COMP4021
Internet Computing

A SPA Example

Gibson Lam

A Basic SPA Example

- In this presentation, we will take a look at a basic single-page application example
- It is a web page for managing students

Students Manager

Student Id	Name	Year of Study	Action	
20000000	Chan Mei Ho	3	Change	Delete
20000001	Chan Tai Man	3	Change	Delete
20000002	Kwan Siu Lai	4	Change	Delete
20000003	Poon Lik Hang	4	Change	Delete
Enter Student Id	Enter Name	Enter Year of Study	+	

Files in the Example

- The example has three files:

  - `manager.html`
    The frontend HTML page of the
    application

    Browser

- -

  - `server.js`
    The server program that contains the
    web server of the application

    Server

  - `students.json`
    The JSON data of the students
    residing on the server

System Overview

Server

Browser

<html>

manager.html

The server serves this HTML file

Node.js

server.js

<html>

Browser Request

<html>

<script>

JSON

students.json

The server updates this JSON file

JSON

AJAX Requests
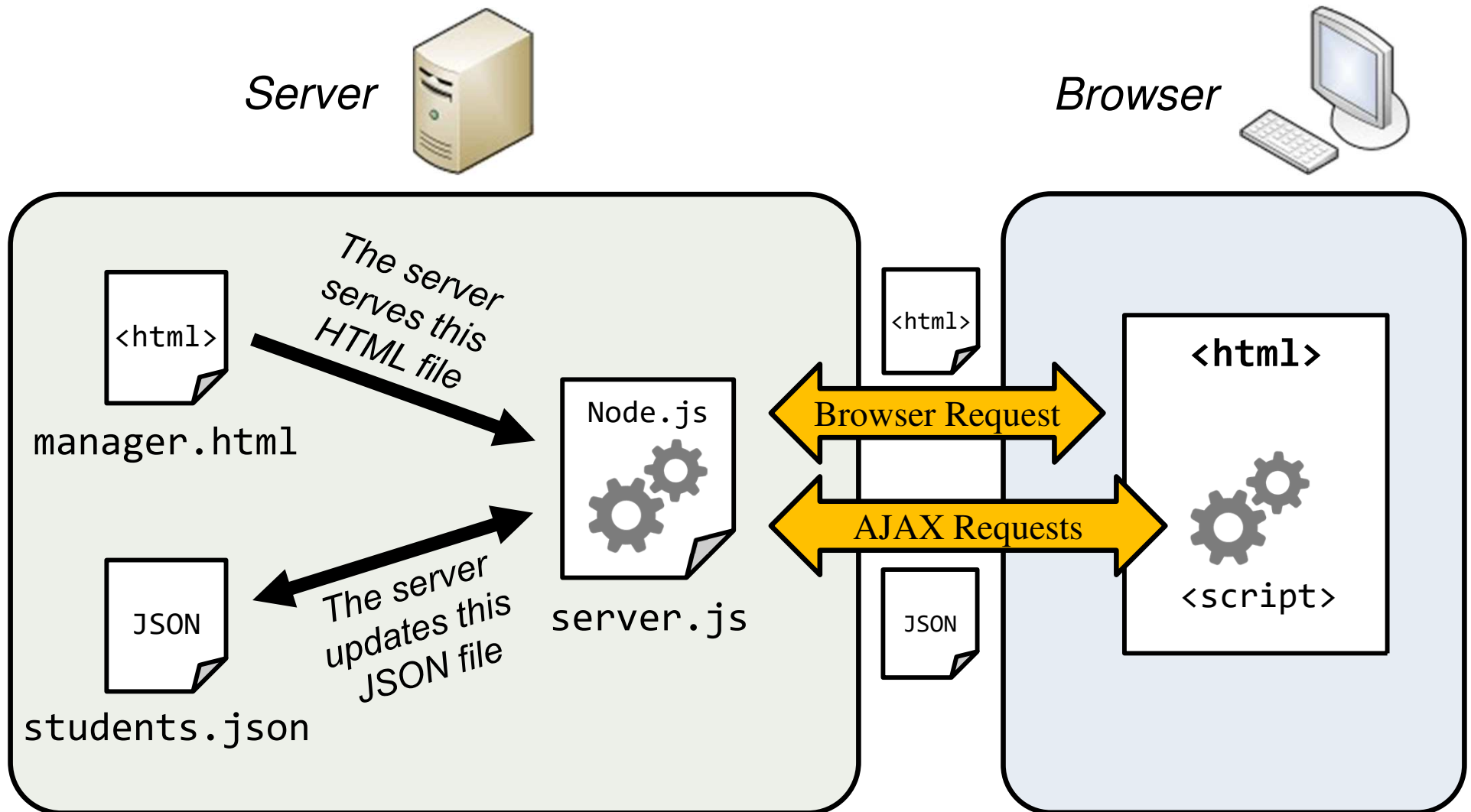
The HTML Page

- The HTML page has a table that shows the current students in the system

Students Manager

Student Id	Name	Year of Study	Action	
20000000	Chan Mei Ho	3	Change	Delete
20000001	Chan Tai Man	3	Change	Delete
20000002	Kwan Siu Lai	4	Change	Delete
20000003	Poon Lik Hang	4	Change	Delete
Enter Student Id	Enter Name	Enter Year of Study	+	

- You can change or delete individual students, or add new ones

- All of these are accomplished by sending AJAX requests to the server, without ever leaving the page

Using the Page

Students Manager

Student Id	Name	Year of Study	Action
20000000	Chan Mei Ho	3	Change · Delete
20000001	Chan Tai Man	3	Change · Delete
20000002	Kwan Siu Lai	4	Change · Delete
20000003	Poon Lik Hang	4	Change · Delete
Enter Student Id	Enter Name	Enter Year of Study	+

Add a new student by filling in the information and then click on '+'

Change the information and click on 'Change' or delete a student by clicking on 'Delete'

Sending AJAX Requests

- All requests are sent using `fetch()`

- For example, to get the current students, this code is used:

The server path to get the students
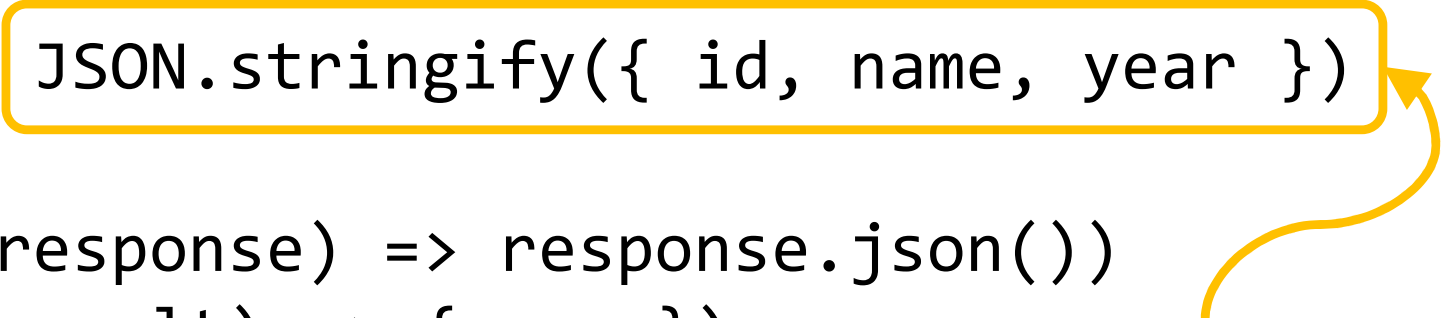
```
fetch("/students/all")
    .then((response) => response.json())
    .then((students) => {
        ...Add the students
        in the table...
    });
```

GET and POST Requests

- The previous example uses a GET request
- The other requests are POST requests as JSON data are sent to the server, e.g.:

```
fetch("/students/add", {
    method: "POST",
    headers: {"Content-type": "application/json"},
    body: JSON.stringify({ id, name, year })
})
    .then((response) => response.json())
    .then((result) => { ... });
```

JSON data is sent to the server in the 'add student' request

Updating the Student Table

- The HTML page makes requests to add, change or delete students

- Whenever changes are made, the HTML page updates the entire student table, i.e.
`<table>`…`</table>`

All done in JavaScript

```
Loading and
Showing the
HTML page
```

```
Waiting for
Actions
```

```
Updating
Student Table
```

```
Add/
Change/
Delete
Students
```

```
fetch()
```
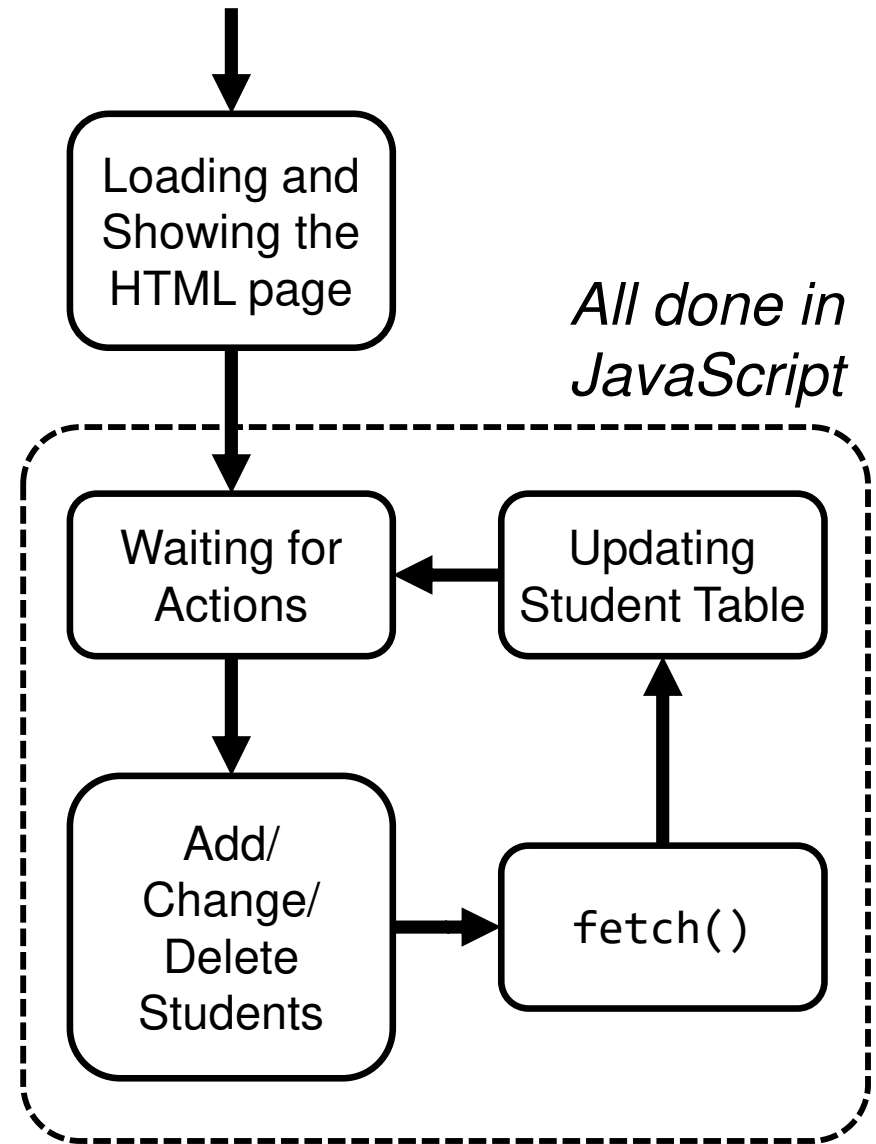
Adding HTML Using jQuery

- jQuery allows you to quickly add HTML by doing something like this:

$(...*Parent*...).append($("...*HTML*..."));

- For example, you can append a header to the end of the body using this code:

```html
<html>
  ▶ <head>…</head>
    <body></body>
</html>
```

➡

```html
<html>
  ▶ <head>…</head>
  ▼ <body>
      <h1>Hi!</h1>
    </body>
</html>
```

$("body").append($("<h1>Hi!</h1>"));

Inserting a Student

20000000   Chan Mei Ho   3   Change | Delete

- The following code inserts a student into the table body:

```html
▼<tr>
   <td>20000000</td>
  ▼<td>
     <input id="name-20000000" value="Chan Mei Ho">
   </td>
  ▼<td>
     <input id="year-20000000" value="3">
   </td>
  ▼<td>
     <button class="change" data-id="20000000">Change</button>
     <span> </span>
     <button class="delete" data-id="20000000">Delete</button>
   </td>
 </tr>
```

An example in Chrome

```javascript
const student = $("<tr></tr>")
    .append($("<td>" + id + "</td>"))
    .append($("<td><input id='name-" + id + "' value='" +
            name + "'></td>"))
    .append($("<td><input id='year-" + id + "' value='" +
            year + "'></td>"))
    .append($("<td></td>")
        .append($("<button class='change' data-id='" + id +
                "'>Change</button>"))
        .append($("<span> </span>"))
        .append($("<button class='delete' data-id='" + id +
                "'>Delete</button>"))
    );
$("#table-body").append(student);
```

append() can be chained for jQuery objects

The Server Endpoints

- Here are the specifications of the server endpoints, i.e. available access paths in the server:

Method	Path	Request Body	Response Body

GET	/students/all	-	JSON of students, e.g.
			`{`
			`    id: {name, year},`
			`    ...`
			`}`

POST	/students/add	`{id, name, year}`	`{ success: true }`, or `{ error: "..." }`
POST	/students/change	`{id, name, year}`	`{ success: true }`, or `{ error: "..." }`
POST	/students/delete	`{id}`	`{ success: true }`

The Students JSON File

- The JSON file `students.js` is the 'database' of the application

- It stores the students in simple JSON format, as shown on the right:

```
{
    "20000000": {
        "name": "Chan Mei Ho",
        "year": 3
    },
    "20000001": {
        "name": "Chan Tai Man",
        "year": 3
    },
    ...More students here...
}
```

Using the File System Module

- On the server program, the JSON file is read every time a request is made

- If there are changes to the students, the data will be written back the file

- The `fs` (file system) module is needed to read and write files, which is imported by:

```
const fs = require("fs");
```

Reading and Writing Files

- The application uses `fs.readFileSync()` and `fs.writeFileSync()` to read and write files respectively

- There are also asynchronous, promise version of the two commands

- In this example, we use the synchronous functions to keep things simple but they may block the server execution if the JSON file is very large

Dealing With the JSON Files

- Here is the code to read and write the JSON file:
  - Reading

```
const students =
    JSON.parse(fs.readFileSync("students.json"));
```

Convert to a JavaScript object before using

  - Writing

```
fs.writeFileSync("students.json",
    JSON.stringify(students, null, "   " ));
```

Add spacing to the JSON content

Convert to JSON content before writing

The Get-Students Endpoint

- The code for the get-students endpoint does two things:

  1. Reading the JSON file

  2. Returning the `students` object as JSON data

```
app.get("/students/all", (req, res) => {
  const students =
    JSON.parse(fs.readFileSync("students.json"));

  res.json(students);
});
```

①
②

The Add-Student Endpoint

- The add-student endpoint does a bit more, which includes:

  1. Reading the JSON body

  2. Checking for duplicate id

  3. Adding the student and writing the JSON file

```
app.post("/students/add", (req, res) => {
①  const { id, name, year } = req.body;

    const students = JSON.parse(...);

②  if (students[id]) {
      res.json({ error: "..." });
      return;
    }

③  students[id] = { name, year };
    fs.writeFileSync(...);

    res.json({ success: true });
  });
```

Return an error message

The Change/Delete Endpoints

- The change-student and delete-student endpoints are almost the same as the add-student endpoint

- The delete-student endpoint uses the `delete` keyword to delete a student in the `students` object, i.e.:

```
delete students[id];
```

where `id` is a key in the JavaScript object

Summary

- This example demonstrates the use of AJAX `fetch()` and a Node.js server to build a basic single-page application

- It does not have strict data validation so you may be able to break the system using some weird input values

- Nevertheless, this will be a good starting point for the work that you will do later in the server-side labs