Asynchronous Client-Server Communication

- Frequently, a web application will need to exchange more information between the client and server than is passed in the initial load of a page.

- A great example of such a situation is the “like” button on a Facebook post. When the user clicks a like button, not only does the client need to update the way the like like button is displayed in order to convey to the user that their action was recognized and took effect, it also needs to send information back to the server about what post was liked and by what user, so the server can record this information in its databases.

- In order to send this information back to the server, the Facebook page needs to make an HTTP request. Up to now, we’ve only seen HTTP requests used to fetch an entire page, e.g. when following a link, but this seems unnecessary in the context of sending information about a “like” back to the Facebook servers. Downloading an entire page again seems wasteful.

- Indeed, in this situation, we can employ a different method for initiating HTTP communication between the client and server, one that does not entail loading an entire page but instead involves sending only that information that is relevant to the situation, e.g. sending only information about a specific “like” action on a specific Facebook post.

- As we will see, this communication will be performed ***asynchronously***, with the client-side JS code initiating an HTTP request and then continuing to run—responding to events, etc.—while it waits for a response from the server, and when the server’s response is received, a callback will be invoked to handle it.

- In order to make all of this work, we need to do work in two places: on the server side, where we’ll set up routes that listen specifically for this type of communication from the client and update the information stored on the server-side in response to requests from the client; and on the client-side, where

we'll put code in place to initiate this type of communication with the server, usually in response to some interaction from the user.

Setting up a simple server-side API

- The server-side processes of a modern web application must typically do more than simply serve pages. In particular, a modern web application typically has a great deal of data stored in databases or elsewhere on the server side that are used to dynamically build the pages associated with the application. Thus, in addition to serving pages, the server processes must also manage access to this stored data.

- To do this, the server-side processes typically serve a set of routes that are specifically associated with data access (e.g. writing data, updating data, reading data, etc).

- For example, Facebook server-side code likely has one or more routes that are specifically associated with the "like" action. When a user likes a post on Facebook, the Facebook client-side code sends to this specific route (or routes) an HTTP request containing data about the like action.

- When the Facebook server-side code receives a request on one of these routes, it reads the "like" data from the request and makes sure the data is valid and that the client has permission to perform a "like" action, and if it does, the server-side code updates the Facebook databases to record information about the new "like".

- The Facebook server-side code likely contains hundreds or thousands of similar routes, each associated with different actions on different kinds of data, e.g. a route to process the creation of new posts, a route to handle a new message being sent, a route to handle the deletion of a photo, a route to handle an update to a specific user setting, etc.

- A set of routes like these that are specifically used to access/update application data is collectively referred to as an ***API*** (***application programming interface***).

- An application's API can be implemented directly within the same server code that handles serving web pages, or it can be implemented in a separate piece of server-side code. In either case, though, the code implementing the API will behave pretty much the same way.

- We'll start to explore writing API routes for a web application in which users can view pages of photos of specific people. In addition to viewing photos, we want to allow the user to add their own photos by specifying a photo URL and the text of a caption to be associated with the photo. When the user adds their own photo, we want to store the URL and caption in a database (or elsewhere) on the server side, so that the photo is saved and the user can see it the next time they visit the application on the web.

- To that end, we want to work on setting up a route to which an HTTP request can be sent to add a photo to our application's database of photos. We'll do this in the context of an Express-based server process like the ones we've been studying so far.

- There are many ways to set up the routes of an API (e.g. the paths associated with the routes and the HTTP methods used for API requests), and a full discussion of these options is beyond the scope of this course. We'll look at one simple way to set up our route for adding a photo to our photos database.

- Specifically, in this case, we'll create a route that listens for HTTP POST requests on a parameterized path, where the body of the POST request will specify the data about the new photo (i.e. the photo URL and the caption), and the request path will encode information about the person for whom the user is adding a photo (e.g. whether it's a photo of Luke Skywalker or Princess Leia or R2-D2, etc.). This kind of separation of data between the request path and the request body is very common in practice.

- To begin, we'll call `app.post()` to create a new route that listens specifically for HTTP POST requests. We'll leave the route path and its associated middleware function undefined for now:

```
app.post('...', function (req, res, next) {...});
```

- Let's assume that our photos database is organized so that each person for whom we have photos has a single database entry containing a list of all of their photos and that each person's database entry is accessed using a specific identifier unique to that person. For example, the photos for Luke Skywalker might be stored in a list accessed via the identifier `luke`, while Princess Leia's photos might be accessed via the identifier `leia`.

- Given this setup, we can specify our route's path so that the client must specify as a route parameter the identifier for the person for which a photo is being added. In particular, we'll use the path `'/people/:personId/addPhoto'`. This way, when a user wants to add a photo for Luke Skywalker, the client will make a request to the path `'/people/luke/addPhoto'`, or when the user wants to add a photo for Princess Leia, the client will make a request to `'/people/leia/addPhoto'`, etc.:

```
app.post('/people/:personId/addPhoto',
  function (req, res, next) {...});
```

- Before we move on to write the middleware function for our route, we'll need to discuss more specifically how data about a specific photo will be encoded in a POST request made to our new route.

- We chose to use POST requests for our route because POST requests typically contain a body of data being sent along with the request. For our specific situation, we'll encode in the POST body the URL and caption of the photo being added to our database.

- There are many ways to actually perform this encoding, however. Here, we will choose to encode our photo data as JSON, which is a common format for encoding HTTP POST body data. Specifically, we will encode our photo data as a JSON object with the fields `photoURL` and `caption`, each representing the corresponding information about the photo being added to the database:

```
{
  photoURL: "...",
  caption: "..."
}
```

- These objects will actually be *created* on the client side, when a user inputs information about a new photo to be added to our database. We will talk later about how this will be done.

- Right now, though, we need to be able to parse these objects when they arrive into our new route within the body of a POST request.

- Specifically, when our server receives a POST request to add a photo to our database, these JSON objects will be encoded as strings in the request bodies. We want to be able to parse these JSON strings into JavaScript objects, so we can easily access the data within our code.

- Fortunately, there is a nice 3rd-party module available on NPM that provides Express middleware that parses request bodies into forms more easily used in the code. This module is called `body-parser`. We'll start by installing it as a dependency within our project:

  ```
  npm install --save body-parser
  ```

- Once this dependency is installed, we can import it into our server code as we usually do:

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

- Once we have this dependency imported, we can plug its JSON body parsing middleware into our Express server as follows (usually before any routes are specified):

  ```
  app.use(bodyParser.json());
  ```

- This will result in all request bodies being parsed as JSON, which is OK here because, in this case, if any request to our server has a body, it will be a JSON-encoded body. Specifically, the JSON-parsed body of each request will be stored within the Express request object as a JS object called `req.body`.

- For example, when a POST request comes into our server whose body is a JSON-encoded object representing a photo, as above, it will be parsed into a JS object with the fields `photoURL` and `caption`, and we will have access to the values of these fields via `req.body.photoURL` and `req.body.caption`.

- With this parsing in place, we can write our route's middleware function:

```
app.post('/people/:personId/addPhoto', function (req, res){
  if (req.body && req.body.photoURL && req.body.caption) {
    console.log("== Client added the following photo:");
    console.log("  - person:", req.params.personId);
    console.log("  - url:", req.body.url);
    console.log("  - caption:", req.body.caption);

    // Add photo to DB here.

    res.status(200).send("Photo successfully added");
  } else {
    res.status(400).send("Requests to this path must " +
      "contain a JSON body with photoURL and caption " +
      "fields.");
  }
});
```

- Note that before we use the fields `req.body.photoURL` and `req.body.caption`, in this function, we need to verify that they exist. If they don't, we send a 400 error back to the client with a brief error message explaining the problem with their request.

- The specific way our server-side code adds the photo for a particular person into our photos database will depend very much on what kind of database we use to store our application data. Until we learn more about databases, we'll stick to simply logging out the photo data, as we do above.

Using `XMLHttpRequest` for client-side requests

- Now that we have mechanisms in place on the server side to listen for requests to add photos into our application's database, we need to modify our client-side code to actually send those requests.

- Let's assume that we already have code in place on the client side that can collect the information needed to formulate a request to add a photo to our database, specifically, the ID of the person for whom the user is adding a photo, the URL of the photo, and the caption text for the photo. Under this assumption, we'll proceed by writing a function that takes these values as arguments and communicates them back to the server:

```
function storePhotoInDB(personId, photoURL, caption) {...}
```

- On the client-side, we'll use a native JS class called `XMLHttpRequest` to generate and send our add-photo requests back to the server. `XMLHttpRequest` is designed to send asynchronous HTTP requests from client-side code.
    - Note that, despite the "XML" in its name, `XMLHttpRequest` is designed to work with many types of data, including JSON, which we'll be using here.

- The first thing we'll do in our function is simply create a new `XMLHttpRequest` object:

```
var request = new XMLHttpRequest();
```

- Once we have our `XMLHttpRequest` object, we need to tell it both which HTTP method to use in the associated request and to what URL to make the request. Here, we'll send the request back to the HTTP POST API route we wrote above. We'll assume that this route is running on the same host that serves our application's pages when formulating the URL:

```
var requestURL = '/people/' + personId + '/addPhoto';
request.open('POST', requestURL);
```

- Next, we'll formulate our POST request's body. In particular, the request body will be a JSON-encoded object formatted as above to represent the photo being added to the database. We can perform this encoding using the native JS method `JSON.stringify()`:

```
var photoObj = {
  photoURL: photoURL,
  caption: caption
};
var requestBody = JSON.stringify(photoObj);
```

- We'll add this body into our request when we send it below.

- Because we'll be sending a JSON string as our request body, we also have to add a header to the request to let the server know it contains JSON content:

```
request.setRequestHeader(
  'Content-Type', 'application/json'
);
```

- Remember, the request we send with `XMLHttpRequest` will be asynchronous. That is, the rest of our client-side code will continue to run while the request is being sent to and processed by the server. We simply need to specify a callback function to be called when the server's response arrives back to the client.

- This callback function will actually be registered with our `XMLHttpRequest` object the same way we registered event listeners with elements in the DOM, using `addEventListener()`. In fact, this is because the arrival of the server's response to our request will generate an event, much the same way a click on a button in the DOM generates an event.

- In this case, the server's response will generate a `'load'` event on our `XMLHttpRequest` object. Thus, in order to process the server's response, we'll specify a callback to be used in response to this `'load'` event:

  ```
  request.addEventListener('load', function (event) {...});
  ```

- Note that there are additional events we can listen for on `XMLHttpRequest` objects, such as `'error'`, `'abort'`, `'progress'`, `'timeout'`, and others. We won't discuss these here, but you can read more in the documentation for `XMLHttpRequest`.

- Within our event listener for the `'load'` event, we'll have access to the `XMLHttpRequest` object via `event.target`. We'll be able to get information about the completed request this way.

- For example, we can get the response status via `event.target.status`. We can use this to determine whether or not our request succeeded. Remember, in our API route above, we responded with status 200 if the request succeeded. Otherwise we responded with an error status and an error message. Thus, in our `'load'` event callback, we can check the response status and respond accordingly:

```
request.addEventListener('load', function (event) {
  if (event.target.status !== 200) {
    var message = event.target.response;
    alert("Error storing photo in database: " + message);
  } else {
    /*
     * Update UI to indicate that photo was successfully
     * stored.
     */
  }
});
```

- If the server's response indicates that the photo was successfully stored in the database, we can update the application UI accordingly, e.g. by actually displaying the photo on the page.

- In practice, we'll often want to perform more robust error handling than simply using `alert()` to display a raw error message from the server to the user.

- Finally, with this callback function in place, we can initiate our request, sending the body we formulated above:

  ```
  request.send(requestBody);
  ```

- And that's it! Calling `request.send()` will result in our request actually being sent back to the specified URL on our server, and our `'load'` event callback will be called whenever the server's response is received.

- With these pieces, we can build many more interactions into our site, incrementally updating our page as we send and receive new information to and from the server.