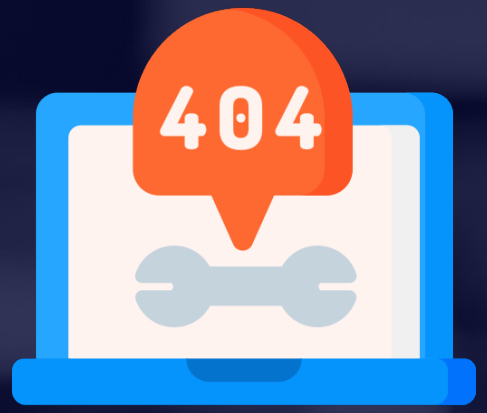
Validation and Error Handling



Validating User Input and Handle Different Type of Errors

Table of Contents

Have a Question?

#js-web

Validation

Why and How Validate Data?

Validation

- Why validate?
  - **Bigger app** === **more data** you will need from your users at some point of time
  - You should prevent the user from entering something **incorrect**
  - The validation can
    - either succeed and allow the data to be written to the database
    - reject the input and return some information

Validation

- How to validate?
  - **Client-Side**
    - Before any request is sent, we can write some JS that watches for input changes and approve the UX
    - It's optional because the user **can see**, **change** and **disable** the code because, JS runs in the browser
    - This is **not** a protection that secures you against incorrect data being sent to your server

Validation

- How to validate?
  - **Server-side**
    - The code **can't** be **seen**, **change** or **disabled**, because it happens on the server, not in the browser. This is the place where you should add validation and filter out the invalid data
    - After that, you will be sure you only work with valid data and store the correct information into the database

Validation

- How to validate?
  - **Database**
    - For most database engines there is a build in validation which you can turn on
    - It's not required, because there should be no scenario where your database work with invalid data
    - Make sure you have proper **server-side validation** and your database works with correct data

Validation

- **validator.js** - Is a library of string validators and sanitizers
  - Installation and Usage

```
npm install validator --save
```

    - Server-side usage

```
const validator = require('validator');
const body = req.body;
validator.isEmail(body.email); // true or false
```

    - Client-side usage

```
<script type="text/javascript" src="validator.min.js"></script>
<script type="text/javascript">
  validator.isEmail($('#email').val()); // true or false
</script>
```

Validation

- **express-validator** - Is a set of express.js middlewares that wraps **validator.js** validator and sanitizer functions
  - Installation and usage

```
npm install express-validator --save
```

```
const { check, validationResult } = require('express-validator');

check('email').isEmail()
check('password').isLength({ min: 5 });

const errors = validationResult(req);

if(!errors.isEmpty()) // Return 422 status and export errors

// Create user...
```

Validation

- **Sanitizers** are functions which implement **sanitization** which is
  - Making sure that the data is in the right format
  - Removing any illegal character from the data
    - **normalizeEmail**: canonicalizes an email address
    - **trim**: trim characters from both sides of the input
    - **blacklist** - remove characters that appear in the blacklist
    - and more...

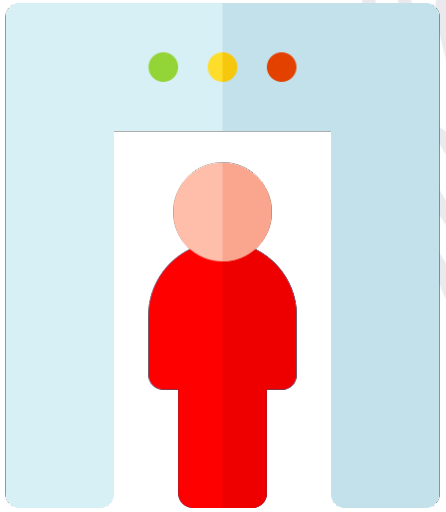
Validation

- **Sanitising** input is also something that makes sense to be done
  - You can do it in one step with validating

```
const { body } = require('express-validator');
body('email')
      .isEmail() // check if the string is an email (validation)
      .normalizeEmail(), // canonicalizes an email address (sanitization)
body('password')
      .isLength({ min: 5 })
      .isAlphanumeric()
      .trim() // trim characters (whitespace by default) - sanitization
```

Validation

- The sanitization **mutates** the request
- This means that if **req.body.email** was sent
  - with the value "**PeteR@ood.bg **"
  - after the sanitization its value will be "**peter@ood.bg**"



Validation

- Express-validators allows you to create **custom validations** and that send **custom messages**
- **Custom validator**

```
const { body } = require('express-validator');

app.post('/user', body.('email').custom(value => {
    return User.findUserByEmail(value)
        .then(user => {
            if(user){
                return Promise.reject('E-mail already in use');
            }
        });
};
```

Validation

- **Custom Sanitizer**
  - Can be implemented by using the method .**customSanitizer()**

```
const { sanitizeParam } = require('express-validator');

app.post('/object/:id', sanitizeParam('id').customSanitizer(value => {
  return ObjectId(value);
}), (req, res) => {
  // Handle the request...
});
```

Mongoose Validation

- Validation is defined in the **SchemaType**
- Validation is middleware
  - Mongoose registers validation as a **`pre('save')`** hook
  - It's **asynchronously recursive**
  - can be customizable
- **unique** option for schemas is not validator
  - It's a convenient helper for building MongoDB unique indexes

Mongoose Save/Validate Hooks

- The **save()** function triggers **validate()** hook
  - all **pre('validate')** and **post('validate')** hooks get called before any **pre('save')** hook

```
schema.pre('validate', function() {
  console.log('this gets printed first');
});
schema.post('validate', function() {
  console.log('this gets printed second');
});
schema.pre('save', function() {
  console.log('this gets printed third');
});
schema.post('save', function() {
  console.log('this gets printed fourth');
});
```

Mongoose Built-in Validators

- All **SchemaTypes** have built-in required validator
  - **Numbers** have min and max validators
  - **Strings** have **enum**, **match**, **minelngth** and **maxlength**

```
const userSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true,
        minlength: 4,
        maxlength: 20,
    },
});
```

Mongoose Custom Validators

- If the build-in validators aren't enough, you can define **custom validators** to suit your needs

```
var userSchema = new Schema({
  phone: {
    type: String,
    validate: {
      validator: function(v) {
        return /\d{3}-\d{3}-\d{4}/.test(v);
      },
      message: props => `${props.value} is not a valid phone number!`
    },
    required: [true, 'User phone number required']
  }
});
```

Mongoose Validation Errors

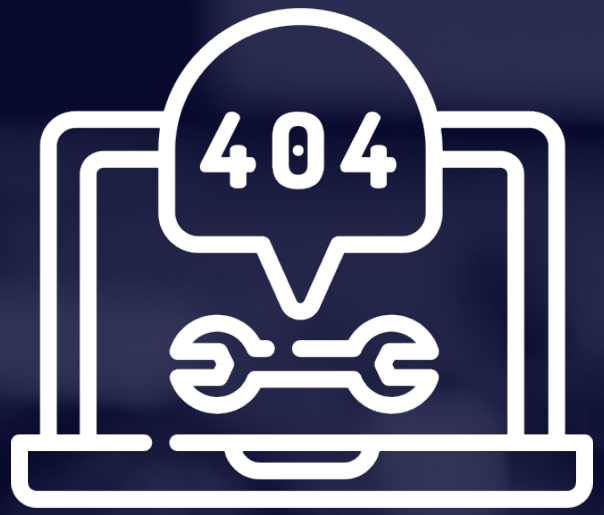
- Errors returned after failed validation contain an **error object** whose values are **ValidatorError** object
  - has **kind**, **path**, **value** and **message** properties

```
toy.save((err) => {
        assert.equal(err.errors.color.message, 'Color');
        assert.equal(err.errors.color.kind, 'Invalid color');
        assert.eqial(err.errors.color.path, 'color');
        assert.equal(err.errors.color.value, 'Green');
        ...
});
```

Validation

- No matter which approach you choose, in the end some of the validations can fail
  - You should **always return** a helpful error **message** to the user
  - **Never reload** the page but always keep the user data inserted because that is a bad user experience
- More info
  - https://express-validator.github.io/docs/
  - https://mongoosejs.com/docs/validation.html

Validation Demo



Error Handing

Error Handling

- Errors in your code should be handled properly
- These errors can be different types
  - **Technical/Network** Errors
  - "**Usual**"/"**Expected**" Errors
  - **Bugs/Logical** Errors

Error Handling

- **Technical/Network** errors
  - MongoDB server might be down
- "**Usual**"/"**Expected**" Errors
  - File can't be reads or some database operation fails
- **Bugs/Logical**
  - User object used when it doesn't exist
    - This errors are our fault
    - They should be fixed during development

Working with Errors

- An error is a **technical object** in a node application. This built-in error object can be thrown
  - Synchronous code
    - **try-catch**
  - Asynchronous code
    - **then()-catch()**
- In the end in both scenario, you have to choice
  - Directly handle the error
  - Use **ExpressJS** functionality

Error Handling

- There is a scenarios where you **can't continue** but there is **no technical error**
  - If some user try to login but the username does not exist
    - You must check the values and decide what to do
      - Thrown an error
      - Directly handle the "error"

Error Handling

- Handling errors synchronously

```
const User = require('../models/User/);

async (req, res, next) => {
    const { username, password } = req.body;
    try{
      const currentUser = await User.findOne({ username });
      // Login...
    } catch (e) {
      // Handle error properly...
    }
};
```

Error Handling

- Handling errors asynchronously

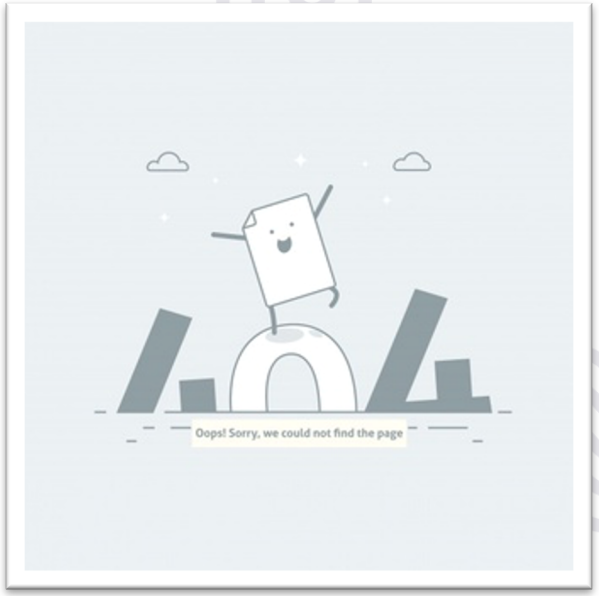
```
Post.findById(postId)
  .then((post) => {
    // Delete post
  })
  .catch(error => {
    if (!error.statusCode) {
       error.statusCode = 500;
    }
    next(error);
  })
```

If there is no status code attached, then something went wrong with the server

The error is sent to the middleware

Error Handling

- In all cases you can
  - Return an **error page**
  - Return a response with **error information**
  - **Redirect**



Error Handling Demo

Summary

- Validation
  - Why and how validate data?
  - Validating and sanitization data with express-validator
  - Mongoose validator
- Error Handling
  - Different types of errors

Questions?

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © Kingsland University – https://kingslanduniversity.com

KINGSLAND UNIVERSITY

THANK YOU