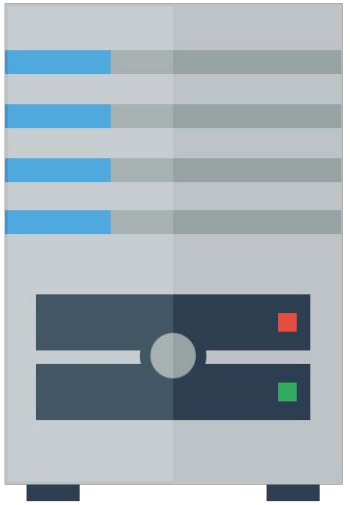
Serving static files

Dynamic content: Web services

Sometimes when you type a URL into your browser, the URL represents **an API endpoint**.



That is, the URL represents **a parameterized request**, and the web server dynamically generates a response to that request.

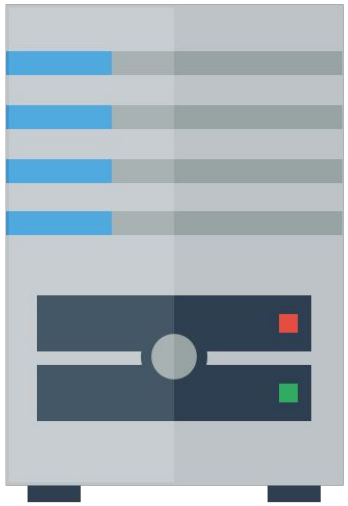
That's how our NodeJS server treats routes defined like this:

```
app.get('/hello', function (req, res) {
  res.send('GET hello!');
});
```

Static content: File servers

Other times when you type a URL in your browser, the URL is a **path to a file** on the hard drive of the server (html, css , js, jpeg, mp3 ….) :

- The web server software grabs that file from the server's local file system, and sends back its contents to you



We can make our NodeJS server also sometimes serve files "statically," meaning instead of treating **all** URLs as API endpoints, some URLs will be treated as file paths.

Statically served files

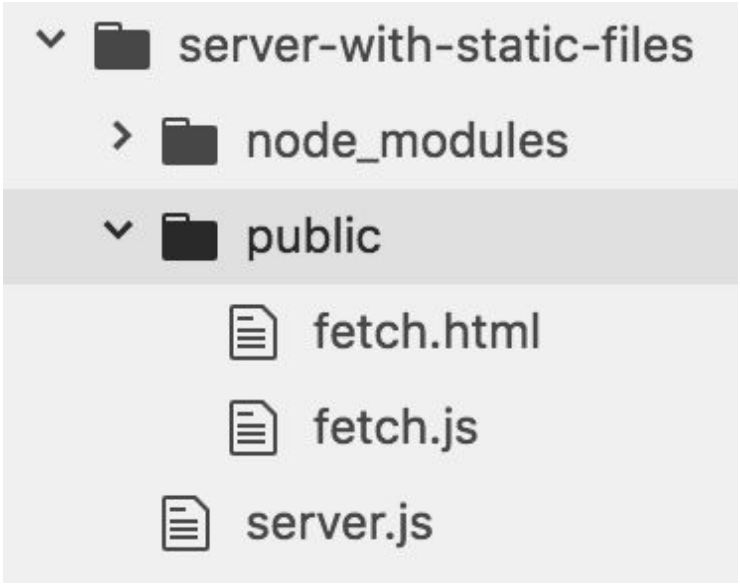
```
const express = require('express');
const app = express();

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

app.get('/', function (req, res) {
  res.send('Main page!');
});
```

This line of code makes our server now start serving the files in the 'public' directory directly.

Server static data




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

Now Express will serve:

http://localhost:3000/fetch.html

http://localhost:3000/fetch.js

Express looks up the files relative to the static directory, so the name of the static directory ("public" in this case) is not part of the URL (GitHub)

Client-side : Querying our server

HTTP requests

Our server is written to respond to HTTP requests (GitHub):

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

Q: How do we sent HTTP requests to our server?

Querying our server

Here are three ways to send HTTP requests to our server:

1. Navigate to http://localhost:3000/<path> in our browser

2. Call `fetch()` in a script

3. `curl` command-line tool
    a. Debug tool we haven't seen yet

Querying our server

Important precision :

Querying our server means that we will make an HTTP request to it. This will be the case when we want to receive content, like asking for the temperatures for the day. But it will also be the case when we want to send content, like sending the content of a form to add a user in an address book app.
Remember : **when you are sending data to the server, you're also making a HTTP Request.**

Querying with curl

`curl`: Command-line tool to send and receive data from a server (Manual)

`curl --request` ***METHOD url***

e.g.

```
$ curl --request POST http://localhost:3000/hello
```

Querying with `fetch()`

```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('http://localhost:3000/')
    .then(onResponse)
    .then(onTextReady);
```

Sending data to the server

Different parts of a url. (SCHEME = METHOD)

3 different ways to send data to the server

Here are three ways to send data to our server:

1. in the PATH part of the url as “route parameters”
   limit: 2048 characters max in a URL

2. in the QUERY STRING part of the url as key:value pairs
   limit : 2048 characters max in a URL

3. in the body part of a POST request
   no limit in HTTP specs. Chrome has a 4GB limit (4Go).

3 different ways to send data to the server

For each one of this option, you need to know how to send them from the client using Fetch(), and how to read them on the Server.

Route parameters

If you use 3rd party API, we saw a few ways to send information to the server via our `fetch()` request.

Example: Spotify Album API

`https://api.spotify.com/v1/albums/`**`7aDBFWp72Pz4NZEtVBANi9`**

- The last part of the URL is a **parameter** representing the album id, `7aDBFWp72Pz4NZEtVBANi9`

A parameter defined in the URL of the request is often called a "**route parameter**."

Route parameters

Q: How do we read route parameters in our server?

A: We can use the **`:variableName`** syntax in the path to specify a route parameter (Express docs):

```
app.get('/hello/:name', function (req, res) {
  const routeParams = req.params;
  const name = routeParams.name;
  res.send('GET: Hello, ' + name);
});
```

We can access the route parameters via **`req.params`**.

Route parameters

```
app.get('/hello/:name', function (req, res) {
  const routeParams = req.params;
  const name = routeParams.name;
  res.send('GET: Hello, ' + name);
});
```

GitHub

Route parameters

You can define multiple route parameters in a URL (docs):

```
app.get('/flights/:from-:to', function (req, res) {
  const routeParams = req.params;
  const from = routeParams.from;
  const to = routeParams.to;
  res.send('GET: Flights from ' + from + ' to ' + to);
});
```

GitHub

Query parameters

The Spotify Search API was formed a little differently:

Example: Spotify Search API

`https://api.spotify.com/v1/search?type=album&q=beyonce`

- There were two query parameters sent to the Spotify search endpoint:
  - `type`, whose value is `album`
  - `q`, whose value is `beyonce`

Query parameters with GET

Q: How do we read query parameters in our server?

A: We can access query parameters via **`req.query`**:

```
app.get('/hello', function (req, res) {
  const queryParams = req.query;
  const name = queryParams.name;
  res.send('GET: Hello, ' + name);
});
```

GitHub

Query params with POST

You can send query parameters via POST as well:

```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('/hello?name=Victoria', { method: 'POST' })
    .then(onResponse)
    .then(onTextReady);
```

(WARNING: We will not be making POST requests like this!

We will be sending data in the body of the request instead of via query params.)

Query params with POST

These parameters are accessed the same way:

```
app.post('/hello', function (req, res) {
  const queryParams = req.query;
  const name = queryParams.name;
  res.send('POST: Hello, ' + name);
});
```

GitHub

(WARNING: We will not be making POST requests like this!

We will be sending data in the body of the request instead of via query params.)

POST message body

However, generally it is poor style to send data via query parameters in a POST request.

Instead, you should specify a message body in your `fetch()` call:

```
const message = {
  name: 'Victoria',
  email: 'vrk@stanford.edu'
};
const serializedMessage = JSON.stringify(message);
fetch('/helloemail', { method: 'POST', body: serializedMessage })
    .then(onResponse)
    .then(onTextReady);
```

POST message body

Handling the message body in NodeJS/Express is a little messy (GitHub):

```
app.post('/helloemail', function (req, res) {
  let data = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk) {
     data += chunk;
  });

  req.on('end', function() {
    const body = JSON.parse(data);
    const name = body.name;
    const email = body.email;
    res.send('POST: Name: ' + name + ', email: ' + email);
  });
});
```

body-parser

We can use the body-parser library to help:

```
const bodyParser = require('body-parser');
```

This is not a NodeJS API library, so we need to install it:

```
$ npm install body-parser
```

body-parser

We can use the body-parser library to help:

```
const bodyParser = require('body-parser');

const jsonParser = bodyParser.json();
```

This creates a JSON parser stored in `jsonParser`, which we can then pass to routes whose message bodies we want parsed as JSON.

POST message body

Now instead of this code:

```
app.post('/helloemail', function (req, res) {
  let data = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk) {
     data += chunk;
  });

  req.on('end', function() {
    const body = JSON.parse(data);
    const name = body.name;
    const email = body.email;
    res.send('POST: Name: ' + name + ', email: ' + email);
  });
});
```

POST message body

We can access the message body through `req.body`:

```
app.post('/helloparsed', jsonParser, function (req, res) {
  const body = req.body;
  const name = body.name;
  const email = body.email;
  res.send('POST: Name: ' + name + ', email: ' + email);
});
```

GitHub

POST message body

We can access the message body through `req.body`:

```
app.post('/helloparsed', jsonParser, function (req, res) {
  const body = req.body;
  const name = body.name;
  const email = body.email;
  res.send('POST: Name: ' + name + ', email: ' + email);
});
```

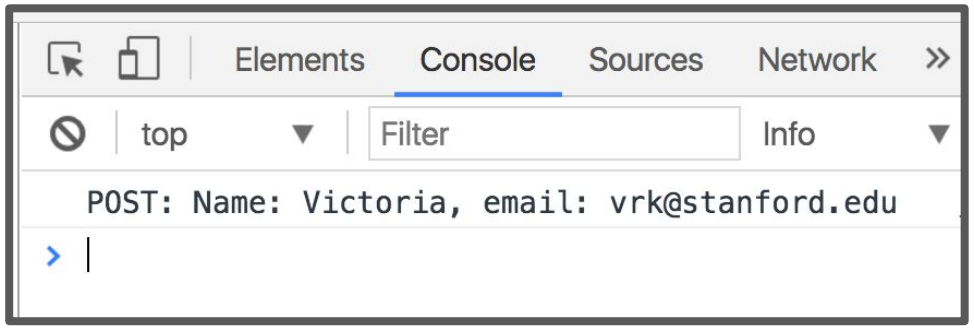
GitHub

Note that we also had to add the `jsonParser` as a parameter when defining this route.

POST message body

Finally, we need to add JSON content-type headers on the `fetch()`-side (GitHub):

```
const message = {
  name: 'Victoria',
  email: 'vrk@stanford.edu'
};
const fetchOptions = {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(message)
};
fetch('/helloparsed', fetchOptions)
  .then(onResponse)
  .then(onTextReady);
```



Recap

You can deliver parameterized information to the server in the following ways:

1. Route parameters
2. GET request with query parameters
   (DISCOURAGED: POST with query parameters)
3. POST request with message body

Q: When do you use route parameters vs query parameters vs message body?

GET vs POST

- Use GET requests for retrieving data, not writing data

- Use POST requests for writing data, not retrieving data
  You can also use more specific HTTP methods:
  - PATCH: Updates the specified resource
  - DELETE: Deletes the specified resource

There's nothing technically preventing you from breaking these rules, but you should use the HTTP methods for their intended purpose.

Route params vs Query params

Generally follow these rules:

- Use **route parameters** for required parameters for the request
- Use **query parameters** for:
  - Optional parameters
  - Parameters whose values can have spaces

These are conventions and are not technically enforced, nor are they followed by every REST API.

Example: Spotify API

The Spotify API mostly followed these conventions:

https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9

- The Album ID is required and it is a route parameter.

https://api.spotify.com/v1/search?type=album&q=the%20weeknd&limit=10

- q is required but might have spaces, so it is a query parameter
- limit is optional and is a query parameter
- type is required but is a query parameter (breaks convention)

Notice both searches are GET requests, too

package.json

Installing dependencies

In our examples, we had to install the express and body-parser npm packages.

```
$ npm install express
$ npm install body-parser
```

These get written to the `node_modules` directory.

Uploading server code

When you upload NodeJS code to a GitHub repository (or any code repository), you should **not** upload the `node_modules` directory:

- You shouldn't be modifying code in the `node_modules` directory, so there's no reason to have it under version control
- This will also increase your repo size significantly

Q: But if you don't upload the node_modules directory to your code repository, how will anyone know what libraries they need to install?

Managing dependencies

If we don't include the `node_modules` directory in our repository, we need to somehow tell other people what `npm` modules they need to install.

`npm` provides a mechanism for this: `package.json`

package.json

You can put a file named `package.json` in the root directory of your NodeJS project to specify metadata about your project.

Create a `package.json` file using the following command:

```
$ npm init
```

This will ask you a series of questions then generate a `package.json` file based on your answers.

Auto-generated package.json

```
{
  "name": "fetch-to-server",
  "version": "1.0.0",
  "description": "Example of fetching to a server",
  "main": "server.js",
  "dependencies": {
    "body-parser": "^1.17.1",
    "express": "^4.15.2"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "Victoria Kirst",
  "license": "ISC"
}
```

GitHub

Saving deps to package.json

Now when you install packages, you should pass in the --save parameter:

```
$ npm install --save express
$ npm install --save body-parser
```

This will also add an entry for this library in package.json.

```
"dependencies": {
  "body-parser": "^1.17.1",
  "express": "^4.15.2"
},
```

Saving deps to package.json

If you remove the node_modules directory:

```
$ rm -rf node_modules
```

You can install your project dependencies again via:

```
$ npm install
```

- This also allows people who have downloaded your code from GitHub to install all your dependencies with one command instead of having to install all dependencies individually.

npm scripts

Your package.json file also defines scripts:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node server.js"
},
```

You can run these scripts using `$ npm` ***scriptName***

E.g. the following command runs "node server.js"

```
$ npm start
```