Sending Data

You can do A LOT with just linked HTML documents

- Most news sites

More Fun: send user-entered data to webserver

- Save this data
- Use saved data to generate dynamic responses
- How Wikipedia gets new articles/updates

Most common method: **HTML Forms**

The `<form>` element

```
<form action="ACTION" method="METHOD">
  <button type="submit">TEXT</button>
</form>
```

- **ACTION** is url to submit to
    - Defaults to current page url!
    - Can be fully-qualified, path, etc
    - e.g. `/login`, `/logout`, `/add/more/stuff`
- **METHOD** is **HTTP Method**
    - `GET` or `POST` for html forms
- "Submit" tells browser to **navigate**
    - Sends **request** (incl data from form)
    - **Renders response as new page**

More about method and forms

- HTTP Method is sent in **web request**
  - `GET` or `POST` for html forms
  - `GET` should cause no changes to data
    - Request to read ("get") data from server
    - Sends form fields (if any) in url
      - **query params** in url
  - `POST` can change server data
    - Anything not a GET for HTML Form
    - Sends form data in **request body**
    - Can **hardcode** query params in action url

Elements inside a `<form>`

Form element can contain most HTML elements

- Only certain elements matter to form submission
- `<input>` is the most common
- `<input type="TYPE">`
    - TYPE defaults to **"text"**
    - see MDN for more

Sending Text with `<input>`

```
<form action="/lookup" method="GET">
  <input name="cat" placeholder="Cat Name">
  <button type="submit">Register Cat</button>
</form>
```

Input **type** defaulted to "text"

```
<input type="text" name="cat" placeholder="Cat Name">
```

On submit:

- Navigates to `/lookup?name=`
    - Followed by url-encoded cat name value
- Will get 404 if server not ready for `/lookup`

Query Parameters

In URL, a `?` separates **path** from **query parameters**

- Query parameters can be any string value
- Almost always html form style
    - `key=value` pairs (no spaces, no quotes)
    - Separated by `&` (when multiple params)
    - `key` and `value` are **url encoded**

URL Encoding

- Any "special" characters are replaced
    - `:/?$[]@!$&'()*+,;=%`
- `%` followed by their **hexadecimal ascii value**
    - Other characters MIGHT be converted
    - Spaces often converted to `+`
        - Or may be `%20`
- Allows us to give these characters meaning
    - Like `?`, `&`, `=`, `%`
    - But still allow the characters in text

URL Encoding example

```
<form action="/demo" method="GET">
  <input type="text" name="cat"
    value="Jorts=Still Unbuttered!"
   />
  <input type="text" name="dog" value="drooling"/>
  <button type="submit">Try it!</button>
</form>
```

Will navigate to

`/demo?cat=Jorts%3DStill+Unbuttered%21&dog=drooling`

- `3D` is the hex ascii value for `=`
- `21` is the hex ascii value for `!`

Mostly don't need to DO URL Encoding

- Browser automatically URL encodes form data
- Server automatically decodes HTML form data

You should know the idea of what is happening

- Don't need to recreate it yourself

Server gets data in request object

Route-handling **callback**

- Is passed **request** and **response objects**
- By convention `req` and `res` (yuck)

```
app.get('/lookup', (req, res) => {
  // handling code here
});
```

Notice the `.get()` to match GET method

- We match method AND path
- We don't match query params

Common Conventions

Verbose:

```
app.get('/lookup', function( request, response ) {
  // handling code here
});
```

Conventional:

```
app.get('/lookup', (req, res) => {
  // handling code here
});
```

You can review more about this in the `fat-arrow` document in `readings` in your repo

Reading the query params

```
app.get('/lookup', (req, res) => {
  console.log(req.query);
  res.send('received');
});
```

- You must `.send()` a response
  - Or browser waits until timeout
  - You can't change response once you `.send()`
- `console.log()` goes to server
  - This JS does not run on browser!
- `req.query` is an **object** of name:value pairs
  - One (common) way of parsing **query string**
- All values are text (strings)
  - Url can only send text

Dynamic response

```
const catStatus = {
  Jorts: 'in trash bin',
  Jean: 'opening closet',
  Nyan: 'flying high',
};

app.get('/lookup', (req, res) => {
  const { cat } = req.query; // Destructuring!
  // Same as:
  // const cat = req.query.cat;
  const activity = catStatus[cat] || 'cat not found';
  res.send(activity);
});
```

Notice you can change the URL in browser

- Don't need to go through web page
- Web is stateless, only request matters

Other Form Elements

More than `<input type="text">`

- Other `input` types
  - "`checkbox`", "`radio`", "`date`", and more
- Other elements entirely
  - `<select>`, `<textarea>`
- Elements that add detail
  - `<label>`, `<fieldset>`
- Won't cover everything
  - Use MDN

`<input type="checkbox">`

A clickable checkbox

- Tends to be small
- Good to surround with a `<label>` element
    - Clicking label counts as clicking input
    - Supply text label for field
- Only sends field when checked(!)
- Defaults to value `"on"`
- A `checked` attribute sets default checked status

Checkbox example

```
<form action="/register-for-spam">
    <label>
        <input type="checkbox" checked>
        Yes! Send me all your spam
    </label>
    <button type="submit">Submit</button>
</form>
```

`<input type="radio">`

- Multiple inputs can have same `name`
- Only one can be selected at a time
    - Defaults to one with `checked` attribute
    - Or none if no `checked` attribute
    - Can never unselect a choice
- You can have multiple radio groups
    - Same `name` will impact each other

Radio example

```
<form action="/register">
    <p>Favorite Pet</p>
    <label>
        Cat
        <input type="radio" name="pet" value="cat"/>
    </label>
    <label>
        Dog...just kidding, still Cat
        <input type="radio" name="pet" value="not-dog"/>
    </label>
    <button type="submit">Confirm</button>
</form>
```

`<input type="password">`

- Just like "text"
- Doesn't show the typed characters
- NOT ENCRYPTION
    - Only about being visible on screen
    - With a GET method, still shows in url
    - If not HTTPS, sent in plain text
        - Even with POST method
        - Like private data on the OUTSIDE of mailing envelope

Password Example

```
<form action="/login">
    <label>
        Secret Password:
        <input type="password" name="secret"/>
    </label>
    <button type="submit">Log In</button>
</form>
```

`<input type="hidden">`

- Like `type="text"`
- Except not shown to user
- Still sent to server

Why?

- Send info in a different way than shown to user
- Persist info across a series of requests

Later:

- Should never be "trusted"
- User can change

Hidden input example

```
<p>Should we trust you?<p>
<form action="/trust">
    <input type="hidden" name="isDog" value="yes"/>
    <button type="submit">Yes</button>
</form>
<form action="/trust">
    <input type="hidden" name="isDog" value="no"/>
    <button type="submit">No</button>
</form>
```

`<textarea>`

- Allows for multiple lines of text
- Some formatting/resizing options
- Otherwise like `<input type="text">`

Textarea Example

```
<form action="/dev/null">
    <label>
        Please describe your complaint in detail
        <textarea name="complaint">
        </textarea>
    </label>
    <button type="Submit">Submit Complaint</button>
</form>
```

Dropdowns

Two elements

- `<select>` wraps `<option>`s
- `<select>` takes `name`
- Each `<option>` has a `value`
    - Defaults to contents (DON'T DO THAT)
- One `<option>` can have `selected` attribute
    - Defaults to first option if no `selected`
- `<select>` has options for selecting multiple
    - Confuses users, avoid :(

Select/Option Example

```
<form action="/update-profile">
    <label>
        Favorite Animal
        <select name="favorite">
            <option value="cat">Cat</option>
            <option value="drool">Dog</option>
            <option value="fish">Fish</option>
            <option value="bird">Bird</option>
        </select>
    </label>
</form>
```

`<input type="file">`

All other elements input/send some form of text string

- `type="file"` is special
- Upload a file (image, document, whatever)
- Requires extra effort server-side
    - To read the file
        - Attackers can send huge/hostile data
    - Do we need to save it as visible to users?
- Can be complicated to make the form look good

More about `<label>`

- Should have `<label>` for each form field
- Can be parent of field
    - Parent of `<input>`, `<select>`, `<textarea>`
    - If so, no `for`/`id` needed (see below)
- Can be separate (sibling or other non-ancestor)
    - field element must have an `id` attribute
    - `<label>` must have `for` attribute with field id

```
<form action="/login">
    <label for="username">Username</label>
    <input id="username" type="text" name="username"/>
    <button type="submit">Log In</button>
</form>
```

Body Data

A `GET` request

- Sends all data in the URL
- Should not cause the server to change data
    - Searches, reads

A `POST` request (from HTML form)

- Sends data in the **body of the request**
- Can cause the server to change data
    - Updates, data entry, etc

Making a POST form

The HTML is identical

- Except `method="POST"` in the `<form>`
- Browser will url-encode and place in request **body**
    - In **body of request**, not in url query params

You can see the sent data in the Browser Dev Tools

HTML form is not the only option

HTML Forms are limited

- We will cover other forms of sending later
- HTML Forms still can send basically everything
    - Differences are in UI and navigation
    - More later

Reading data from a request body

Server doesn't know if you are using HTML Forms

- It can handle countless options
    - With the right libraries
- We translate the request before our handler
    - In the "middle" of incoming and next step
    - Known as **middleware**
        - Generic term, this is how Express does it

Express allows a "chain" of handlers

All requests regardless of method

- Converts the body of all requests
  - Using a `url-encoded` approach

```
app.use( express.urlencoded() );
```

Or a specific route

```
app.post( '/cats', express.urlencoded(), (req, res) => {
  // code here
});
```

Notice `app.post()` to match POST method on form!

Accessing the parsed body

`express.urlencoded()` gives warning

- Unless you pass `{ extended: false }`

Body fields are mapped to the `req.body` object

- If `req.body` is `undefined`
    - You probably forgot to parse the body w/middleware!

```
// middleware     vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
app.post('/cats', express.urlencoded({extended: false}), (req, res) => {
  // code here
});
```

Example POST Form

```
<form action="/cats" method="POST">
  <label>
    Name:
    <input name="name" placeholder="Cat Name">
  </label>
  <label>
    Is Tabby?
    <input type="checkbox" name="isTabby">
  </label>
  <button type="submit">Register Cat</button>
</form>
```

Making the form look good

Appearance is the job of CSS!

- Build a **semantic** form
    - Don't use elements for appearance
- Good class names helpful
    - I skipped in demo for space
- `<label>` are important for a11y
    - `for` on `<label>` must match target `id`
    - No `id`/`for` needed if wrapping target element
- Good styling for forms a very common need!
    - Forms often very customized by designers
    - Be cautious of fighting browser standards

Server POST example

```
const cats = {
  Jorts: {
    isTabby: false,
  },
  Jean: {
    isTabby: true,
  },
};

app.post(
  '/cats',
  express.urlencoded({ extended: false }),
  ( req, res ) => {
    const { name, isTabby } = req.body;
    cats[name] = {
      isTabby,
    };
    res.redirect('/cats');
  }
);
```

Redirects

A redirect is a special response

- Sets the status code of the response
- Includes a `Location` header with destination
- Browser will get response with 3xx status
    - And auto request that Location url
    - Request for Location will be a GET request
- Non-browser clients make decision
    - Whether to follow redirect response

Redirect after a successful POST for pages

Why?

- If they "reload" page
  - Browser RE-sends POST to change server
- After a redirect
  - "reload" just loads the Location again

Prevents unintended double-entry (or worse)

This is for *pages*

- Requests that return HTML pages
- Service calls follow different rules

Server-side Data

This class doesn't cover databases

- But the web doesn't change databases
- Your server program talks to database
    - Storing data from requests
    - Reading data to make responses
- We will do the same thing
    - Only using data in variables
    - Skip the "read/write" from/to Database step
- Our data will "reset" when server is restarted

Dynamic responses

How do we generate interesting HTML in responses?

- Based off of data?

Response Methods

- `res.redirect()` we've seen
- `res.status()` sends a status code
    - Defaults to 200 if you don't use
    - Does NOT complete response
- Remember response structure and order
    - one status line (first)
    - headers
    - body
- `res.send()` sends body content
    - Content is a string
        - Not a rendered page, just text

Dynamic Response Example

```
const cats = {
  Jorts: { isTabby: false },
  Jean: { isTabby: true },
};

function catList() {
  return Object.keys(cats).map( cat => `
    <li>
      ${cat}
      ${ cats[cat].isTabby ? 'is' : 'is NOT' } a Tabby
    </li>
  `).join('\n');
}

app.get('/cats', (req, res) => {
  res.send(` <html>
    <head> <title>Cat Results</title> </head>
    <body>
      ${ catList() }
    </body>
    </html> `);
});
```

Generating HTML

Using template literals to build HTML

- Tedious
- Minimal functionality

BUT it shows how webservers serve dynamic pages

- Using stored and passed data
- Generate a string of HTML
- Return it in response

All templating libraries and frameworks

- Still do this same process

Have Patience

This is what we'll use for now

- Including upcoming project(!)

To ensure you understand

- Server vs Client
- Request vs Response
- Stateless web
- Browser rendering

Later we'll write React and web services

Dynamic Routes

One more way to pass data to server

- In the path itself
    - Not query params AFTER path

Example: `/students/12345657`

- Might show student records with NEUID `1234567`
- Can handle ANY NEUID
- Uses what is in the path

Server Route can have "params"

- Not "query params"
- Stored in `req.params` object
- Defined in path with `:`

```
app.get('/student/:neuid', (req, res) => {
  console.log(req.params);
  res.send(`I see neuid of ${req.params.neuid}`);
});
```

- `:` not part of variable name
- `/` can separate multiple params

We will make more use of this when we get to services

Summary - HTML Forms

`<form>` element

- Various data elements
    - `<input>` with many options for `type`
    - `<select>` with `<option>`
    - `<textarea>`
- `<button type='submit'>` to submit
- `action` for target
    - Will NAVIGATE
- `method` of **GET** or **POST**
    - GET if not changing server data
    - POST if changing server data

Summary - Query Params

Sends form data in URL

- after `?`
- `key=value` pairs
- separated by `&`
- `key` and `value` are **URL-encoded**

A GET method form sends query params

Summary - URL Encoding

- Converts "special" characters to `%NN`
    - Where `NN` is hex ascii code for character
    - spaces often converted to `+` instead

Summary - Reading query in Express

Request and Response objects in Express

- Conventionally `req, res`

`req.query` object holds the key/value pairs from URL

Summary - Reading body in Express

POST method forms send data in body

- Still URL-encoded
- Must tell express to use middleware to translate
    - Can be for all requests, or per matching route
    - `express.urlencoded({ extended: false })`
- Will populate `req.body` object
    - If `undefined`, forgot to call middleware
- Servers reading POST forms should `redirect(...)`

Summary - Reading path params in Express

- Define route with colon(`:`) + variable name in path
- Will populate `req.params`
- Colon isn't part of variable name in `req.params`
- Can have multiple variables separated by `/`