Data Validation

When setting up HTML forms, it's important to check and clean the incoming data to ensure that the data is correct.

In this lesson, you will:

1. Understand what data validation is and why it's necessary for the server to validate incoming data.
2. Validate user-provided data from within an Express route handler function and return a response containing user-friendly validation error messages when necessary.

Importance of server-side data validation

Data validation is the process of ensuring that the incoming data is correct. This section will cover the rationale for validating incoming data on the server side.

Even though you could add add validations on the client side, client-side validations are not as secure and can be circumvented. Because client-side validations can be circumvented, it's necessary to implement server-side data validations.

Lack of trust in client-side validations

Let's talk through an example. Suppose you had an HTML form that collects a user's age. First of all, the whole point of the form is to collect the user's age, so you want to ensure that the "age" field is not blank. To account for this, you set a `required` attribute on the age `<input>`.
You also want to make sure that users submit a number for their age, so you set the `<input>` field's `type` attribute equal to "number":

```html
<for method="post" action="/age">
  <label for="age">Age: </label>
  <input required type="number" id="age" />
  <input type="submit" />
</form>
```

Excellent, now, whenever users fill out this form, they're unable to submit the form unless the "age" field is filled out with a number. This seems like it would ensure that you have clean and correct data being submitted to your server.

Unfortunately, those frontend validations are not reliable. Someone could open up the developer's console and remove the `required` attribute, and then change the `type` to equal "text".

Another situation to account for is that the end user might not even be using that specific form to submit data. Someone could be programmatically submitting a POST request to the server. In this scenario, they would never interact with the HTML form and its validations.

Ultimately, client-side validations are good for immediate feedback to the user, but they should not be relied upon for enforcing clean data submission.

Server-side validations

So what kind of data validations should you implement on the server side? Let's walk through a few examples.

Expected data types

The previous reading discussed how when a form is submitted, the data is typically urlencoded. One effect of this url encoding is that each value will arrive at the server as a string. Because of this, there's a tremendous need to validate that the provided string can be successfully converted to the desired type.

The previous example about the "age" field discussed how a user could circumvent the `type="number"` attribute on the frontend. Without server-side data validation on that "age" field, you could end up with an invalid value (for example, `NaN`) when trying to convert the "age" value into a number in the server.

Other examples of data type validations include:

- Checking for integer vs. float (perhaps you want to only store the user's age as an integer)
- Checking that an input date string can be converted to a valid date.

Valid ranges and format

To continue with our "age" field example, one logical validation you might want to enforce is that users submit a valid age. For example, it's unlikely that a user is over 120 years old.

You could also check that values come in the correct format. A telephone number should not have any letters in it, and if you want to ensure that it is a US-based telephone number, you might also want to check that the phone number is 10 digits long.

Another example might be that you want to ensure that your users are creating strong and secure passwords. To do this, you could require and check for the presence of a symbol and a number in the password, or prevent users from setting "password" as their password.

Other validations

Validations do not have to be constrained to just checking one field at a time. To continue with the example on passwords, let's suppose you also wanted to add a "Confirm Password" field to ensure that users did not make a typo on their password when creating an account. In this scenario, it's necessary to add a validation to ensure that the "Password" and "Confirmed Password" fields have the same value.

Validations could get even more complex based on the needs of your application. For example, let's suppose you have a form for users to order products. You probably want to validate that their selected shipment order is valid given the order's weight and destination postal code. After all, you don't want users trying to select "1-day delivery" for a couch that needs to be transported across the country.

Server-side validations: an example

Let's pick up where last reading's example left off and add some server-side validations. As a reminder, in the last reading, you built a website that allows you to add guests to a guest list.

Setup

At this moment, the directory of that example should look like this:

```
forms-demo
│   node_modules/
│   views/
│   │   guest-form.pug
│   │   index.pug
│   │   layout.pug
│   index.js
│   package-lock.json
│   package.json
```

The `index.js` file should look like this:

```
const express = require("express");

// Create the Express app.
```

```
const app = express();

// Set the pug view engine.
app.set("view engine", "pug");
app.use(express.urlencoded());

const guests = [];

// Define a route.
app.get("/", (req, res) => {
  res.render("index", { title: "Guest List", guests });
});

app.get("/guest", (req, res) => {
  res.render("guest-form", { title: "Guest Form" });
});

app.post("/guest", (req, res) => {
  const guest = {
    fullName: req.body.fullName,
    email: req.body.email,
    numGuests: req.body.numGuests
  };
  guests.push(guest);
  res.redirect("/");
});

// Define a port and start listening for connections.
const port = 8081;

app.listen(port, () => console.log(`Listening on port
${port}...`));
```

The `views/layout.pug` template should look like this:

```
doctype html
html
  head
    title= title
  body
    h1= title
    div
      a(href="/") Home
    div
      a(href="/guest") Add Guest

    block content
```

The `views/index.pug` file should look like this:

```
extends layout.pug

block content
```

```
  table
    thead
      tr
        th Full Name
        th Email
        th # Guests
    tbody
      each guest in guests
        tr
          td #{guest.fullName}
          td #{guest.email}
          td #{guest.numGuests}
```

Finally, the `guest-form.pug` should look like this:

```
extends layout.pug

block content
  h2 Add Guest
  form(method="post" action="/guest")
    label(for="fullName") Full Name:
    input(type="text" id="fullName" name="fullName")
    label(for="email") Email:
    input(type="email" id="email" name="email")
    label(for="numGuests") Num Guests
    input(type="number" id="numGuests" name="numGuests")
    input(type="submit" value="Add Guest")
```

Validations: checking that all fields are filled

First, because all three fields are important, let's add validations to ensure that each of the fields is filled out with a value before it can be successfully submitted.

To be clear, you can add a `required` attribute to the three `input` fields, and the user would not be able to submit until each field has a value. However, as was discussed in the previous reading, these kinds of front-end validations can be circumvented, so for this reading, let's focus on how to implement these validations in the server.
To do this, instantiate an `errors` array in `app.post('/guest')`. Then, check for truthy values in each of the `req.body` fields, and if any of the fields are missing, then push in an error message into the `errors` array to notify the user about how that field is required:

```
app.post("/guest", (req, res) => {
  const { fullName, email, numGuests } = req.body;
  const errors = [];

  if (!fullName) {
    errors.push("Please fill out the full name field.");
  }

  if (!email) {
    errors.push("Please fill out the email field.");
  }

  if (!numGuests) {
    errors.push("Please fill out the field for number of
guests.");
  }

  const guest = {
    fullName,
    email,
    numGuests
  };
  guests.push(guest);
  res.redirect("/");
});
```

Now, if the `errors` array has an error message in it, then don't add the `guest` to the `guests` array. Instead, give the user an opportunity to fix the errors before resubmitting the form again. You can do this by rendering the `guest-form.pug` template again along with the `errors` array. Then, update that template to display each error message to the user. Also, if there are errors, let's go ahead and return out of the callback function and ensure that none of the code below executes. Add this below the validations:

```
// VALIDATIONS HERE

if (errors.length > 0) {
  res.render("guest-form", { title: "Guest Form", errors });
  return; // `return` if there are errors.
}

// REST OF CODE NOT SHOWN
```

Here's what that route should look like now:

```
app.post("/guest", (req, res) => {
  const { fullName, email, numGuests } = req.body;
  const errors = [];
  if (!fullName) {
    errors.push("Please fill out the full name field.");
  }

  if (!email) {
    errors.push("Please fill out the email field.");
  }

  if (!numGuests) {
    errors.push("Please fill out the field for number of
guests.");
  }

  if (errors.length > 0) {
    res.render("guest-form", { title: "Guest Form", errors });
    return;
  }

  const guest = {
    fullName,
    email,
    numGuests
  };
  guests.push(guest);
  res.redirect("/");
});
```

Update `guest-form.pug` to display error messages whenever they exist:

```
extends layout.pug

block content
  div
    ul
      each error in errors
        li #{error}
  h2 Add Guest
  form(method="post" action="/guest")
    label(for="fullName") Full Name:
    input(type="text" id="fullName" name="fullName")
    label(for="email") Email:
    input(type="email" id="email" name="email")
    label(for="numGuests") Num Guests
    input(type="number" id="numGuests" name="numGuests")
    input(type="submit" value="Add Guest")
```

There's one more thing to fix here. One problem is that when the user navigates to `localhost:8081/guest` now, when `guest-form.pug` is rendered, the `app.get('/guest')` route is not

rendering an `errors` variable. Therefore, when `guest-form.pug` tries to iterate through the `errors` array, there's an error because `errors` does not exist.
You have a couple of options here: you can either check for the truthiness of `errors` in `guest-form.pug`, or you can render an empty `errors` array variable in the `app.get('/guest')` route callback. For now, let's go ahead and go with the first option and update the `guest-form.pug` template:

```
extends layout.pug

block content
  if errors
    div
      ul
        each error in errors
          li #{error}
  h2 Add Guest
  form(method="post" action="/guest")
    label(for="fullName") Full Name:
    input(type="text" id="fullName" name="fullName")
    label(for="email") Email:
    input(type="email" id="email" name="email")
    label(for="numGuests") Num Guests
    input(type="number" id="numGuests" name="numGuests")
    input(type="submit" value="Add Guest")
```

Things should be working properly now! If the user forgets to submit any of the fields, the user should get a very specific message about which field needs to be filled out still.

Validations: ensuring that `numGuests` is valid

Let's add a couple more validations on the `numGuests` field to get really comfortable with data validations. First, it probably makes sense that the number of guests per entry on the guest list is at least one. Also, as previously mentioned, each of the values will arrive at the server as a string. Although, JavaScript automatically converts strings into numbers when a string is being compared to a number, it's good practice to compare values of the same type.
This brings up another necessary validation. Add a validation that checks to make sure that the `numGuests` field is actually a value that can be converted into a number. When you've added these validations, your `app.post('/guest')` route should look something like this:

```
app.post("/guest", (req, res) => {
  const { fullName, email, numGuests } = req.body;
  const numGuestsNum = parseInt(numGuests, 10);
  const errors = [];

  if (!fullName) {
    errors.push("Please fill out the full name field.");
  }

  if (!email) {
    errors.push("Please fill out the email field.");
  }

  if (!numGuests || numGuests < 1) {
    errors.push("Please fill out a valid number for number of
guests.");
  }

  if (errors.length > 0) {
    res.render("guest-form", { title: "Guest Form", errors });
    return;
  }

  const guest = {
    fullName,
    email,
    numGuests: numGuestsNum
  };
  guests.push(guest);
  res.redirect("/");
});
```

There are now validations in place to ensure that the `numGuests` field is a valid number that is greater than 0. Test to make sure this is working properly by changing the `numGuests` input type to "text" and submitting invalid data (you can either edit this directly in `guest-form.pug` or by opening up the developers console to edit the HTML)

Improve user experience

One thing that's somewhat annoying right now is that any time there is a server-side error, all the fields get wiped, and the user has to fill them all out again. For example, even if the `fullName` and `email` fields were filled out without any issue, a small mistake on the `numGuests` field would require the user to have to start the whole process over again and fill out each field. Let's improve the user experience by pre-setting each field with the values that they had just submitted. To do this, whenever there is an error, not only should you render the `errors` array, but also go ahead and render back the values from `req.body`:

```
if (errors.length > 0) {
  res.render("guest-form", {
    title: "Guest Form",
    errors,
    email,
    fullName,
    numGuests
  });
  return;
}
```

Then, in `guest-form.pug`, set each input's `value` attribute to equal the associated variables that was rendered back:

```
extends layout.pug

block content
  if errors
    div
      ul
        each error in errors
          li #{error}
  h2 Add Guest
  form(method="post" action="/guest")
    label(for="fullName") Full Name:
    input(type="text" id="fullName" name="fullName"
value=fullName)
    label(for="email") Email:
    input(type="email" id="email" name="email" value=email)
    label(for="numGuests") Num Guests
    input(type="number" id="numGuests" name="numGuests"
value=numGuests)
    input(type="submit" value="Add Guest")
```

Go ahead and test out the improved user experience! Now, each field's value should persist even if there was an error.

Recap

In this lesson, you learned:

1. What data validation is and why it's necessary for the server to validate incoming data.
2. How to validate user-provided data from within an Express route handler function and return a response containing user-friendly validation error messages when necessary.