

Validation and Error Handling

Validating User Input and Handle Different Type of Errors



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

1. Validation

- Why and how to validate data?
- Validation and sanitization data with express-validator
- Mongoose validation

2. Error Handling

- Different types of errors



sli.do

#js-web



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**

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

- How to validate?
 - **Server-side**
 - The code **can't** be **seen**, **changed** or **disabled**, because it happens on the server, not in the browser.
 - **The server** 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

- 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

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

- Installation and Usage

```
npm install validator
```

- 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>
```

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

```
npm install express-validator
```

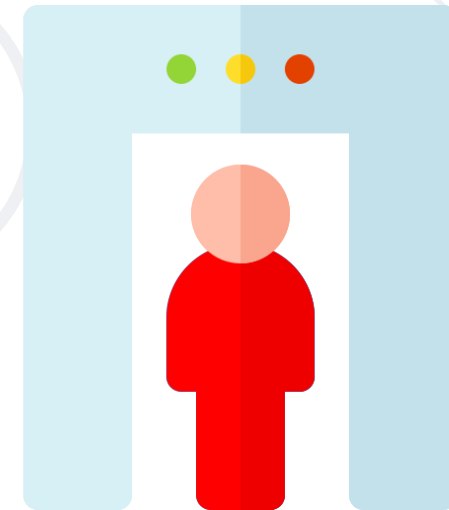
```
const { check, validationResult } = require('express-validator');  
  
check('email').isEmail()  
check('password').isLength({ min: 5 });  
  
const errors = validationResult(req);  
  
if(!errors.isEmpty()) // Return 422 status and export errors  
  
// Create user...
```

- **Sanitizers** are functions that implement **sanitization** which is
 - Make 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 on the blacklist
 - and more...

- **Sanitizing** input is also something that makes sense to be done
 - You can do it in one step by 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
```

- 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**"



- 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');
      }
    });
}));
```

- 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...
});
```

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

- 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');  
});
```

- All **SchemaTypes** have built-in required validator
 - **Numbers** have min and max validators
 - **Strings** have **enum**, **regex**, **minLength** and **maxLength**

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

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

```
const 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']
  }
});
```

- Errors returned after failed validation contain an **error object** whose values are **ValidatorError** object
 - has a **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.equal(err.errors.color.path, 'color');  
  assert.equal(err.errors.color.value, 'Green');  
  ...  
});
```

- No matter which approaches 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 Handling

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

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

- 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 scenarios, you have to choice
 - Directly handle the error
 - Use **ExpressJS** functionality

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

- 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...
  }
};
```

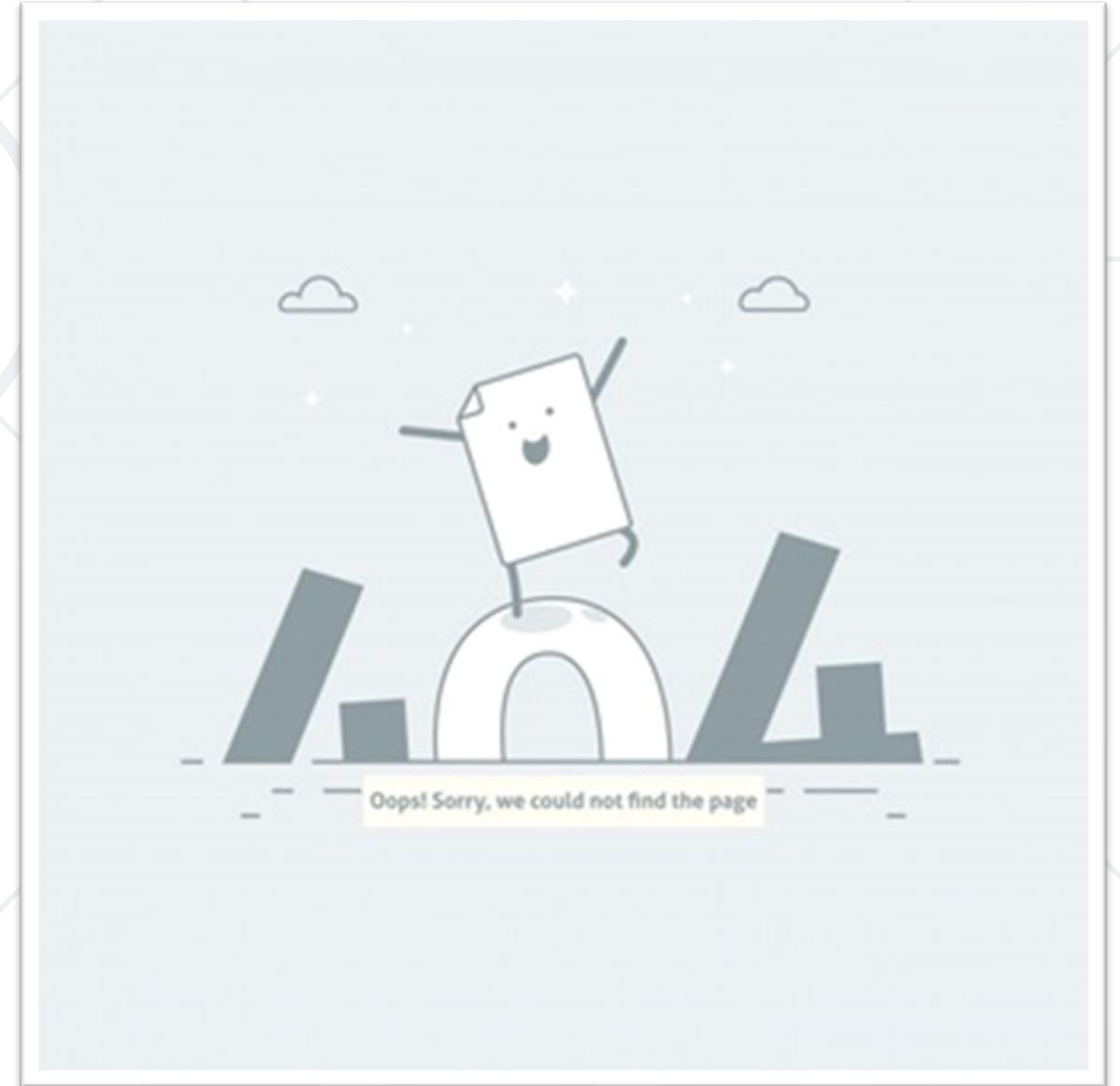
- Handling errors asynchronously

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

If status code is missing, then something went wrong with the server

The error is sent to the middleware

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



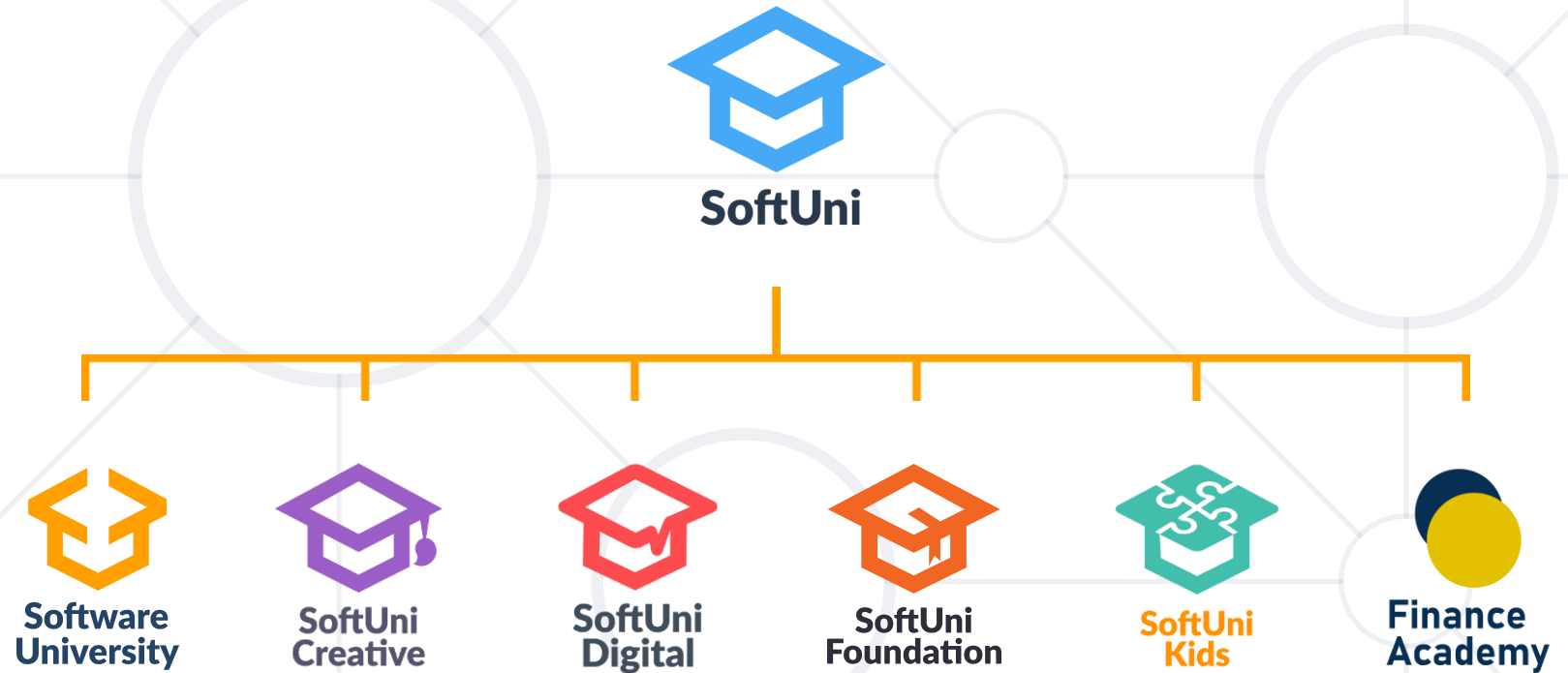


Error Handling Demo

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



Questions?



SoftUni Diamond Partners

**SUPER
HOSTING
.BG**



**Coca-Cola HBC
Bulgaria**



POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers



AMBITIONED

 **DRAFT
KINGS**



**SOFTWARE
GROUP**

createX



Postbank
Решения за твоето утре

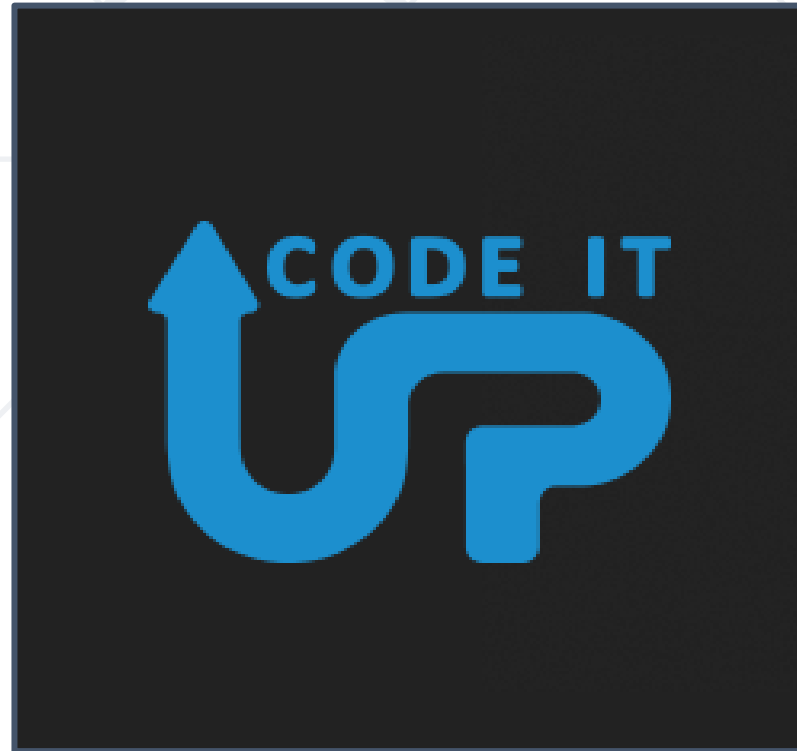


BOSCH

DXC
TECHNOLOGY



SmartIT



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
 - Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

