Creating our Express Webserver

- We have a package
- Express framework is installed and available
- We start by creating our main server file
    - `server.js`
    - Nothing special/magical about the name

First Attempt

```
const express = require('express'); // No path = library
const app = express(); // how express works

app.listen(3000, () => console.log('server running'));
```

- `3000` is the **port number** we are listening to
  - Nothing special about 3000
  - Ports < 1024 may require admin permissions
  - 80 (HTTP) / 443 (HTTPS) for "real" web
  - "Dev" ports often for development
  - Common: 3000, 4000, 5000, 8000, 8080, 8888
- Can visit `http://localhost:3000/`

Well that sucks

```
Cannot GET /
```

- Actually this is fine

What does this tell us?

- Response Received!
    - 404, but that's a full response
- `Cannot GET /` is specific
    - Browser request was `GET /`
    - Server doesn't know how to respond

Static Responses

- Create `public` folder
    - Will hold **static assets**
    - `public` will be our **document root**
        - Base of "reachable" material
- Create `public/index.html`
    - With whatever HTML

Edit and restart `server.js`:

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

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

Document Root and Static Assets

- `public/` is our **document root**
- No `public/` in URLs

Example:

```
public/
public/index.html
public/css/styles.css
```

✅ `<link rel="stylesheet" href="/css/styles.css">`

✅ `<link ... href="css/styles.css">`

❌ `<link ... href="public/css/styles.css">`

app.use(express.static(DOCUMENT_ROOT));

Ex: `app.use(express.static('./public'));`

- Defines directory to use as **document root**
- Will try to match paths and files from requests
    - To paths and files in that directory
- Example is simple case
    - Assumes this operating system
    - Assumes starting in certain directory

```
const path = require('path'); // 'path' lib ships with Node
app.use(express.static(path.join(__dirname, 'public')));`
// __dirname is built-in special variable
```

Creating Dynamic Routes

- Express works as a loop, started by `app.listen()`
- Each request tries to match stack of **middleware**
    - Request matches **route** (**method** and **path**)
    - Middleware **handler** can end or pass to next
- `app.use()` - all methods, defaults to all paths
- `app.get()` - `GET` method, requires a path
- `app.post()` - `POST` method, requires a path

```
app.get('/', (req, res) => {
    res.send("Qapla'!");
});
```

I'm sorry, what was that?

When a request matches **method** and **path**

- The **route handler callback** is called
    - A function we define and provide
- Route handler is called with:
    - **request** object (commonly `req`)
        - `req` gives data about the request
    - **response** object (commonly `res`)
        - `res` represents not-yet-sent response

Why do we still see static page?

- Express uses **Chain of Responsibility** pattern
    - Request checks each middleware in turn
    - `app.use(express.static` BEFORE `app.get('/'`
    - Our route handler is never given a chance

You fixed that but *still* seeing the static page?

- Did you save your changes?
- Did you remember to restart the server?
    - Changes in public: no restart required
    - Changes in server.js: restart required
- Changing a diff copy of the files than is running?

Server State

"**State**" is an important term in coding

- Represents the possible changing values
- "state" of a sports event:
    - Names of players/teams
    - Scores
    - Current half/quarter/inning/period
    - Time remaining
    - etc
- All UI and interaction based on **state**

Simple Server State Example Setup

```
const express = require('express');
const app = express();
const PORT = 3000; // A constant!

const isTabbyCat =  {
  Jorts: false,
  Maru: true,
};

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

app.get('/tabbies', (req, res) => {
    // Do stuff here
});

app.listen(
  PORT,
  () => console.log(`http://localhost:${PORT}`),
);
```

Simple Server Example Using State

```
app.get('/tabbies', (req, res) => {
  const tabbyList = Object.keys(isTabbyCat).map( cat => {
    return `
      <li>
        ${cat}
        ${isTabbyCat[cat] ? "Is" : "Is NOT"}
        a tabby.
      </li>
    `;
  }).join('');
  res.send(`
<!DOCTYPE html>
<html lang="en">
<head> <title>Known Cats</title> </head>
<body>
  <ul class="tabbies">${tabbyList}</ul>
</body>
</html>
  `);
});
```

Works, but getting messy

All mixed together:

- HTML code
- Data JS code
- General Server Config JS code
- Route-specific JS code

Separation of Concerns

- General computing approach
- **Separate** the many (many) pieces
    - Into related collections
    - For their common purpose (**concern**)
- Groups interact only through defined ways
    - Allows for easier change
    - If outside code doesn't use a part
        - It can't break when you change that part

Model-View-Controller (MVC)

- **MVC** is a specific way to implement SOC
  - Many specific variations and details
  - I will only focus on the larger concept
- **Model**
  - Your data and how to change it
- **View**
  - Your presentation
  - Must be given the data (model)
- **Controller**
  - Connects other pieces

Model - Data and how to change it

- No changes yet
- Let's move to another file for better separation

```
// cats.js
const isTabbyCat =  {
  Jorts: false,
  Maru: true,
};

module.exports = {
  isTabbyCat,
};
```

- Import using `const cats = require('./cats');`
  - Explicit Path! (`'./cats'` not `'cats'`)
- References to `isTabbyCat` now `cats.isTabbyCat`

View - Presentation, must be passed data

```
// views.js - Let's move this to another file too

function showTabbyList(isTabbyCat) {
  const tabbyList = Object.keys(isTabbyCat).map( cat => {
    return `
      <li>
        ${cat} ${isTabbyCat[cat] ? "Is" : "Is NOT"} a tabby.
      </li>
    `;
  }).join('');

  return wrapInPageHtml(
    `<ul class="tabbies">${tabbyList}</ul>`
  );
}

function wrapInPageHtml( body ) {
    return `<html><etc/>${body}<etc/></html>`;
}
module.exports = {
  showTabbyList,
};
```

`server.js` after separating model and view

```
const express = require('express');
const cats = require('./cats');
const views = require('./views');
const app = express();
const PORT = 3000;

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

app.get('/tabbies', (req, res) => {
  res.send(views.showTabbyList(cats.isTabbyCat));
});

app.listen(
  PORT,
  () => console.log(`http://localhost:${PORT}`),
);
```

Sending Data using HTML Forms

- Let's start with writing a Search
- Can send data in request with **HTML Forms**
- HTML structure representing a form
    - Browser collects data
    - Sends in request

The `<form>` element

- `<form>` has two essential attributes
    - `method`
    - `action`
- `method` - `GET` (default) or `POST`
    - More later
    - For now, `GET` is typical for searches
        - Following links or typing URL is `GET`
- `action` - `URL` form request is sent to
    - Every web request is a URL

Which URL for `action`?

- Already have `GET` on `/tabbies` show ALL tabbies
    - Can say that is default
    - If specific search, show that instead
- We'll use `/tabbies` as our form `action`
    - If no `action`, uses current page URL
        - Should always have an explicit `action`

Form Element Content

```
<form action="/tabbies" method="GET">
????
</form>
```

- Any HTML (except another form) as content
- **form fields** and **buttons** have special interactions

Text Input Form Field

- `<input>` element defaults to `<input type="text">`
- Self-closing
- `name` attribute names the field when sending
- `value` attribute sets initial value
- `placeholder` provides hint
    - NOT a label!
    - Has accessibility issues!

Label element

- Every field should have a text label
- One of most common accessibility omissions
- Functional benefits (Ex: Click to select field)
- `<label>` associates text label with form field
- Form field can be in content
- `<label>` `for` attribute can be `id` of field

```
<label>
  Name: <input name="name"/>
</label>

<label for="age">Age</label>
<input id="age" name="years"/>
<!-- id and name can be same or different -->
```

Button

- `<button>A button</button>`
- In Form: defaults to `type="submit"`
    - "submits" form data to action URL
- Out of Form: Defaults to `type="button"`
    - Has no automatic effect
- Don't use other ways to make a button
    - Not `type="button"` for `<input>`
    - Not `type="reset"` for `<button>` or `<input>`

Focus

A field/button that is currently selected has **focus**

- Sometimes visible as (by default) blue outline
    - The `outline` CSS property
- **Never remove the focus outline**
    - ❌ `:focus { outline: none; }` - BAD!
    - Lots of older, bad examples do this
    - Horrible accessibility to do so
    - Users won't see it unless they should
        - Seen (needed) for keyboard navigation
        - `Tab` key cycles links, fields, and buttons

Confirming a Field Label

Course Requirement:

- **Properly associated labels for every field**
- Easy to make mistakes on
- Confirm by clicking label
    - Field should get focus
    - Checkbox will toggle checked state
- Easy to confirm if label is correct
    - No reason to not notice

Submitting a Form

```
<form action="/tabbies" method="GET" class="tabby-search">
  <label for="name" class="name-label">Cat Name</label>
  <input id="name" name="name" class="name"/>
  <button class="submit">Search</button>
</form>
```

URL: `/tabbies?name=Jorts`

- `GET` method submits form data in URL
- As **query**/**query parameters**
- Data is **URL-Encoded** (**percent encoding**)

URL Encoding

- Special chars become `%` + 2 ASCII Hex characters
    - Ex: `1+1=2%` becomes `1%2B1%3D2%25`
        - `+` became `%2B`
        - `=` became `%3D`
        - `%` became `%25`
- Space *usually* becomes `+`
    - Might be `%20`
- Fields send `name=value`
    - Ex: `name=Jorts` or `age=3`
- Multiple fields separated by `&`
    - Ex: `name=Jorts&age=3`

You mostly don't do URL Encoding

- Browser automatically encodes form data
- Server automatically decodes form/URL data
- This is the what and why of the funky text

Reading query data on server

Form data in the request URL query

- Available in `req.query` object
- key is field `name`
- Ex: `const name = req.query.name;`
- Ex: `const age = req.query.age;`
- Can destructure
    - Ex: `const { name, age } = req.query;`

Using `query` Data In Dynamic Response

```
app.get('/tabbies', (req, res) => {
  const name = req.query.name;

  if (!name) {
    res.send(views.showTabbyList(cats.isTabbyCat));
    return; // We want to end here
  }

  res.send(views.showTabbyList(
    { [name]: cats.isTabbyCat[name] } // obj w/single entry
  ));
});
```

Lots of issues

- Search random name
  - Shows as if real name
- Ugly/Confusing output
  - Unclear that was search result
- Search only
- Issue for this moment:
  - Code is getting messy again
  - Mixing route logic with route handling logic

Leave Server in charge of routes

```
// server.js
const express = require('express');
const controllers = require('./controllers');
const app = express();
const PORT = 3000;

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

app.get('/tabbies', controllers.showTabbies);

app.listen(
  PORT,
  () => console.log(`http://localhost:${PORT}`),
);
```

- Seems a small change
    - But **separates a concern**

Controller - Connecting Models and Views

```
// controllers.js
const cats = require('./cats');
const views = require('./views');

function showTabbies(req, res) {
  const name = req.query.name;

  if (!name) {
    res.send(views.showTabbyList(cats.isTabbyCat));
    return; // We want to end here
  }

  res.send(views.showTabbyList(
    { [name]: cats.isTabbyCat[name] } // obj w/single entry
  ));
}

module.exports = {
  showTabbies,
};
```

Controller - Giving Model More

```
// controllers.js
const cats = require('./cats');
const views = require('./views');

function showTabbies(req, res) {
  const name = req.query.name;

  if (!name) {
    res.send(views.showTabbyList(cats.isTabbyCat));
    return; // We want to end here
  }

  const match = cats.findIfTabby(name);
  res.send(views.showTabbyList(match));
}
```

Model - Updated Without Improvements

```
// Added to cats.js
function findIfTabby(name) {
  return { [name]: cats.isTabbyCat[name] };
}

module.exports = {
  showTabbyList,
  findIfTabby,
};
```

Benefits of Separated Concerns

- We can now make improvements
- Parts define **contract** of expected input/output
    - Ex: `views.showTabbyList()` expects an object
- Can change internals freely if contract unchanged
- When contract changes are appropriate
    - We have less to keep in mind
    - Ex: What to do if no matching cat?

What About Changing Data On Server?

- Changing Data done with `POST` method
- For fun, let's use same `action` URL

```
<form action="/tabbies" method="POST" class="tabby-update">
  <label for="name">Name</label>
  <input id="name" name="name"/>
  <label for="is-tabby">Is Tabby?</label>
  <input type="checkbox" id="is-tabby" name="isTabby"/>
  <button>Add Cat</button>
</form>
```

- `name="isTabby"`
    - No common casing for field names
    - Often based on backend language

POST is a different route

```
// server.js
const express = require('express');
const controllers = require('./controllers');
const app = express();
const PORT = 3000;

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

app.get('/tabbies', controllers.showTabbies);
app.post('/tabbies', controllers.updateTabby); // Incomplete

app.listen(
  PORT,
  () => console.log(`http://localhost:${PORT}`),
);
```

POST Puts Form Data In Request Body

- Not into query
- Not in URL
- `GET` requests do not have a body
- Data still defaults to URL-encoded
    - But in body
- We can see this in DevTools
- Checkbox sends value (default: 'on')
    - name not even sent if not checked

Request body data can be read in `req.body`

- But there's a complication
- Web requests can send ANY data in body
- URL encoding is just one way to send it
- Server has to decide how to parse the data

```
// Inside controllers.js

function updateTabby(req, res) {
  const { name, isTabby } = req.body;

  // req.body is undefined!
```

Tell Route or Server to use a certain parser

```
app.post(    // wrapped to fit small screen
  '/tabbies',
  express.urlencoded(),
  controllers.updateTabby
);
```

- `express.urlencoded()` - parses urlencoded body
- Works, but gives warning message (in 4.x, not 5.x)
    - Wants `extended` option explicitly set
    - `express.urlencoded({ extended: false })`
- Can also be used for all requests with such bodies

```
// Must be BEFORE any routes using request body data
app.use(express.urlencoded({ extended: false }));
```

Model Handles Changes to Data

```
// cats.js

function updateTabby(name, isTabby) {
  if (name) {
    isTabbyCat[name] = !!isTabby;
  // !! = bang bang, Converts to boolean from truthy/falsy
  }
}

module.exports = {
  isTabbyCat,
  findIfTabby,
  updateTabby,
};
```

- Why? Controller could just do this!
- Because the data structure is model's **concern**
- Controller just gives high level order

What does Controller send in response?

```
function updateTabby(req, res) {
  const { name, isTabby } = req.body;
  cats.updateTabby(name, isTabby);
  // ??? Now What?
}
```

- You don't want to send a web page response
- It would work
- But what happens if they "reload" the page?
    - Would send update again
    - "Harmless" in this case?
    - ...as long as only you are updating data(!)
        - That's not reliable on web

You can send a "redirect" response

- 3xx status code
- Sends `Location` header with redirect url
- Browser automatically loads that URL
- Any reload will load that new URL
    - Not the POST request changing data

```
function updateTabby(req, res) {
  const { name, isTabby } = req.body;
  cats.updateTabby(name, isTabby);
  res.redirect('/tabbies');
}
```

Server responds to follow-up request

Flow goes like this:

- `POST /form-action-url` request w/data
- Server updates data, sends redirect response
- Browser sends `GET /redirect-location`
- Server responds to that new request
    - Updated data used in response
- User hits "reload/refresh"
    - Browser requests `GET /redirect-location`

What was that? Someone else changing data?

- Server gets requests from many users
    - Each gets a response
    - All interact with same server data
- Same server data can be updated by others
- Or even yourself on a different device
- Web apps should keep that in mind
    - New devs often make false assumptions
- **"this request" only meaningful within handler**

Related: Any Web page could be out of date

- Server sends response and connection done
- Sent page not updated when server data changes
- Server doesn't know when tab closed or page left
- User may submit a form on page loaded last week

Related: Page is not only source of requests

- Anyone can send requests to server
- Doesn't have to come from your page
- Anyone can also edit copy of page in browser
    - *Most* won't, that's not none
- Never assume people follow your intended flow
- Any request can be made at any time
- More on Security soon

More HTML Form Elements

- Not exhaustive, check MDN

`<input>` has many `type` options

Just a few:

- `text` (default)
- `checkbox`
- `password`
    - Hides value visually
    - NOT encryption, sends as plain text
- `number`
    - Sends as text string
- `radio`
    - Multiple share `name`, max one selected
    - UI can't unselect

`<select>` "dropdown" field with `<option>`s

```
<label for="favorite">Your Favorite Cat</label>
<select id="favorite" name="favorite">
  <option value="">(Select One)></option>
  <option value="jorts">Jorts</option>
  <option value="jean">Jean</option>
  <option value="maru">Maru</option>
</select>
```

- Selected `<option>` value sent as `name`
- Notice "(Select One)" prompt is an actual option
- Always have explicit `value` for each `<option>`
- Always have an associated `<label>`

<textarea> allows for multiline text

```
<textarea name="comment">Content is default value
Can contain and accept multiple lines
</textarea>
```

- UI defaults to resizable with corner drag
    - Can change with CSS