- ⏱️ 2 hours
- ✅ Completed

Data-Driven Websites - Part 3: Using Sequelize to Perform CRUD Operations

Welcome to part three of creating the data-driven Reading List website!

Over the course of three articles, you'll create a data-driven Reading List website that will allow you to view a list of books, add a book to the list, update a book in the list, and delete a book from the list. In the first article, you created the project. In the second article, you learned how to integrate Sequelize with an Express application. In this article, you'll create the routes and views to perform CRUD (create, read, update, and delete) operations using Sequelize.

When you finish this article, you should be able to:

- Define a collection of routes and views that use Sequelize to perform CRUD operations against a single resource; and
- Handle Sequelize validation errors when users are attempting to create or update data and display error messages to the user so that they can resolve any data quality issues.

You'll also review the following:

- Using a wrapper function to catch errors thrown within asynchronous route handler functions;
- Using Pug to create HTML forms;
- Using the `csurf` middleware to protect against CSRF exploits;
- Using the built-in `express.urlencoded()` middleware function to parse incoming request body form data;
- Using Sequelize model validations to validate user-provided data;
- Using the `express-validator` validation library to validate user-provided data within an Express route; and
- Using Pug includes and mixins to remove unnecessary code duplication.

Planning the routes and views

Before creating any new routes or views, it's a good idea to plan out what pages need to be added to support the required CRUD (create, read, update, and delete) operations along with their associated routes, HTTP methods, and views.

Here's a list of the proposed pages to add to the Reading List application:

Page Name	Route Path	HTTP Methods	View Name
Book List	`/`	`GET`	`book-list.pug`
Add Book	`/book/add`	`GET POST`	`book-add.pug`
Edit Book	`/book/edit/:id`	`GET POST`	`book-edit.pug`
Delete Book	`/book/delete/:id`	`GET POST`	`book-delete.pug`

There are a number of acceptable ways that you could approach implementing the required CRUD operations for the `Book` resource or model. The above approach is a common, tried-and-true way of implementing CRUD operations within a server-side rendered web application.

*The term **server-side rendered** simply means that all of the work of generating the HTML for the web application's pages is done on the server. Later on, you'll learn how to use client-side technologies like React to move some of that work to the client (i.e. the browser).*

Notice that the "Add Book", "Edit Book", and "Delete Book" pages need to support both the `GET` and `POST` HTTP methods. The `GET` HTTP method will be used to initially retrieve each page's HTML form while the `POST` HTTP method will be used to process each page's HTML form submissions.

Also notice that the route paths for the "Edit Book" and "Delete Book" pages define an `:id` route parameter. Without a book ID, those pages wouldn't know what book record they were supposed to be editing or deleting. The "Add Book" page doesn't need an `:id` route parameter because that page is adding a new book record, so a book ID isn't needed (the ID for the new record will be created by the database when the record is inserted into the table).

Now that you have a plan, let's start building out the proposed pages—starting with the "Book List" page!

Creating the Book List page

As a reminder, here's what the default route (`/`) in the `routes` module (i.e. the `routes.js` file) looks like at this point:

```
router.get('/', async (req, res, next) => {
```

```
  try {
    const books = await db.Book.findAll({ order: [['title',
'ASC']] });
    res.render('index', { title: 'Home', books });
  } catch (err) {
    next(err);
  }
});
```

And the `./views/index.pug` view:

```
//- ./views/index.pug

extends layout.pug

block content
  p Hello from the Reading List app!
  h3 Books
  ul
    each book in books
      li= book.title
```

It's a small change, but start with renaming the `./views/index.pug` view to `./views/book-list.pug`. Changing the name of the view will make it easier to identify the purpose of the view at a glance.

After renaming the view, update the call to the `res.render()` method in the default route:

```
router.get('/', async (req, res, next) => {
  try {
    const books = await db.Book.findAll({ order: [['title',
'ASC']] });
    res.render('book-list', { title: 'Books', books });
  } catch (err) {
    next(err);
  }
});
```

Notice that the `title` property—on the object passed as the second argument to the `res.render()` method—was changed from "Home" to "Books".

Applying Bootstrap styles to the Book List page

When you added Bootstrap to the project in the first article in this series, it was mentioned that the look of the application wouldn't change much at that point. Let's change that!

Update the `./views/book-list.pug` view with the following code:

```
//- ./views/book-list.pug
```

```
extends layout.pug

block content
  div(class='py-3')
    a(class='btn btn-success' href='/book/add' role='button')
Add Book
  table(class='table table-striped table-hover')
    thead(class='thead-dark')
      tr
        th(scope='col') Title
        th(scope='col') Author
        th(scope='col') Release Date
        th(scope='col') Page Count
        th(scope='col') Publisher
        th(scope='col')
    tbody
      each book in books
        tr
          td= book.title
          td= book.author
          td= book.releaseDate
          td= book.pageCount
          td= book.publisher
          td
            a(class='btn btn-primary'
href=`/book/edit/${book.id}` role='button') Edit
            a(class='btn btn-danger ml-2'
href=`/book/delete/${book.id}` role='button') Delete
```

Here's an overview of the above Pug template code:

- A hyperlink (`<a>`) at the top of the page (`a(class='btn btn-success' href='/book/add' role='button') Add Book`) gives users a way to navigate to the "Add Book" page. The hyperlink is styled to look like a button using the Bootstrap button CSS classes (`btn btn-success`).
- An HTML table is used to render the list of books. The Bootstrap table CSS classes (`table table-striped table-hover`) are used to style the table.
- Each row in the books HTML table contains two hyperlinks—one to navigate to the "Edit Book" page and another to navigate to the "Delete Book" page. Again, both hyperlinks are styled to look like buttons using the Bootstrap button CSS classes.

For more information about the Bootstrap front-end component library, see the official documentation.

Adding an asynchronous route handler wrapper function

In an earlier article, you learned that Express is unable to catch errors thrown by asynchronous route handlers. Given that, asynchronous route handlers need to catch their own errors and pass them to the `next()` method. That's exactly what the default route handler is currently doing:

```
router.get('/', async (req, res, next) => {
  try {
    const books = await db.Book.findAll({ order: [['title',
'ASC']] });
    res.render('book-list', { title: 'Books', books });
  } catch (err) {
    next(err);
  }
});
```

While you could continue to add `try`/`catch` statements to each of your route handlers, defining a simple asynchronous route handler wrapper function will keep you from having to write that boilerplate code:

```
const asyncHandler = (handler) => (req, res, next) =>
handler(req, res, next).catch(next);

router.get('/', asyncHandler(async (req, res) => {
  const books = await db.Book.findAll({ order: [['title',
'ASC']] });
  res.render('book-list', { title: 'Books', books });
}));
```

For your reference, here's what the `./routes.js` file should look like at this point in the project:

```
// ./routes.js

const express = require('express');

const db = require('./db/models');

const router = express.Router();

const asyncHandler = (handler) => (req, res, next) =>
handler(req, res, next).catch(next);

router.get('/', asyncHandler(async (req, res) => {
 const books = await db.Book.findAll({ order: [['title', 'ASC']]
});
 res.render('book-list', { title: 'Books', books });
}));

module.exports = router;
```

Testing the Book List page

Open a terminal and browse to your project folder. Run the command `npm start` to start your application and browse to `http://localhost:8080/`. You should see the list of books from the database rendered to the page—but instead of using an unordered list to format the list of books you should see a nicely Bootstrap formatted HTML table!

Adding the Add Book page

The next page that you'll add to the Reading List application is the "Add Book" page. As the name clearly suggests, this page will allow you to add a new book to the reading list.

Adding protection from CSRF attacks

Before adding the route and view for the "Add Book" page, go ahead and prepare to add protection from CSRF attacks by installing and configuring the necessary dependencies and middleware.

To review, Cross-Site Request Forgery (CSRF) is an attack that results in an end user executing unwanted actions within a web application. Imagine that the Reading List website requires users to login before they can view and make changes to their reading list (in a future article you'll learn how to implement user login within an Express application!) If a user was currently logged into the Reading List website, a CSRF attack would trick the user into clicking a link that unexpectedly sends a POST request to the Reading List website—a request that might add or delete a book without the user's consent!

While this particular example is trivial in terms of its impact to the user, imagine that the affected web application is a banking application. The end user could end up unintentionally transferring money to the hacker's bank account!

For a detailed walkthrough of a CSRF attack and how to protect against CSRF attacks, see the "Protecting Forms from CSRF" article in the Express HTML Forms lesson.

From a terminal, install the following dependencies into your project:

```
npm install csurf@^1.0.0
npm install cookie-parser@^1.0.0
```

Within the `app` module (i.e. the `./app.js` file), use the `require()` function to import the `cookie-parser` middleware and call the `app.use()` method to add the middleware

just after adding the `morgan` middleware to the request pipeline. While you're updating the `app` module, go ahead and add the built-in Express `urlencoded` middleware after adding the `cookie-parser` middleware (you'll need the `urlencoded` middleware to parse the request body form data in just a bit):

```
// ./app.js

const express = require('express');
const morgan = require('morgan');
const cookieParser = require('cookie-parser');

const routes = require('./routes');

const app = express();

app.set('view engine', 'pug');
app.use(morgan('dev'));
app.use(cookieParser());
app.use(express.urlencoded({ extended: false }));
app.use(routes);

// Code removed for brevity.

module.exports = app;
```

Defining the routes for the Add Book page

Now you're ready to define the routes for the "Add Book" page!

At the top of the `routes` module (i.e. the `./routes.js` file), add a call to the `require()` function to import the `csurf` module:

```
// ./routes.js

const express = require('express');
const csrf = require('csurf');

const db = require('./db/models');

// Code removed for brevity.
```

Then call the `csurf()` function to create the `csrfProtection` middleware that you'll add to each of the routes that need CSRF protection:

```
// ./routes.js

const express = require('express');
const csrf = require('csurf');
```

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

const router = express.Router();

const csrfProtection = csrf({ cookie: true });

const asyncHandler = (handler) => (req, res, next) =>
handler(req, res, next).catch(next);

// Code removed for brevity.
```

Now you're ready to add the routes for the "Add Book" page to the `routes` module just after the existing default route (`/`)—a `GET` route to initially retrieve the "Add Book" page's HTML form and a `POST` route to process the page's HTML form submissions:

```
router.get('/book/add', csrfProtection, (req, res) => {
  const book = db.Book.build();
  res.render('book-add', {
    title: 'Add Book',
    book,
    csrfToken: req.csrfToken(),
  });
});

router.post('/book/add', csrfProtection, asyncHandler(async
(req, res) => {
  const {
    title,
    author,
    releaseDate,
    pageCount,
    publisher,
  } = req.body;

  const book = db.Book.build({
    title,
    author,
    releaseDate,
    pageCount,
    publisher,
  });

  try {
    await book.save();
    res.redirect('/');
  } catch (err) {
    res.render('book-add', {
      title: 'Add Book',
      book,
      error: err,
      csrfToken: req.csrfToken(),
```

```
    });
  }
}));
```

Here's an overview of the above routes:

- Two routes are defined for the "Add Book" page—a `/book/add GET` route and a `/book/add POST` route. As mentioned earlier, the `GET` route is used to initially retrieve the page's HTML form while the `POST` route is used to process submissions from the page's HTML form.
- Both routes use the `csrfProtection` middleware to protect against CSRF attacks.
- Within the `GET` route handler, the Sequelize `db.Book.build()` method is used to create a new instance of the `Book` model which is then passed to the `book-add` view.
- Within the `POST` route handler, destructuring is used to declare and initialize the `title`, `author`, `releaseDate`, `pageCount`, and `publisher` variables from the `req.body` property. The `title`, `author`, `releaseDate`, `pageCount`, and `publisher` variables are then used to create a new instance of the `Book` model with a call to the `db.Book.build()` method. The `book.save()` method is called on the instance to persist the model to the database and if that operation succeeds the user is redirected to the default route (`/`). If an error occurs, the `book-add` view is rendered and sent to the client (so the error can be displayed to the end user).

Creating the view for the Add Book page

Add a view to the `views` folder named `book-add.pug` containing the following code:

```
//- ./views/book-add.pug

extends layout.pug

block content
  if error
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      pre= JSON.stringify(error, null, 2)
  form(action='/book/add' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    div(class='form-group')
      label(for='title') Title
      input(type='text' id='title' name='title'
value=book.title class='form-control')
    div(class='form-group')
      label(for='author') Author
      input(type='text' id='author' name='author'
value=book.author class='form-control')
    div(class='form-group')
```

```
      label(for='releaseDate') Release Date
      input(type='text' id='releaseDate' name='releaseDate'
value=book.releaseDate class='form-control')
    div(class='form-group')
      label(for='pageCount') Page Count
      input(type='text' id='pageCount' name='pageCount'
value=book.pageCount class='form-control')
    div(class='form-group')
      label(for='publisher') Publisher
      input(type='text' id='publisher' name='publisher'
value=book.publisher class='form-control')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Add Book
      a(href='/' class='btn btn-warning ml-2') Cancel
```

Here's an overview of the above Pug template code:

- A conditional statement checks to see if the `error` variable is truthy (i.e. has a reference to an error) and if there's an error, the `JSON.stringify()` method is used to render the error to the page as JSON. Later in this article, you'll refactor this part of the view to improve the display of errors to the end user.
- A hidden `<input>` element is used to render the CSRF token value to the page (i.e. `input(type='hidden' name='_csrf' value=csrfToken)`).
- A series of `<label>` and text `<input>` elements are rendered to create the form fields for the `Book` model `title`, `author`, `releaseDate`, `pageCount`, and `publisher` properties. The Bootstrap form CSS classes (`form-group`, `form-control`) are used to style the form.
- At the bottom of the form, a submit `<button>` element is rendered along with a "Cancel" hyperlink that allows the end user to navigate back to the "Book List" page.

Note: *HTML `<input>` element types aren't used to their fullest extent in the above code. Feel free to experiment with using the available `<input>` element types to add client-side validation but remember that client-side validation is intended only to improve the end user experience. Because client-side validation can easily be thwarted, validating data on the server is absolutely essential to do. You'll implement server-side validation in just a bit.*

Testing the Add Book page

Run the command `npm start` to start your application and browse to `http://localhost:8080/`. Click the "Add Book" button at the top of the "Book List" page to browse to the "Add Book" page. Provide a value for each of the form fields and click the "Add Book" button to submit the form to the server. Be sure that you provide a valid date value (i.e. "2000-01-31"). You should now see your new book in the list of books on the "Book List" page!

If you click the "Add Book" button again and submit the "Add Book" page form without providing any values, an error occurs when attempting to persist an instance of

the `Book` model to the database. The lengthy error message displayed just above the form will look like this:


```
{
  "name": "SequelizeDatabaseError",
  "parent": {
    "name": "error",
    "length": 116,
    "severity": "ERROR",
    "code": "22007",
    "file": "datetime.c",
    "line": "3774",
    "routine": "DateTimeParseError",
    "sql": "INSERT INTO \"Books\"
(\"id\",\"title\",\"author\",\"releaseDate\",\"pageCount\",\"p
ublisher\",\"createdAt\",\"updatedAt\") VALUES
(DEFAULT,$1,$2,$3,$4,$5,$6,$7) RETURNING *;",
    "parameters": [
      "",
      "",
      "Invalid date",
      "",
      "",
      "2020-04-02 15:20:33.668 +00:00",
      "2020-04-02 15:20:33.668 +00:00"
    ]
  },
  "original": {
    "name": "error",
    "length": 116,
    "severity": "ERROR",
    "code": "22007",
    "file": "datetime.c",
    "line": "3774",
    "routine": "DateTimeParseError",
    "sql": "INSERT INTO \"Books\"
(\"id\",\"title\",\"author\",\"releaseDate\",\"pageCount\",\"p
ublisher\",\"createdAt\",\"updatedAt\") VALUES
(DEFAULT,$1,$2,$3,$4,$5,$6,$7) RETURNING *;",
    "parameters": [
      "",
      "",
      "Invalid date",
      "",
      "",
      "2020-04-02 15:20:33.668 +00:00",
      "2020-04-02 15:20:33.668 +00:00"
    ]
  },
  "sql": "INSERT INTO \"Books\"
(\"id\",\"title\",\"author\",\"releaseDate\",\"pageCount\",\"p
```

```
ublisher\",\"createdAt\",\"updatedAt\") VALUES
(DEFAULT,$1,$2,$3,$4,$5,$6,$7) RETURNING *;",
  "parameters": [
    "",
    "",
    "Invalid date",
    "",
    "",
    "2020-04-02 15:20:33.668 +00:00",
    "2020-04-02 15:20:33.668 +00:00"
  ]
}
```

From the error message, you can see that a `SequelizeDatabaseError` occurred when attempting to insert into the `Books` table. The underlying error is a date/time parse error, which is occurring because you didn't supply a value for the `releaseDate` property on the `Book` model.

It's not just empty strings that result in date/time parse errors. Improperly formatted date/time `string` values—or simply bad `string` values—can also produce date/time parse errors. For example, all of the following `string` date/time values cannot be parsed to date/time values:

- Jan 31st 2002
- 100/31/2002
- Jaanuary 31, 2002

You can use the `input` element's `placeholder` attribute to communicate to users an example of the expected input format. Refactor your `input#releaseDate` element to include a placeholder:

```
input(type='text'
      id='releaseDate'
      name='releaseDate'
      value=book.releaseDate
      class='form-control'
      placeholder='ex: 2000-01-31')
```

Time to implement server-side validations! You'll see how to implement validations using two different approaches—within the `Book` database model using Sequelize's built-in model validation and within the "Add Book" page `POST` route using the `express-validator` validation library.

Implementing server-side validation using Sequelize

Before updating the `Book` model (the `./db/models/book.js` file), make a copy of the existing code by copying the entire file with a file extension of `.bak` (i.e. `book.js.bak`) or simply copying and pasting the code within the existing file and commenting it out.

When implementing validation at the route level using a validation library, you'll want a convenient way to remove or disable the validations in the `Book` model.

Adding validations to the `Book` model

Now you're ready to update the `Book` model to the following code:

```
// ./db/models/book.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Book = sequelize.define('Book', {
    title: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: 'Please provide a value for Title',
        },
        notEmpty: {
          msg: 'Please provide a value for Title',
        },
        len: {
          args: [0, 255],
          msg: 'Title must not be more than 255 characters
long',
        }
      }
    },
    author: {
      type: DataTypes.STRING(100),
      allowNull: false,
      validate: {
        notNull: {
          msg: 'Please provide a value for Author',
        },
        notEmpty: {
          msg: 'Please provide a value for Author',
        },
        len: {
          args: [0, 100],
          msg: 'Author must not be more than 100 characters
long',
        }
      }
    },
    releaseDate: {
      type: DataTypes.DATEONLY,
      allowNull: false,
      validate: {
        notNull: {
```

```
          msg: 'Please provide a value for Release Date',
        },
        isDate: {
          msg: 'Please provide a valid date for Release Date',
        }
      }
    },
    pageCount: {
      type: DataTypes.INTEGER,
      allowNull: false,
      validate: {
        notNull: {
          msg: 'Please provide a value for Page Count',
        },
        isInt: {
          msg: 'Please provide a valid integer for Page Count',
        }
      }
    },
    publisher: {
      type: DataTypes.STRING(100),
      allowNull: false,
      validate: {
        notNull: {
          msg: 'Please provide a value for Publisher',
        },
        notEmpty: {
          msg: 'Please provide a value for Publisher',
        },
        len: {
          args: [0, 100],
          msg: 'Publisher must not be more than 100 characters
long',
        }
      }
    }
  }, {});
  Book.associate = function(models) {
    // associations can be defined here
  };
  return Book;
};
```

Here's an overview of the above code:

- Sequelize validation rules or validators are applied to model properties—referred to by Sequelize as "attributes"—using the `validate` property. The `validate` property is set to an object whose properties represent each validation rule to apply to the model attribute.

- For the `string` based model attributes (i.e. text based database table columns) that don't allow `null` values—the `title`, `author`, `publisher` properties—the `notNull` and `notEmpty` validators are applied to disallow `null` values **and** empty string values.
- Notice the nuance between the `allowNull` model attribute property and the `notNull` validation rule. The `allowNull` model attribute property is set to `false` to configure the underlying database table column to disallow `null` values and the `notNull` validation rule is applied to validate that a model instance attribute value is not `null`.
- The `len` validation is also applied to the `string` based model attributes to give feedback to the end user when a model instance attribute value exceeds the configured maximum length for the underlying database table column.
- The `isDate` and `isInt` validators are applied respectively to the `releaseDate` and `pageCount` model attributes to validate that the model instance attribute values can be successfully parsed to the underlying database table column data types.

Sequelize provides a variety of validators that you can apply to model attributes. For a list of the available validators, see the official Sequelize documentation.

For more information about Sequelize model validations see the "Model Validations With Sequelize" article in the SQL ORM lesson.

Updating the Add Book page `POST` route

With the model validations in place, now you need to update the "Add Book" page `POST` route in the `routes` module (the `./routes.js` file) to process Sequelize validation errors.

To start, add the `next` parameter to the route handler function's parameter list:

```
router.post('/book/add', csrfProtection, asyncHandler(async
(req, res, next) => {
  // Code removed for brevity.
}));
```

Then update the `try/catch` statement to this:

```
try {
  await book.save();
  res.redirect('/');
} catch (err) {
  if (err.name === 'SequelizeValidationError') {
    const errors = err.errors.map((error) => error.message);
    res.render('book-add', {
      title: 'Add Book',
      book,
      errors,
      csrfToken: req.csrfToken(),
    });
  } else {
    next(err);
  }
}
```

Within the `catch` block, the `err.name` property is checked to see if the error is a `SequelizeValidationError` error type which is the error type that Sequelize throws if a validation error has occurred.
If it's a validation error, the `Array.map()` method is called on the `err.errors` array to create an array of error messages. Currently, `err` is an object with an `errors` property. The `err.errors` property is an array of *error objects* that provide detailed information about each validation error. Each element in `err.errors` has a `message` property.
The `Array.map()` method plucks the `message` property from each *error object* to create an array of validation messages. This array of validation messages will be rendered on the form, instead of the array of *error objects*.
If the error isn't a `SequelizeValidationError` error, then the error is passed as an argument to the `next()` method call which results in Express handing the request off to the application's defined error handlers for processing.
For your reference, the updated "Add Book" page `POST` route should now look like this:

```
router.post('/book/add', csrfProtection, asyncHandler(async
(req, res, next) => {
  const {
    title,
    author,
    releaseDate,
    pageCount,
    publisher,
  } = req.body;

  const book = db.Book.build({
    title,
    author,
    releaseDate,
    pageCount,
    publisher,
  });

  try {
    await book.save();
    res.redirect('/');
  } catch (err) {
    if (err.name === 'SequelizeValidationError') {
      const errors = err.errors.map((error) => error.message);
      res.render('book-add', {
        title: 'Add Book',
        book,
        errors,
        csrfToken: req.csrfToken(),
      });
    } else {
      next(err);
    }
  }
}));
```

Updating the Add Book page view

The final part of implementing validations is to update the "Add Book" page view (the `./views/book-add.pug` file) to render the array of validation messages. Replace the existing `if error` conditional statement with the following code:

```
//- ./views/book-add.pug

extends layout.pug

block content
  if errors
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      ul
        each error in errors
          li= error

//- Code removed for brevity.
```

The Bootstrap `alert alert-danger` CSS classes are used to style the unordered list of validation messages.

For your reference, the updated "Add Book" page view should now look like this:

```
//- ./views/book-add.pug

extends layout.pug

block content
  if errors
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      ul
        each error in errors
          li= error
  form(action='/book/add' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    div(class='form-group')
      label(for='title') Title
      input(type='text' id='title' name='title'
value=book.title class='form-control')
    div(class='form-group')
      label(for='author') Author
      input(type='text' id='author' name='author'
value=book.author class='form-control')
    div(class='form-group')
      label(for='releaseDate') Release Date
      input(type='text' id='releaseDate' name='releaseDate'
value=book.releaseDate class='form-control' placeholder='ex:
2000-01-31')
```

```
    div(class='form-group')
      label(for='pageCount') Page Count
      input(type='text' id='pageCount' name='pageCount'
value=book.pageCount class='form-control')
    div(class='form-group')
      label(for='publisher') Publisher
      input(type='text' id='publisher' name='publisher'
value=book.publisher class='form-control')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Add Book
      a(href='/' class='btn btn-warning ml-2') Cancel
```

Testing the server-side validations

Run the command `npm start` to start your application and browse to `http://localhost:8080/`. Click the "Add Book" button at the top of the "Book List" page to browse to the "Add Book" page. Click the "Add Book" button to submit the "Add Book" page form without providing any values. You should now see a list of validation messages displayed just above the form.

Provide a value for each of the form fields and click the "Add Book" button to submit the form to the server. You should now see your new book in the list of books on the "Book List" page!

Implementing server-side validation using a validation library

Keeping your application's validation logic out of your database models makes your code more modular. Improved modularity allows you to more easily update one part of your application without worrying as much about how that change will impact another part of your application.

In this section, you'll replace the Sequelize model validations with route level validations using the `express-validator` validation library.

Removing the Sequelize model validations

Before you updated the `Book` model (the `./db/models/book.js` file), you made a copy of the existing code by either copying the entire file with a file extension of `.bak` (i.e. `book.js.bak`) or copying and pasting the code within the existing file and commenting it out. It's time to use your backup copy of the `Book` model to remove the Sequelize validations.
For your reference, here's what the `Book` model (the `./db/models/book.js` file) should look like before proceeding:

```
// ./db/models/book.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Book = sequelize.define('Book', {
    title: {
      type: DataTypes.STRING,
      allowNull: false
    },
    author: {
      type: DataTypes.STRING(100),
      allowNull: false
    },
    releaseDate: {
      type: DataTypes.DATEONLY,
      allowNull: false
    },
    pageCount: {
      type: DataTypes.INTEGER,
      allowNull: false
    },
    publisher: {
      type: DataTypes.STRING(100),
      allowNull: false
    }
  }, {});
  Book.associate = function(models) {
    // associations can be defined here
  };
  return Book;
};
```

Updating the Add Book page `POST` route

From the terminal, use npm to install the `express-validator` package:

```
npm install express-validator@^6.0.0
```

In the `routes` module (i.e. the `./routes.js` file), use the `require()` function to import the `express-validator` module (just after importing the `csurf` module) and destructuring to declare and initialize the `check` and `validationResult` variables:

```
// ./routes.js

const express = require('express');
const csrf = require('csurf');
```

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

const db = require('./db/models');

// Code remove for brevity.
```

The `check` variable references a function (defined by the `express-validator` validation library) that returns a middleware function for validating a request. When you call the `check()` method, you pass in the name of the field—in this case a request body form field name—that you want to validate:

```
const titleValidator = check('title');
```

The value returned by the `check()` method is a validation chain object. The object is referred to as a validation "chain" because you can add one or more validators by making a series of method calls.

One of the validators that you can add to the validation chain is the `exists()` validator:

```
const titleValidator = check('title')
  .exists({ checkFalsy: true });
```

The `exists()` validator will fail if the request body is missing a form field with the name (or key) `title` or because we set the `checkFalsy` option to `true` the validator will fail if the request body contains a form field with the name `title` but the value is set to a falsy value (eg `""`, `0`, `false`, `null`).

When a validator fails, it'll add a validation error to the current request. You can chain a call to the `withMessage()` method to customize the validation error message for the previous validator in the chain:

```
const titleValidator = check('title')
  .exists({ checkFalsy: true })
  .withMessage('Please provide a value for Title');
```

Now if the `exists()` validator for the field `title` fails, a validation error will be added to the request with the message "Please provide a value for Title".

The `express-validator` validation library is built on top of the validator.js library. This means that all of the available validators within the validator.js library are available for you to use in your validation logic.

One of the available validators is the `isLength()` validator, which can be used to check the length of a string based field:

```
const titleValidator = check('title')
  .exists({ checkFalsy: true })
  .withMessage('Please provide a value for Title')
  .isLength({ max: 255 })
  .withMessage('Title must not be more than 255 characters
long');
```

Notice how the `isLength()` method is called directly on the return value of the `withMessage()` method? This is the validation chain in action—each method call in the validation chain returns the validation chain so you can keep adding validators. This is also known as "method chaining".

APIs that make use of method chaining are often referred to as fluent APIs.

Instead of declaring a variable for each field that you want to define a validation chain for, you can declare a single variable that's initialized to an array of validation chains:

```
const bookValidators = [
```

```
  check('title')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Title')
    .isLength({ max: 255 })
    .withMessage('Title must not be more than 255 characters
long'),
  check('author')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Author')
    .isLength({ max: 100 })
    .withMessage('Author must not be more than 100 characters
long'),
  check('releaseDate')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Release Date')
    .isISO8601()
    .withMessage('Please provide a valid date for Release
Date'),
  check('pageCount')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Page Count')
    .isInt({ min: 0 })
    .withMessage('Please provide a valid integer for Page
Count'),
  check('publisher')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Publisher')
    .isLength({ max: 100 })
    .withMessage('Publisher must not be more than 100
characters long'),
];
```

Each validation chain is an Express middleware function. After initializing an array containing all of your field validation chains, you can simply add the array directly to your route definition:

```
router.post('/book/add', csrfProtection, bookValidators,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));
```

Because each field validation chain is a middleware function and the Express Application `post()` method accepts an array of middleware functions, each validation chain will be called when the request matches the route path.

Within the route handler function, `validationResult()` function is used to extract any validation errors from the current request:

```
router.post('/book/add', csrfProtection, bookValidators,
  asyncHandler(async (req, res) => {
    const {
      title,
```

```
      author,
      releaseDate,
      pageCount,
      publisher,
    } = req.body;

    const book = db.Book.build({
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await book.save();
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('book-add', {
        title: 'Add Book',
        book,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));
```

The `validatorErrors` object provides an `isEmpty()` method to check if there are any validation errors. If there aren't any validation errors, then the `book.save()` method is called to persist the book to the database and the user is redirected to the default route (i.e. the "Book List" page).

If there are validation errors, the `array()` method is called on the `validatorErrors` object to get an array of validation error objects. Each error object has a `msg` property containing the validation error message. The `Array.map()` method plucks the `msg` property from each error object into a new array of validation messages named `errors`.

For more information about the `express-validator` library, see the official documentation.

For your reference, here's what the `./routes.js` file should look like after being updated:

```
// ./routes.js

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

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

const router = express.Router();

const csrfProtection = csrf({ cookie: true });

const asyncHandler = (handler) => (req, res, next) =>
handler(req, res, next).catch(next);

router.get('/', asyncHandler(async (req, res) => {
  const books = await db.Book.findAll({ order: [['title',
'ASC']] });
  res.render('book-list', { title: 'Books', books });
}));

router.get('/book/add', csrfProtection, (req, res) => {
  const book = db.Book.build();
  res.render('book-add', {
    title: 'Add Book',
    book,
    csrfToken: req.csrfToken(),
  });
});

const bookValidators = [
  check('title')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Title')
    .isLength({ max: 255 })
    .withMessage('Title must not be more than 255 characters
long'),
  check('author')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Author')
    .isLength({ max: 100 })
    .withMessage('Author must not be more than 100 characters
long'),
  check('releaseDate')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Release Date')
    .isISO8601()
    .withMessage('Please provide a valid date for Release
Date'),
  check('pageCount')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Page Count')
    .isInt({ min: 0 })
    .withMessage('Please provide a valid integer for Page
Count'),
```

```
  check('publisher')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Publisher')
    .isLength({ max: 100 })
    .withMessage('Publisher must not be more than 100
characters long'),
];

router.post('/book/add', csrfProtection, bookValidators,
  asyncHandler(async (req, res) => {
    const {
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    } = req.body;

    const book = db.Book.build({
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await book.save();
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('book-add', {
        title: 'Add Book',
        book,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));

module.exports = router;
```

Testing the updated server-side validations

Run the command `npm start` to start your application and browse to `http://localhost:8080/`. Click the "Add Book" button at the top of the "Book List" page to browse to the "Add Book" page. Click the "Add Book" button to submit the "Add Book" page form without providing any values. You should now see a list of validation messages displayed just above the form.

Provide a value for each of the form fields and click the "Add Book" button to submit the form to the server. You should now see your new book in the list of books on the "Book List" page!

Adding the Edit Book page

The next page that you'll add to the Reading List application is the "Edit Book" page. As the name clearly suggests, this page will allow you to edit the details of a book from the reading list.

Defining the routes for the Edit Book page

Add the routes for the "Edit Book" page to the `routes` module (i.e. the `./routes.js` file) just after the routes for the "Add Book" page—a`GET`route to initially retrieve the "Edit Book" page's HTML form and a`POST` route to process the page's HTML form submissions:

```
router.get('/book/edit/:id(\\d+)', csrfProtection,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);
    res.render('book-edit', {
      title: 'Edit Book',
      book,
      csrfToken: req.csrfToken(),
    });
  }));

router.post('/book/edit/:id(\\d+)', csrfProtection,
bookValidators,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const bookToUpdate = await db.Book.findByPk(bookId);
```

```
    const {
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    } = req.body;

    const book = {
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    };

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await bookToUpdate.update(book);
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('book-edit', {
        title: 'Edit Book',
        book: { ...book, id: bookId },
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));
```

Here's an overview of the above routes:

- Just like you did for the "Add Book" page, two routes are defined for the "Edit Book" page—a `GET` route and a `POST` route, both with a path of `/book/edit/:id(\\d+)`. The `:id(\\d+)` path segment defines the `id` property in your `req.params`, the route parameter to capture the book ID to edit. The `\\d+` segment uses regexp to ensure that only numbers (or **d**igits) will match this segment.
- Within both route handlers, the `parseInt()` function is used to convert the `req.params.id` property from a string into an integer.
- Within both route handlers, the Sequelize `db.Book.findByPk()` method uses the book ID to retrieve which book to edit from the database.
- Just like in the `/book/add` route, destructuring is used to declare and initialize the `title`, `author`, `releaseDate`, `pageCount`, and `publisher` variables from the `req.body` property. Those variables are then used to create a `book` object literal whose properties align with the `Book` model properties. If there aren't any validation

errors, the object literal is passed into the `book.update()` method to update the book in the database and the user is redirected to the default route `/`. If there are validation errors, the `book-edit` view is re-rendered with the validation errors.

When passing the `book` object into the `book-edit` view, you can use spread syntax to copy the `book` object literal properties into a new object. To the right of spreading the `book` object, an `id` property is declared and assigned to the `bookId` variable value:

```
book: {
  ...book,
  id: bookId
}
```

The spread syntax above actually creates this `book` object:

```
book: {
  title,
  author,
  releaseDate,
  pageCount,
  publisher,
  id: bookId
}
```

Creating the view for the Edit Book page

Add a view to the `views` folder named `book-edit.pug` containing the following code:

```
//- ./views/book-edit.pug

extends layout.pug

block content
  if errors
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      ul
        each error in errors
          li= error
  form(action=`/book/edit/${book.id}` method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    div(class='form-group')
      label(for='title') Title
      input(type='text' id='title' name='title'
value=book.title class='form-control')
    div(class='form-group')
      label(for='author') Author
      input(type='text' id='author' name='author'
value=book.author class='form-control')
    div(class='form-group')
```

```
      label(for='releaseDate') Release Date
      input(type='text' id='releaseDate' name='releaseDate'
value=book.releaseDate class='form-control' placeholder='ex:
2000-01-31')
    div(class='form-group')
      label(for='pageCount') Page Count
      input(type='text' id='pageCount' name='pageCount'
value=book.pageCount class='form-control')
    div(class='form-group')
      label(for='publisher') Publisher
      input(type='text' id='publisher' name='publisher'
value=book.publisher class='form-control')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Update Book
      a(href='/' class='btn btn-warning ml-2') Cancel
```

This view is almost the same as the view for the "Add Book" page. On the form element's `action` attribute and the submit button content are different.

In just a bit, you'll see how you can leverage features built into Pug to avoid unnecessary code duplication.

Testing the Edit Book page

Run the command `npm start` to start your application and browse to `http://localhost:8080/`. Click the "Edit" button for one of the books listed in the table on the "Book List" page to edit that book. Change one or more form field values and click the "Update Book" button to submit the form to the server. You should now see the update book in the list of books on the "Book List" page!

Including view templates for DRYer code

Currently, the "Add Book" and "Edit Book" views contain very similar code. Pug allows you to `include` the contents of a template within another template. You can use this feature to eliminate the code duplication between the `./views/book-add.pug` and `./views/book-edit.pug` files.
Start by adding a new file named `book-form-fields.pug` to the `views` folder containing the following code:

```
//- ./views/book-form-fields.pug

input(type='hidden' name='_csrf' value=csrfToken)
div(class='form-group')
  label(for='title') Title
```

```
  input(type='text' id='title' name='title' value=book.title
class='form-control')
div(class='form-group')
  label(for='author') Author
  input(type='text' id='author' name='author'
value=book.author class='form-control')
div(class='form-group')
  label(for='releaseDate') Release Date
  input(type='text' id='releaseDate' name='releaseDate'
value=book.releaseDate class='form-control' placeholder='ex:
2000-01-31')
div(class='form-group')
  label(for='pageCount') Page Count
  input(type='text' id='pageCount' name='pageCount'
value=book.pageCount class='form-control')
div(class='form-group')
  label(for='publisher') Publisher
  input(type='text' id='publisher' name='publisher'
value=book.publisher class='form-control')
```

Then update the `book-add.pug` and `book-edit.pug` views to the following code:

```
// ./views/book-add.pug

extends layout.pug

block content
  if errors
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      ul
        each error in errors
          li= error
  form(action='/book/add' method='post')
    include book-form-fields.pug
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Add Book
      a(href='/' class='btn btn-warning ml-2') Cancel
//- ./views/book-edit.pug

extends layout.pug

block content
  if errors
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      ul
        each error in errors
          li= error
  form(action=`/book/edit/${book.id}` method='post')
    include book-form-fields.pug
    div(class='py-4')
```

```
      button(type='submit' class='btn btn-primary') Update Book
      a(href='/' class='btn btn-warning ml-2') Cancel
```

Notice the use of the `include` keyword to include the contents of the `book-form-fields.pug` template.

Another Pug feature—mixins—allows you to create reusable blocks of Pug code. You can use this Pug feature to further eliminate code duplication.

Add a new file named `utils.pug` to the `views` folder containing the following code:

```
//- ./views/utils.pug

mixin validationErrorSummary(errors)
  if errors
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      ul
        each error in errors
          li= error
```

Notice that the `validationErrorSummary` mixin defines an `errors` parameter. As you might expect, mixin parameters allow you to pass data into the mixin.
Next, update the `book-add.pug` and `book-edit.pug` views to the following code:

```
// ./views/book-add.pug

extends layout.pug

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action='/book/add' method='post')
    include book-form-fields.pug
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Add Book
      a(href='/' class='btn btn-warning ml-2') Cancel
//- ./views/book-edit.pug

extends layout.pug

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action=`/book/edit/${book.id}` method='post')
    include book-form-fields.pug
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Update Book
      a(href='/' class='btn btn-warning ml-2') Cancel
```

Notice the use of the `include` keyword again to include the contents of the `utils.pug` template which makes the `validationErrorSummary` mixin available

within the `book-add.pug` and `book-edit.pug` templates. The mixin is called by prefixing the mixin name with a plus sign (`+`) and adding a set of parentheses after the mixin name. Inside of the parentheses, the `errors` variable is passed as an argument to the `validationErrorSummary` mixin.

You can go a bit further to eliminate more code duplication. Update the `./views/utils.pug` template to contain the following code:

```
//- ./views/utils.pug

mixin validationErrorSummary(errors)
  if errors
    div(class='alert alert-danger' role='alert')
      p The following error(s) occurred:
      ul
        each error in errors
          li= error

mixin textField(labelText, fieldName, fieldValue, placeholder)
  div(class='form-group')
    label(for=fieldName)= labelText
    input(type='text' id=fieldName name=fieldName
value=fieldValue class='form-control' placeholder=placeholder)
```

Then update the `./views/book-form-fields.pug` template to contain this code:

```
//- ./views/book-form-fields.pug

include utils.pug

input(type='hidden' name='_csrf' value=csrfToken)
+textField('Title', 'title', book.title)
+textField('Author', 'author', book.author)
+textField('Release Date', 'releaseDate', book.releaseDate,
'ex: 2000-01-31')
+textField('Page Count', 'pageCount', book.pageCount)
+textField('Publisher', 'publisher', book.publisher)
```

Run the command `npm start` to start your application and browse to `http://localhost:8080/`. Use the "Add Book" page to add a new book and then use the "Edit Book" page to edit the book. Everything should work as it did before the refactoring of the view code.

Congratulations on making your code DRYer!

Add the Delete Book page

The next page that you'll add to the Reading List application is the "Delete Book" page. This page is relatively simple as it only needs to prompt the user if the selected book is the book that they want to delete.

Defining the routes for the Delete Book page

Add the routes for the "Delete Book" page to the `routes` module (i.e. the `./routes.js` file) just after the routes for the "Edit Book" page—a `GET` route to initially retrieve the "Delete Book" page's HTML form and a `POST` route to process the page's HTML form submissions:

```
router.get('/book/delete/:id(\\d+)', csrfProtection,
asyncHandler(async (req, res) => {
  const bookId = parseInt(req.params.id, 10);
  const book = await db.Book.findByPk(bookId);
  res.render('book-delete', {
    title: 'Delete Book',
    book,
    csrfToken: req.csrfToken(),
  });
}));

router.post('/book/delete/:id(\\d+)', csrfProtection,
asyncHandler(async (req, res) => {
  const bookId = parseInt(req.params.id, 10);
  const book = await db.Book.findByPk(bookId);
  await book.destroy();
  res.redirect('/');
}));
```

Here's an overview of the above routes:

- Just like you did for the "Add Book" and "Edit Book" pages, two routes are defined for the "Delete Book" page—a `/book/delete/:id(\\d+) GET` route and a `/book/delete/:id(\\d+) POST` route.
- Within both route handlers, the `parseInt()` function is used to convert the `req.params.id` property string value into a `number`.
- Within both route handlers, the Sequelize `db.Book.findByPk()` method is used to retrieve the book to delete from the database.
- Within the `POST` route handler, the `book.destroy()` method is called to delete the book from the database and the user is redirected to the default route (`/`).

Creating the view for the Delete Book page

Add a view to the `views` folder named `book-delete.pug` containing the following code:

```
//- ./views/book-delete.pug

extends layout.pug
```

```
block content
  h3= book.title
  div(class='py-4')
    p Proceed with deleting this book?
  div
    form(action=`/book/delete/${book.id}` method='post')
      input(type='hidden' name='_csrf' value=csrfToken)
      button(class='btn btn-danger' type='submit') Delete Book
      a(class='btn btn-warning ml-2' href='/' role='button')
Cancel
```

The purpose of this view is simple: display the title of the book that's about to be deleted and render a simple form containing a hidden `<input>` element for the CSRF token and a `<button>` element to submit the form.

Testing the Delete Book page

Run the command `npm start` to start your application and browse to `http://localhost:8080/`. Click the "Delete" button for one of the books listed in the table on the "Book List" page to delete that book. On the "Delete Book" page, click the "Delete Book" button to delete the book. You should now see that the book has been removed from the list of books on the "Book List" page!

What you learned

In this article, you learned how to:

- Define a collection of routes and views that use Sequelize to perform CRUD operations against a single resource; and
- Handle Sequelize validation errors when users are attempting to create or update data and display error messages to the user so that they can resolve any data quality issues.

You also reviewed the following:

- Using a wrapper function to catch errors thrown within asynchronous route handler functions;
- Using Pug to create HTML forms;
- Using the `csurf` middleware to protect against CSRF exploits;
- Using the built-in `express.urlencoded()` middleware function to parse incoming request body form data;
- Using Sequelize model validations to validate user-provided data;

- Using the `express-validator` validation library to validate user-provided data within an Express route; and
- Using Pug includes and mixins to remove unnecessary code duplication.

Did you find this lesson helpful?

No

Yes

✓

Continue To Next Page →