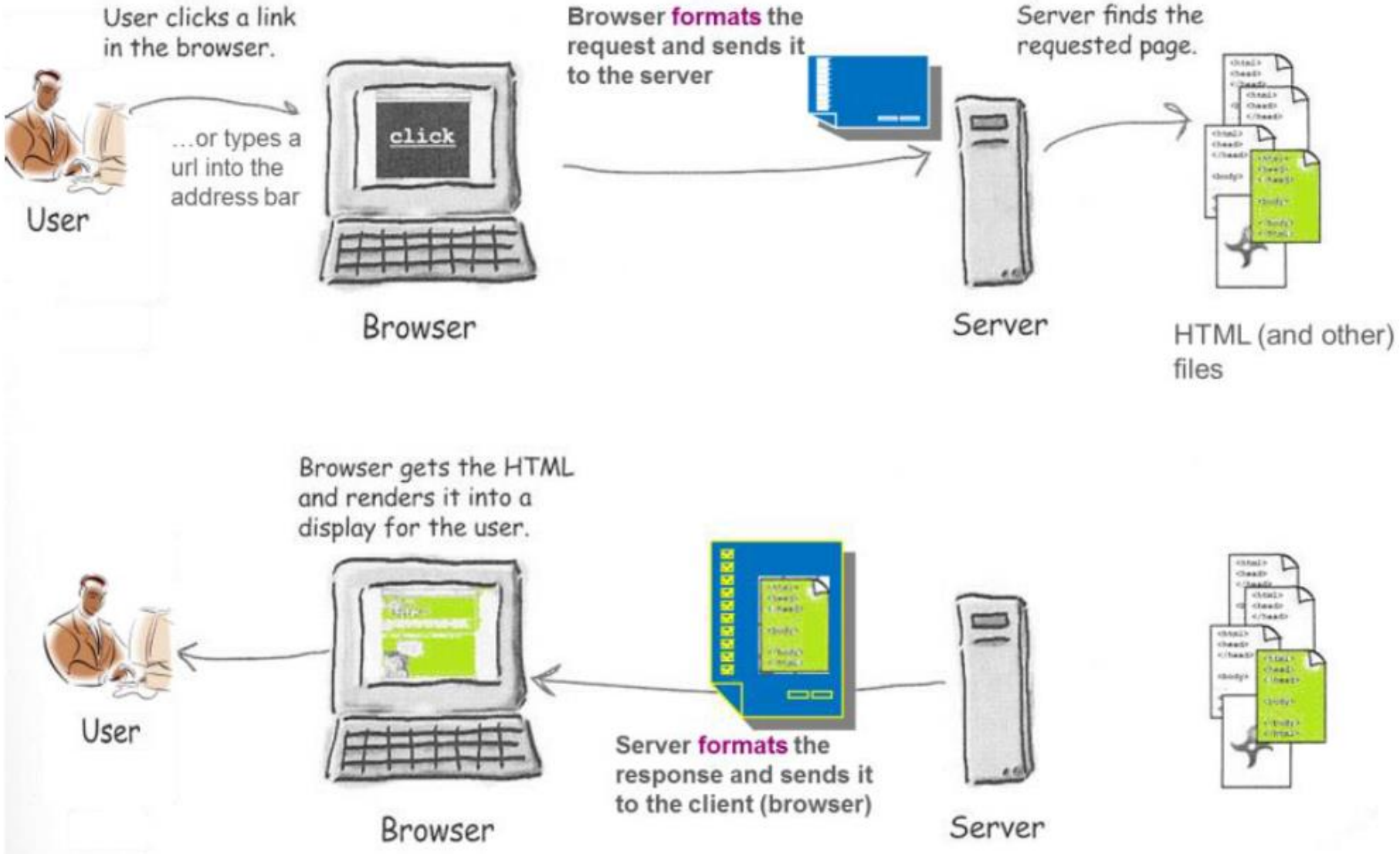
Commonly encoded values:

ASCII Character	URL-encoding
space	%20
!	%21
"	%22
#	%23
$	%24
%	%25
&	%26

Web uses Request/Response interaction model
HTTP is the *message protocol* of the Web

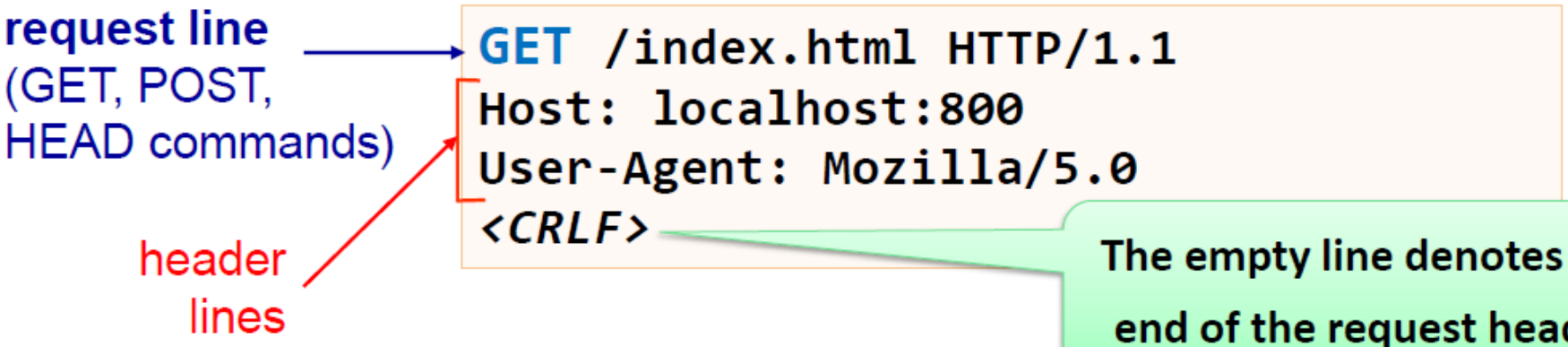


The sequence for retrieving a resource

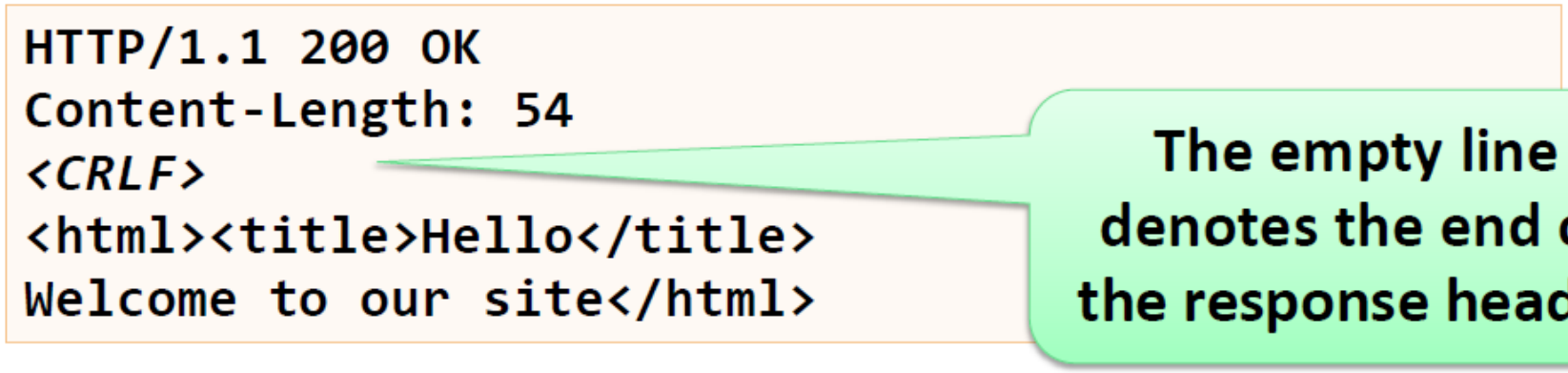


Request and Response Examples

- **HTTP request:**



- **HTTP response:**



HTTP Request Message

- Request message sent by a client consists of

– **Request line** – request method (GET, POST, HEAD, ...), resource URI, and protocol version

– **Request headers** – additional parameters

– **Body** – optional data

  - e.g. posted form data, files, etc.

```
<request method> <URI> <HTTP version>
<headers>
<empty line>
<body>
```

HTTP Request Methods

- **GET**
  - – **Retrieve a resource** (could be static resource such as an image or a dynamically generated resource)
  - – Input is appended to the request URL E.g.,

  http://google.com/?q=Qatar
- **POST**
  - – **Create or Update a resource**
  - – Web pages often include form input. Input is submitted to server in the **message body.** E.g.,


```
POST /calc HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
num1=20&operation=*&num2=10
```

HTTP Response Message

- Response message sent by the server

– **Status line** – protocol version, status code, status phrase

– **Response headers** – provide metadata such as the Content-Type

– **Body** – the contents of the response (i.e., the requested resource)

```
<HTTP version> <status code> <status text>
<headers>
<empty line>
<response body>
```

HTTP Response – Example

status line (protocol status code status text)

Try it out and see HTTP in action using **HttpFox**

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: QU Web Server
Content-Length: 131
<CRLF>
<html>
  <head><title>Calculator</title></head>
  <body>20 * 10 = 200
      <br><br>
      <a href='/calc'>Calculator</a>
  </body>
</html>
```

HTTP response headers

The empty line denotes the end of the response header

Response body. e.g., requested HTML file

Common Internet Media Types

- The **Content-Type** header describes the media type contained in the body of HTTP message
- **Full list @** http://en.wikipedia.org/wiki/MIME_type
- Commonly used media types (**type/**subtype):

Type/Subtype	Description
application/json	JSON data
image/gif	GIF image
image/png	PNG image
video/mp4	MP4 video
text/xml	XML
text/html	HTML
text/plain	Just text

HTTP Response Codes

- Status code appears in 1st line in response message
- HTTP response code classes
  - 2xx: success (e.g., “200 OK”)
  - 3xx: redirection (e.g., "302 Found")

  "302 Found" is used for redirecting the Web browser to another URL
  - 4xx: client error (e.g., “404 Not Found”)
  - 5xx: server error (e.g., “503 Service Unavailable”)

Popular Status Codes

Code	Reason	Description
200	OK	Success!
301	Moved Permanently	Resource moved, don't check here again
302	Moved Temporarily	Resource moved, but check here again
304	Not Modified	Resource hasn't changed since last retrieval
400	Bad Request	Bad syntax?
401	Unauthorized	Client might need to authenticate
403	Forbidden	Refused access
404	Not found	Resource doesn't exist
500	Internal Server Error	Something went wrong during processing
503	Service Unavailable	Server will not service the request

Browser Redirection

- HTTP browser redirection example

– HTTP GET requesting a moved URL:

(Request-Line)	GET /qu HTTP/1.1
Host	localhost:800
User-Agent	Mozilla/5.0 (Windows NT 6.3; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

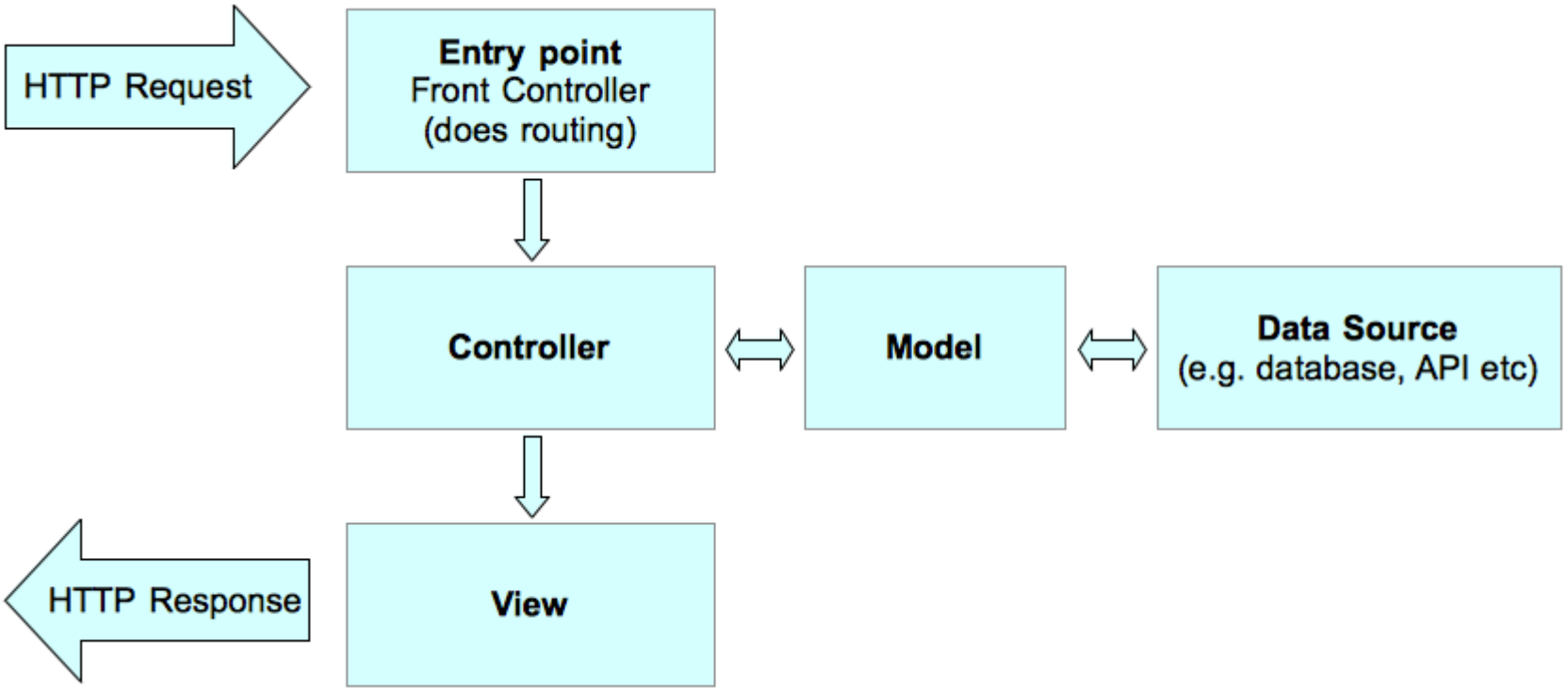
– The HTTP response says that the browser should request another URL:

(Status-Line)	HTTP/1.1 301 Moved Permanently
Location	http://qu.edu.qa

Typical server steps to process an HTTP Request

- Parse the HTTP request (i.e., convert a textual representation of the request into an object)
- Generate a response either static one by reading a file or a dynamic response
  - Dynamic response could be either generated programmatically from scratch or it could be generated by filling-up a page template read from a file
- Send the response to the client including:
  - Response headers
  - Response body

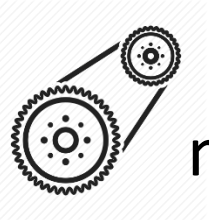
MVC-based Web applications



MVC-based Web application

Controller

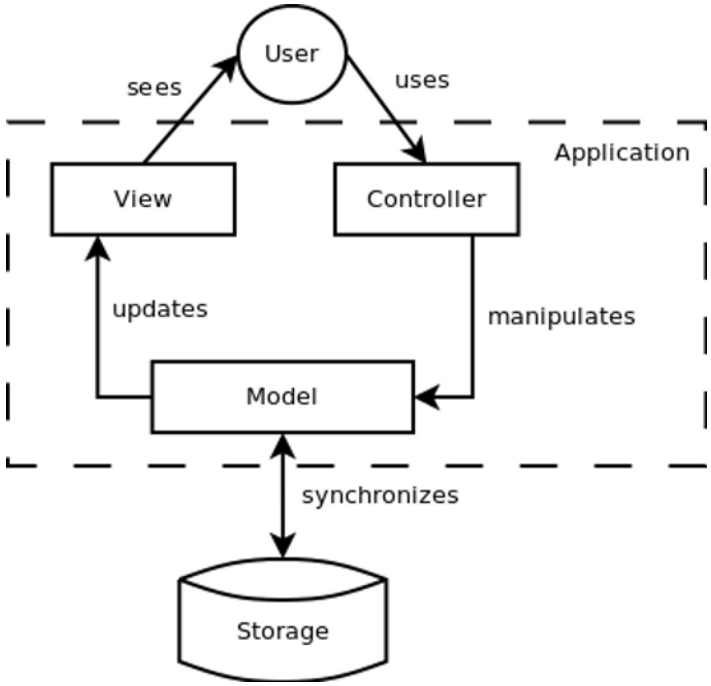
- accepts incoming requests and user input and **coordinates** request handling
- instructs the model to perform actions based on that input
  - e.g. add an item to the user's shopping cart
- decides what view to display for output



Model : implements business logic and computation, and manages application's data

View : responsible for

- collecting input from the user
- displaying output to the user

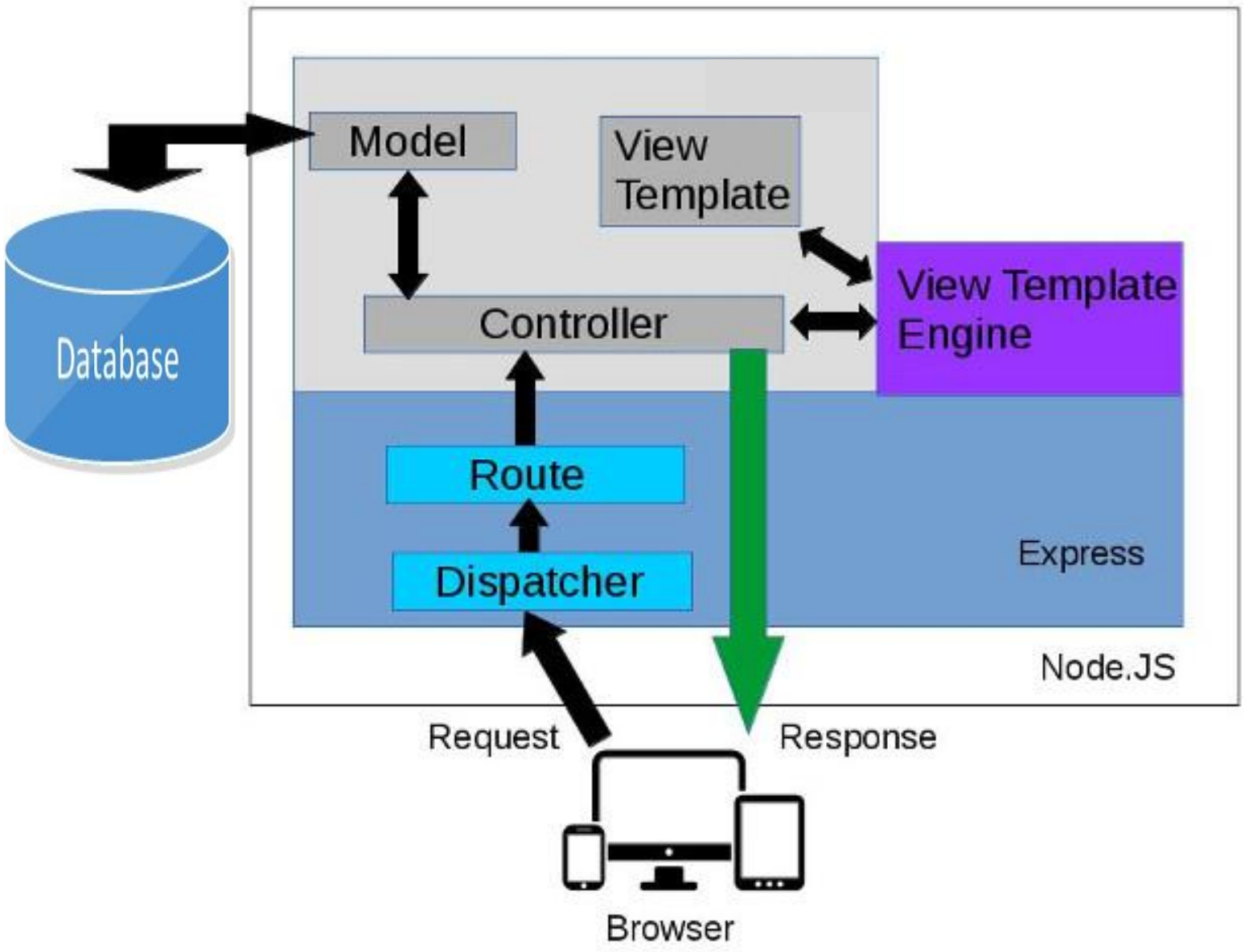


Advantages of MVC

- ***Separation of concerns***
  - Views, controller, and model are separate components. This allows modification and change in each component without significantly disturbing the others.
    - Computation is not intermixed with Presentation. Consequently, code is cleaner and easier to understand and change.
- **Flexibility**
  - The view component, which often needs changes and updates to keep the users continued interests, is separate
    - The UI can be completely changed without touching the model in any way
- **Reusability**
  - The same model can used by different views (e.g., Web view and mobile view)
- **Allows for parallel work**, e.g., a UI designer can work on the View while a software engineer works on the Controller and Model

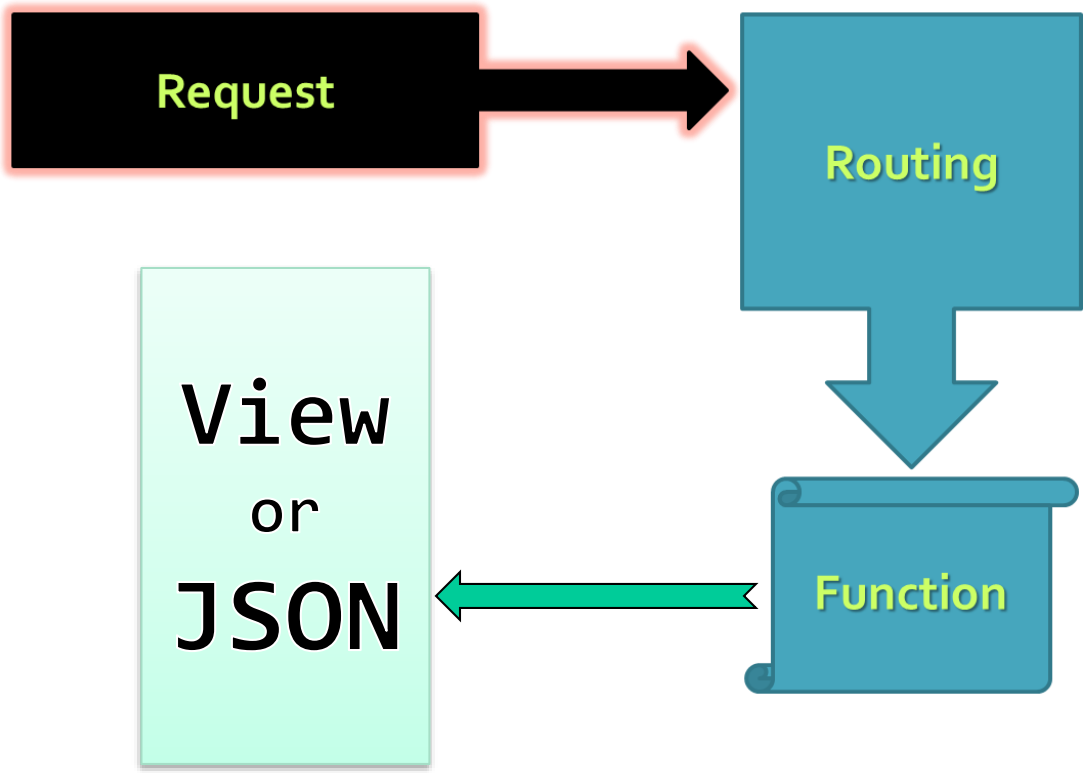
MVC is widely used and recommended particularly for interactive web-applications

MVC using Node.js Express



Express

Web Application Framework for Node.js



Create and Start an Express App

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

app.get('/', (req, res) => {
    res.send('السلام عليكم ورحمة الله وبركاته')
})

let port = 3000
app.listen(port, () => {
    console.log(`App is available @ http://localhost:${port}`)
})
```

- A function **registered** to listen to the URL http://localhost:3000/
- When someone visits this Url the function associated with get '/' will run and 'السلام عليكم ورحمة الله وبركاته' will be returned to the requester

Routing

- Routing is a way to **map** of an HTTP verb (like GET or POST) and a URI (like /users/123) to a **handler**

- To a receive a **query string**, a **parameter** can be added to the route uri with a colon in front of it. To grab the value, you'll use the **params** property of the request

```
app.get('/api/students/:id', (req, res) => {
    let studentId = req.params.id
    console.log('req.params.id', studentId)
})
```


26

Route Parameters

- Route parameters are **named** URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the **req.params** object

```
app.get('/users/:userId/books/:bookId', (req, res) => {
    // If the Request URL was http://localhost:3000/users/34/books/8989
    // Then req.params: { userId: "34", bookId: "8989" }
    res.send(req.params)
})
```


27

Post Example

```
<form method="post" action="/">
  <input type="text" name="username" />
  <input type="text" name="email" />
  <input type="submit" value="Submit" />
</form>
```

```
/* body-parser extracts URL encoded text from the
body of the incoming request and assigns it to req.body */
app.use(express.bodyParser())

app.post('/', (req, res) => {
    console.log(req.body)
    res.send('Welcome ' + req.body.username)
})
```

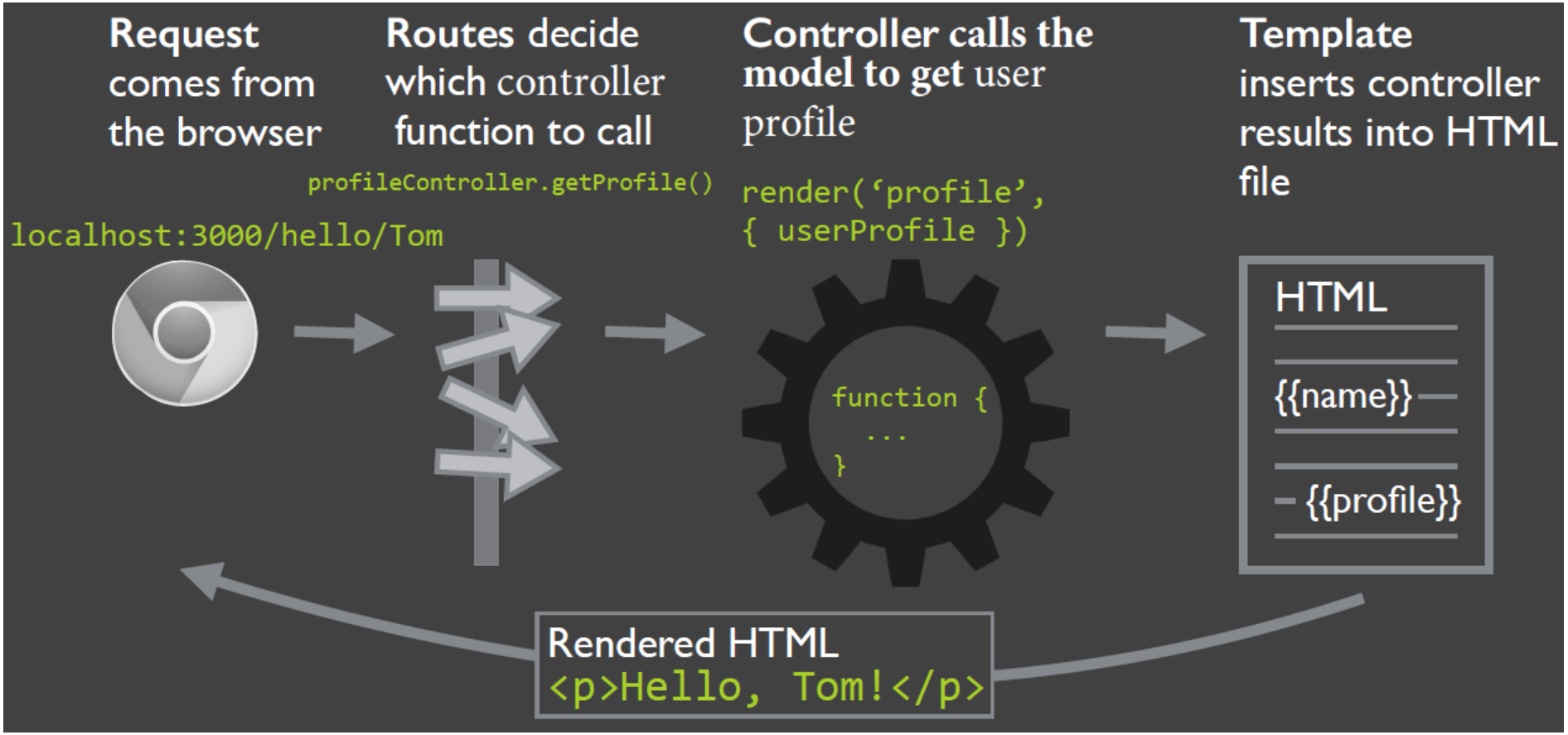
Express Router

- For simple app routes can defined in app.js
- For large application, Express Router allows defining the routes in a separate file(s) then attaching routes to the app to:
  - Keep app.js clean, simple and organized
  - Easily find and maintain routes

```
// routes.js file
let router = express.Router()
router.get('/api/students', studentController.getStudents )

//app.js file - mount the routes to the app
let routes = require('./routes')
app.use('/', routes)
```

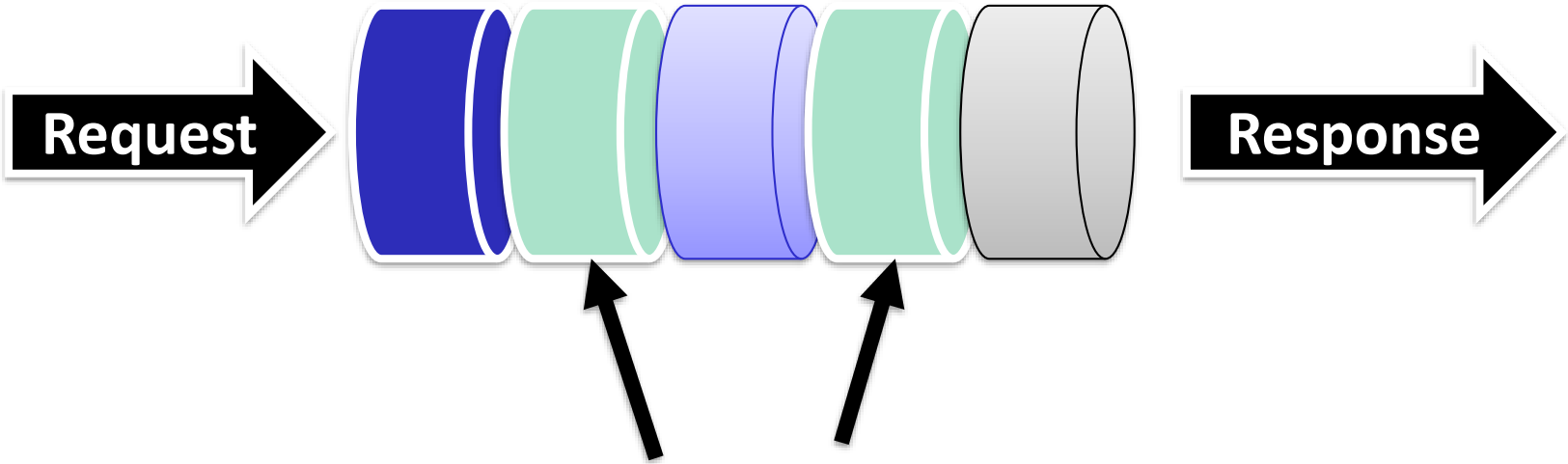
Interaction between App Modules



Express Middleware

- Express middleware allows you to **pipeline** a single request through a series of functions.
- Request Processing Pipeline: the request passes through an *array* of functions before it reaches your route handler. e.g.,

```
/* body-parser extracts URL encoded text from the
body of the incoming request and assigns it to req.body */
app.use( bodyParser.urlencoded({extended: true}) )
```

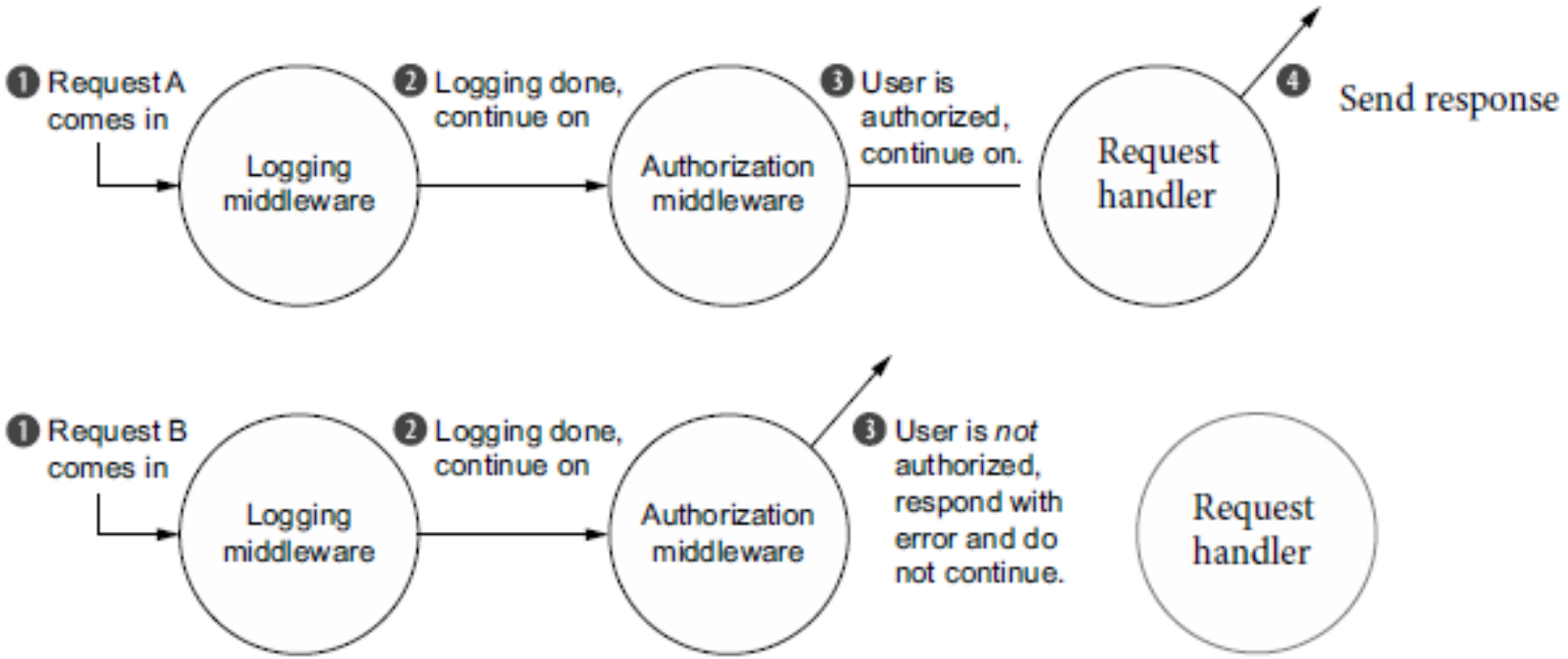


Middleware (bodyParser, logging, authentication, router etc.)

Middleware Example

- Each middleware function may modify the request or the response. This modularity make it easier to use and compose existing middleware packages such the middleware for serving static files

```
//Allow serving static files from __dirname which is the current folder)
app.use( express.static(__dirname) )
```



Custom Middleware Example

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

//Define a middleware function
function logger (req, res, next) {
    req.requestTime = new Date()
    console.log(`Request received at ${req.requestTime}`)
    next()
}

// Attach it to the app
app.use(logger)

app.get('/', function (req, res) {
    let responseText = `Hello World!<br>
        Requested at: ${req.requestTime}`
    res.send(responseText)
})

let port = 3000
app.listen(port, () => {
    let host = "localhost"
    console.log(`App is running and available @ http://${host}:${port}`)
})
```

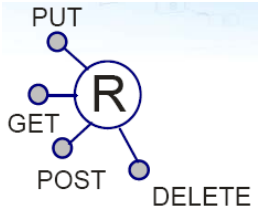
Web API
(aka REST Services)

What is a REST Service?

- Web API = Web accessible Application Programming Interface. Also known as REST Services.
- Web API in is a web service that accepts requests and returns **structured data** (JSON in most cases)
  - Programmatically accessible at a particular URL
  - You can think of it as a Web page returning json instead of HTML

- Major goal = **interoperability between heterogeneous systems**

REST Principles



- **Addressable Resources** (nouns): Identified by a URI

(e.g., http://example.com/customers/123)

- **Uniform Interface** (verbs): GET, POST, PUT, and DELETE

  -Use verbs to exchange application state and representation

  -Embracing HTTP as an Application Protocol

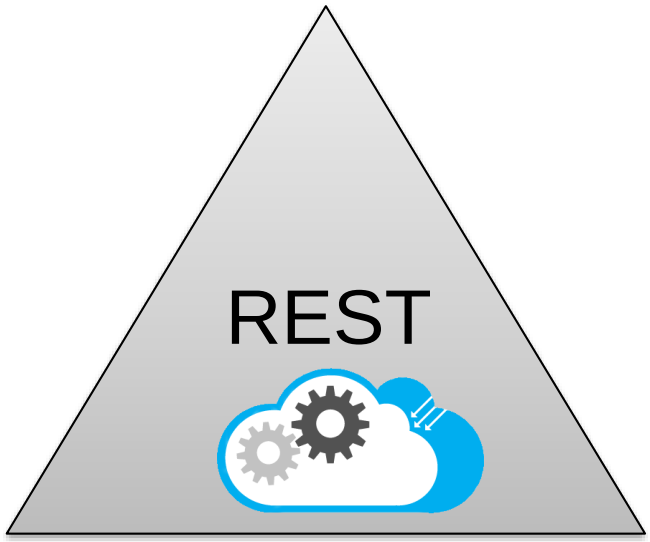
- **Representation-oriented**

  -**Representation of the resource state** transferred between client and server in a variety of data formats: XML, JSON, (X)HTML, RSS..

- **Hyperlinks** define relationships between resources and valid state transitions of the service interaction

REST Services Main Concepts

Nouns (Resources)
e.g., http://example.com/employees/12345



Verbs
e.g., GET, POST

Representations
e.g., XML, JSON

Resources

- The key abstraction in REST is a **resource**
- A resource is a conceptual mapping to a set of entities
  - Any **information that can be named can be a resource**: a document or image, a temporal service (e.g. "today's weather in Doha"), a collection of books and their authors, and so on
- Represented with a global identifier (URI in HTTP)
  - http://www.boeing.com/aircraft/747

Naming Resources

- REST uses URI to identify resources

Dedicated **api** path is recommended for better organization

  - http://localhost/**api**/books/
  - http://localhost/api/books/ISBN-0011
  - http://localhost/api/books/ISBN-0011/authors

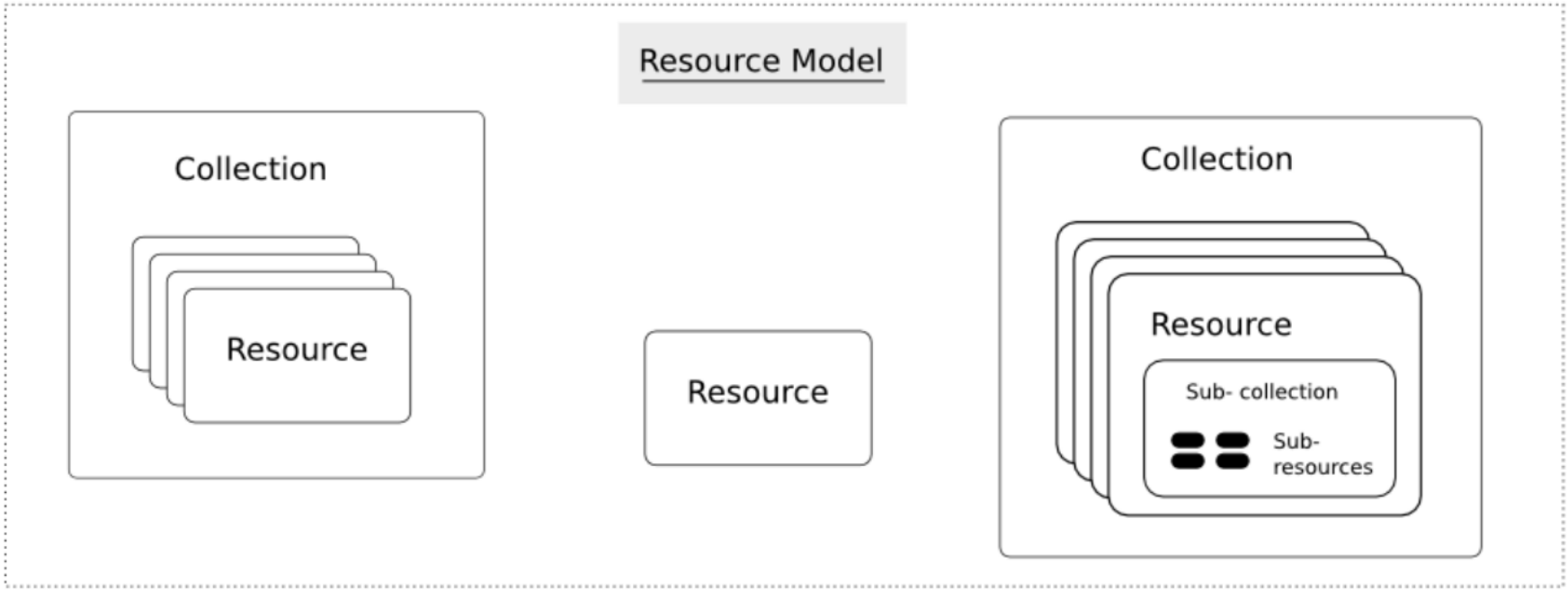
  - http://localhost/api/classes
  - http://localhost/api/classes/cmps356
  - http://localhost/api/classes/cs356/students

- As you traverse the **path** from more generic to more specific, you are navigating the data

Example CRUD (Create, Read, Update and Delete) API that manages books

- Create a new book
  - **POST** /books
- Retrieve all books
  - **GET** /books
- Retrieve a particular book
  - **GET** /books/:id
- Replace a book
  - **PUT** /books/:id
- Update a book
  - **PATCH** /books/:id
- Delete a book
  - **DELETE** /books/:id

A Collection with Resources



A Collection with Resources

A Singleton Resource

Sub-collections and Sub-resources

Representations

Two main formats:

- **JSON**

```
{
    code: 'cmps356',
    name: 'Enterprise Application Development'
}
```

- **XML**

```
<course>
    <code>cmps356</code>
    <name>Enterprise Application Development</name>
</course>
```

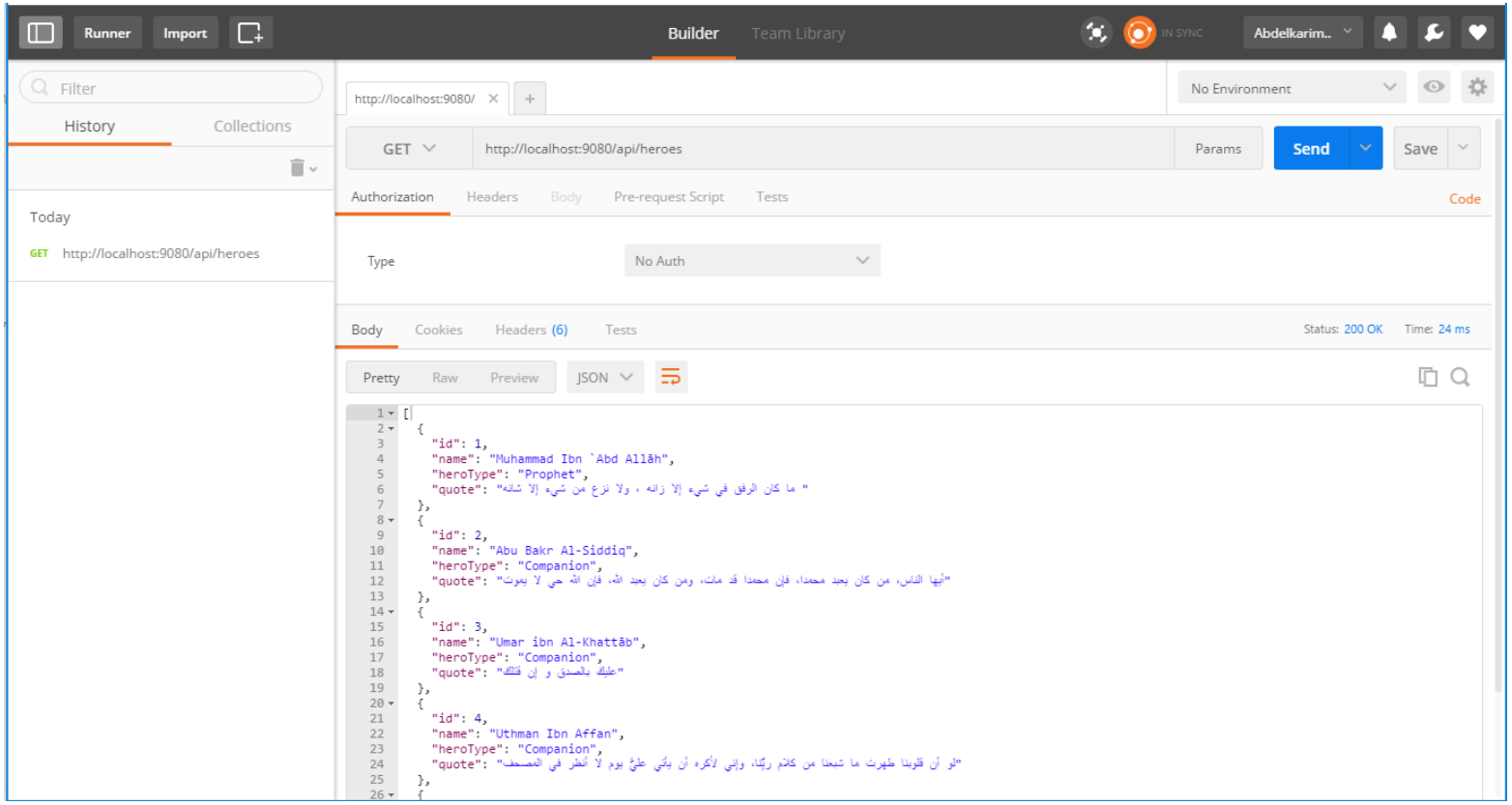
HTTP Verbs

- Represent the actions to be performed on resources
- Retrieve a representation of a resource: **GET**
- Create a new resource:
  - Use **POST** when the server decides the new resource URI
    - Post is not repeatable
  - Use **PUT** when the client decides the new resource URI
    - Put is repeatable
- **PUT** is typically used for update
- Delete an existing resource: **DELETE**
- Get metadata about an existing resource: **HEAD**
- See which of the verbs the resource understands: **OPTIONS**

REST Services using Node.js

- See posted MVC App Example and test the services using Postman Chrome plugin

https://www.getpostman.com/



Views

Template using Handlebars

http://handlebarsjs.com/

View Template

- **View template** used to dynamically generate HTML pages on-demand based on user input
- **View engine** (template engine) is a library that generates HTML page based on a template and a given JavaScript object
  - Provide cleaner solution by separating the view
- There are lots of JavaScript view engines such as Handlebars.js, KendoUI, Jade, Angular, etc.
- **Handlebars.js** is recommended. It is a library for creating client-side or server-side UI templates

Usage

- Add Handlebars script

```
<script src="path/to/handlebars.js"></script>
```

- Create a template

```
let studentTemplate =`
    <table>
        <tr>
            <td>StudentId</td>
            <td>{{id}}</td>
        </tr>
        <tr>
            <td>Name</td>
            <td>{{firstname}} {{lastname}}</td>
        </tr>
    </table>`
```

Template variable to be replaced with a value that is passed to the template

- Render the template

```
let student = {id: '…', firstname: '…', lastname: '…'},
    htmlTemplate = Handlebars.compile(studentTemplate)
studentDetails.innerHTML = htmlTemplate(student)
```

Creating HTML Templates

- HTML template has **placeholders** that will be replace by **data** passed to the template
- Handlebars.js marks placeholders with double curly brackets `{{value}}`
  - When rendered, the placeholders between the curly brackets are replaced with the corresponding value

Iterating over a list of elements

- **{{#list}} {{/list}}** block expression is used to iterate over a list of objects
  - Everything in between will be evaluated for each object in the collection

```
<select id="studentsDD">
    <option value=""></option>
    {{#students}}
        <option value="{{studentId}}">
              {{studentId}} - {{firstname}} {{lastname}}
        </option>
     {{/students}}
</select>
```

```
let students = [{
    "studentId": 2015001,
    "firstname": "Fn1",
    "lastname": "Ln1"
  },
  {
    "studentId": 2015002,
    "firstname": "Fn2",
    "lastname": "Ln2"
  }]
```

Conditional Expressions

- Render fragment only if a property is true
  - Using `{{#if property}} {{/if}}`
    or `{{unless property}} {{/unless}}`

```
<div class="entry">
  {{#if author}}
    <h1>{{firstName}} {{lastName}}</h1>
  {{else}}
    <h1>Unknown Author</h1>
  {{/if}}
</div>
```

```
<div class="entry">
  {{#unless license}}
  <h3 class="warning">WARNING: This entry does not have a license!</h3>
  {{/unless}}
</div>
```

The with Block Helper

- `{{#with obj}} {{/with}}`
  - Used to minify the path
  - Write `{{prop}}` Instead of `{{obj.prop}}`

```
<div class="entry">
  <h1>{{title}}</h1>

  {{#with author}}
  <h2>By {{firstName}} {{lastName}}</h2>
  {{/with}}
</div>
```

```
{
  title: "My first post!",
  author: {
    firstName: "Abbas",
    lastName: "Ibn Farnas"
  }
}
```

Server-side Rendering of Views

Client-side vs. Server-side Rendering of Views

- Client-side Views Rendering **frees the server** from this burden and enhances scalability
  - But one of the main disadvantages is slower initial loading speed as the client receive a lot of JavaScript files to handle views rendering
- Views could be generated on the server side to reduce the amount of client-side JavaScript and speed-up initial page loads particularly for slow clients but this puts the rending burden on the server
  - Web servers may render the page faster than a client side rendering. As a result, the initial loading is quicker.

Configure Handlebars View Engine

```
let handlebars    =  require('express-handlebars')
let app            = express()

/* Configure handlebars:
 set extension to .hbs so handlebars knows what to look for
 set the defaultLayout to 'main'
 the main.hbs defines define page elements such as the menu
 and imports all the common css and javascript files
*/
app.engine('hbs', handlebars({ defaultLayout: 'main',
  extname: '.hbs'}))

// Register handlebars as our view engine as the view engine
app.set('view engine', 'hbs')

//Set the location of the view templates
app.set('views', __dirname + '/views')
```

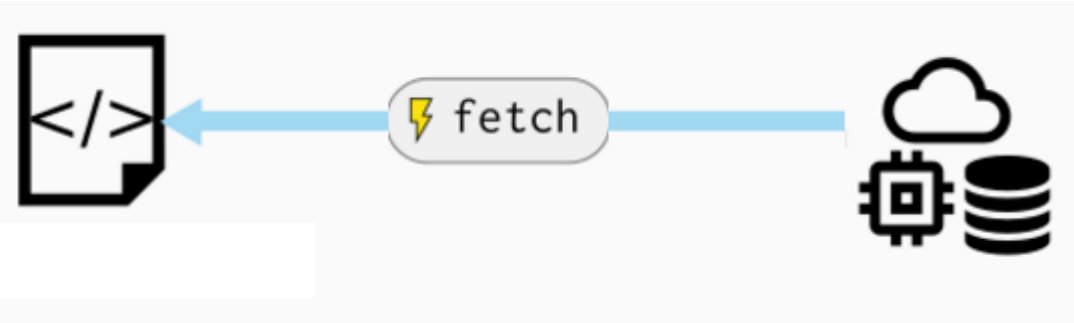
res.render

- Call **res.render** method to perform server-side rendering and return the generated html to the client

```
res.render('shopCart', { shoppingCart })
```

The above example passes the shopping cart to the **'shopCart'** template to generate the html to be returned to the browser

Communicating with the server using Fetch API



- AJAX is acronym of Asynchronous JavaScript and XML
  - AJAX is used for **asynchronously** loading (in the background) of dynamic Web content and data from the Web server into a HTML page
  - Allows dynamically adding elements into the DOM
- Two styles of AJAX
  - Partial page rendering
    - Load an HTML fragment and display it in a **`<div>`**
  - Call REST service then use the received JSON object to update the page at the client-side using JavaScript / jQuery

Call REST Service using Fetch API

- Fetch content from the server

```
async function getStudent(studentId) {
    let url = `/api/students/${studentId}`
    let response = await fetch(url)
    return await response.json()
}
```

- **.json()** method is used to get the response body as a JSON object

Posting a request to the server using Fetch API

- Fetch could be used to post a request to the server

```
let email = document.querySelector( "#email" ).value,
  password = document.querySelector("#password").value

fetch( "/login", {
    method: "post",
    headers: { "Accept": "application/json",
               "Content-Type": "application/json" },
    body: JSON.stringify({
        email,
        password
    })
})
//headers parameter is optional
```

Resources

- NodeSchool

https://nodeschool.io/

- Mozilla Developer Network

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs

- Learn Handlebars in 10 Minutes
  http://tutorialzine.com/2015/01/learn-handlebars-in-10-minutes/

- JavaScript Standard Style

https://github.com/feross/standard