Client-Server Communication

James Wilcox and Kevin Zatloukal

May 2023

Sending Requests

GET

Suppose that we want to call a route called "`my-route`" on the server. If this route is used only to *retrieve* information (not to update any data) on the server, then it is reasonable to use a `GET` request, which is the simplest option. We make a `GET` request by calling the built-in `fetch` function, passing it only a URL.

Suppose that our route wants us to pass three arguments, `arg1`, `arg2`, and `arg3`, where `arg1` and `arg3` are strings and `arg2` is a number. For `GET` requests, the arguments must be passed in the URL, so all arguments must be converted to strings. Furthermore, URLs are restricted in size, so a `GET` request is only reasonable when we have small arguments that are easily converted to strings.

With those stipulations, we perform our `GET` request as follows:

```
const url = "/api/my-route" +
    "?arg1=" + encodeURIComponent(val1) +
    "&arg2=" + encodeURIComponent(String(val2)) +
    "&arg3=" + encodeURIComponent(val3);
fetch(url).then(this.doMyRouteResp)
    .catch((res) => this.doMyRouteError("failed to connect to server"));
```

The `encodeURIComponent` function must be applied to any string before it is added into the URL. This necessary to escape things like "&" that have special meaning in a URL. Also, since `val2`, the value we want to pass for `arg2`, is a number, we want to first convert it to a string using the function `String`.

The `fetch` method returns a `Promise<Response>`, which will invoke either the method we pass to "`then`", if it is able to connect and get a response status code from the server, or "`catch`", if it is not. Here, we create a function on the spot to pass to "`catch`" that calls our `doMyRouteError` method with an explanation of what went wrong.

The code above uses the method `doMyRouteResp`, which is defined as follows:

```
doMyRouteResp = (res: Response): void => {
  if (res.status === 200) {
    res.json().then(this.doMyRouteJson)
       .catch(() => this.doMyRouteError("200 response is not valid JSON"));
  } else if (res.status === 400) {
    res.text().then(this.doMyRouteError)
       .catch(() => this.doMyRouteError("400 response is not text"));
  } else {
    this.doMyRouteError(`bad status code ${res.status}`);
  }
};
```

This handles the response by, first, checking the status code. If it is a 200, then we know the server sent back JSON, so we can call `res.json()` to begin parsing the response into a JavaScript object. If it is a 400, then we know the server sent back an error message, so we can call `res.text()` to parse the response into a JavaScript string. If we see any other status code, our error message simply notes the code we received.

Once the JSON is parsed into a record, it is passed to `doMyRouteJson`, which is defined as follows:

```
doMyRouteJson = (val: unknown): void => {
  if (!isRecord(val)) {
    this.doMyRouteError(`bad type for val: ${typeof val}`);
  } else if (typeof val.ans1 !== "string") {
    this.doMyRouteError(`bad type for ans1: ${typeof val.ans1}`);
  } else if (typeof val.ans2 !== "number") {
    this.doMyRouteError(`bad type for ans2: ${typeof val.ans2}`);
  } else {
    // TODO: do something with val.ans1: string, val.ans2: number
  }
};
```

The type of the JSON value "`val`" passed to `doMyRouteJson` is `unknown`, so we must do run-time checks to make sure it is as expected. Here, we verify that it is a record, containing a field `ans1` of type `string` and a field `ans2` of type `number`. If they do not have these types, then we record an error.

All the error cases result in a call to this function that prints an error message in the console:

```
doMyRouteError = (msg: string): void => {
   console.error(`Error fetching /api/my-route: ${msg}`);
};
```

Error Handling

The code above is a lot of work for a single request to the server, but all of these lines are necessary. In particular, it is critical that we catch every possible error case, at least by printing an error message, so that we can figure out what happened when failures occur. Debugging will be extremely difficult if the code fails without any indications of what happened!

For the same reason, we need to carefully check the data types of the data that was returned by the server. When data is sent over the network, like this, there is no type checking provided. While ignoring the types by declaring `val` to have type "`any`" seems expedient, it will lead to painful debugging when errors inevitably occur. Instead, above, we accurately described in the type of `val` that the field values are `unknown` and then wrote code to confirm they their types were what we expected at run time.

While the arguments to a `GET` request are always (converted to) strings, the values returned in the response can be anything that can be sent via JSON: strings, numbers, null, arrays, and records. While any of these things can be sent, however, we still have to write code to check their types, so simpler data structures are always easier. Even sending an array of records would require us to check that it is an array, using `Array.isArray`, and then iterate through each element of the array, confirm that it is a record, by checking that "`isRecord(item)`", and then verify that each of the expected fields exist and has the right type, e.g., "`typeof item.field === 'number'`". As you can see, it is a lot of work, so simpler types of data should be preferred.

Note that receiving a response of the wrong shape is a *bug*, so the important thing is to fail fast and friendly. Report the error as quickly as possible so that the debugging will be as easy as it can be.

POST

A `POST` request should be used whenever we want to write data on the server or when we want to send arguments that may be large. `POST` request include a "body", which will be a JSON-encoded record containing, at the very least, the arguments that might be large. Note that `POST` requests are also allowed to use query parameters, but as noted above, their values must be (converted to) strings. In contrast, the values in the `POST` body can be anything that is valid JSON (see above). For that reason and for the sake of consistency, we will just use the `POST` body for sending arguments.

Our example from above can be converted into a `POST` request as follows, with all the arguments now passed in the `POST` body. Note that we no longer need to convert `arg2` to a String in this case because numbers are valid JSON. Here is the result:

```
const url = "/api/my-route";
const body = {arg1: val1, arg2: val2, arg3: val3};
fetch(url, {method: "POST", body: JSON.stringify(body),
      headers: {'Content-Type': 'application/json'}})
    .then(this.doMyRouteResp)
    .catch((res) => this.doMyRouteError("failed to connect to server"));
```

The rest of the code is unchanged.

When we are forming the request in the client, we are responsible for converting the `body` record into a string by calling `JSON.stringify`. The "`body`" field in the record passed to `fetch` must have a string value, just as the query arguments must have string values, and we are responsible for doing these conversions.

Receiving Requests

The route function on the sever must also check the types of the arguments.

```
// index.ts contains: app.post("/api/my-route", myRoute);

const myRoute = (req: SafeRequest, res: SafeResponse): void => {
  if (typeof req.body.arg1 !== "string") {
    res.status(400)
       .send(`my-route: arg1 has the wrong type ${typeof req.body.arg1}`);
  } else if (typeof req.body.arg2 !== "number") {
    res.status(400)
       .send(`my-route: arg2 has the wrong type ${typeof req.body.arg2}`);
  } else if (typeof req.body.arg3 !== "string") {
    res.status(400)
       .send(`my-route: arg3 has the wrong type ${typeof req.body.arg3}`);
  } else {
    // Now, know that req.body.arg1 and req.body.arg3 are strings and
    // that req.body.arg2 is a number.
    res.send({
        ans1: req.query.arg1 + "-" + req.body.arg3,  // concatenate some strings
        ans2: req.body.arg2 + 20                     // add some numbers
      });
  }
};
```

Note that passing numbers is easier in a POST body than in query parameters. If we wanted to pass `arg2` as a query parameter, like we did in our first example, then it would arrive on the server as a string, and we would need to convert it to a number using `parseInt(req.query.arg2)`. If the string value of `arg2` is not a valid number, this will return `NaN`, so we would need to include an additional check that `!Math.isNaN(parseInt(req.query.arg2))` to know that we have a valid number.

The code above, however, takes `arg2` (and `arg3`) in the POST body, so we get its value using `req.body.arg2`, and since the `POST` body is sent as JSON, it will arrive as a number, just as we sent it. Nonetheless, we still need to check that its type is correct. Again, skipping these checks will lead to some painful debugging.

Testing Requests

When testing our route function, we need to pass in a request with all the correct arguments in the correct places. For the example above, we would call it in a test as follows:

```
const req1 = httpMocks.createRequest({
    method: "POST", url: "/api/my-route",
    body: {arg1: "oak", arg2: 100, arg3: "cabinet"}
  });
const res2 = httpMocks.createResponse();
myRoute(req1, res1);
assert.strictEqual(res1._getStatusCode(), 200);
assert.deepStrictEqual(res1._getData(), {ans1: "oak-cabinet", ans2: 120});
```

Here, we are passing the arguments in the "`body`" field of the request because this is a POST request. For a GET request, we would put the arguments in a record inside the "`query`" field, and we would be restricted to using string values for each argument.

When we call the route function, it will invoke "`res.send(..)`" (with `res` referring to `res1`). The mock response handles a call to `send` by simply recording the record passed to it in a field. Later, when we call "`res1._getData()`" in the test, it returns the value that it stashed away in the field, and we can compare that to the record that we expected it to send. (Note that, in the test, the response is not actually converted to JSON. That only happens at run time.)

Remember to **never reuse** a request or response object in a test. Each call to a route function should use its own mock request and mock response object.

Debugging a Request

Client-server communication frequently involves debugging because there are so many different places where it can fail and because we are responsible for making sure that everything is handled properly. When we pass an argument to a function, the type checker will tell us if it is wrong, but when we are passing an argument to the server, the type checker cannot help us. We have to make sure that we do it right, which means we will inevitably make mistakes in some cases and end up debugging.

When that occurs, the first step is to figure out exactly what part of the client-communication process is failing. We can go through them in order as follows.

1. **Do you see the request in the Network tab?**

   If there is no request, then our code didn't actually make the call to `fetch`.

2. **Does the request in the Network tab show a 404 status code?**

   The arguments to `fetch` are wrong. We could have typed in the wrong URL, or used a `POST` when only `GET` is expected, or not passed a valid JSON string as `body`, or forgot the `Content-Type` header. Any mistake in any of those places will result in a 404 because the server cannot parse the request and get it to our route function.

3. **Does the request in the Network tab show a 400 status code?**

   Our code on the server rejected the request as invalid. This was most likely because an argument was missing or had the wrong type. Look at the Response in the Network tab. Since we took the time to send back an error message explaining the problem for each of our 400 responses, it should say there what the problem is. (You can also see the request that was sent to see what the value of the invalid argument actually was.))

   If that is not enough information, either change the error message to include more or add some `console.log` statements in the server code that will tell you more about what is going on. You will need to shut down the server and restart it to get it running with the new code.

4. **Does the request in the Network tab show a 500 status code?**

   Our code on the server crashed! Check the console where the server was started (using `npm run start`). There should be a stack trace.

5. **Does the request stay "pending" forever, never showing a status code?**

   Our route function got to the end without calling `send`. Fix the code to make sure that it always sends back either a record (which will have a 200 status message) or a a 400 status along with a string explanation of the error.

6. **Does it end up in `doMyRouteError` even though there is a 200 status code?**

   The error message passed to `doMyRouteError` should explain what went wrong. If that is not enough information, then either change the error message to include more or add some `console.log` statements to tell you more about what is happening.