Code Review #1

tests

- naming conventions should be consistent. if file is "Books.model", variable should be BooksModel, not BookModel

- this test does not really do anything tests-related. no assertions/expects are used and nothing is tested

- unit tests should be isolated from outside and be mock-based, so no actual requests is sent.
- POST request (book creation) returns us json of { status: "ok" } if everything went well. we can rely on that response to check whether a new book was added or not, without additional request.

```
27  it("book has necessary properties", async () => {
28    const books = await bookModel.getAll();
29
30    let passed = true;
31
32    books.forEach((book) => {
33      passed = "id" in book;
34      passed = "name" in book;
35      passed = "ownerId" in book;
36      passed = "author" in book;
37    });
38
39    expect(passed).toBe(true);
40  });
41
```

- such assumption can easily be false, because forEach does not break if some values do not exist. further iterations will just rewrite "passed" variable. Instead, we can compare a sorted array of ['id', 'name', 'ownerId', 'author'] with sorted array of Object.keys(book). If all values are equal, we can jump to next iteration until false.

---

env file

- it is considered a good practice to use dotenv to store such variables as it is more versatile and supports multiple environments out of the box

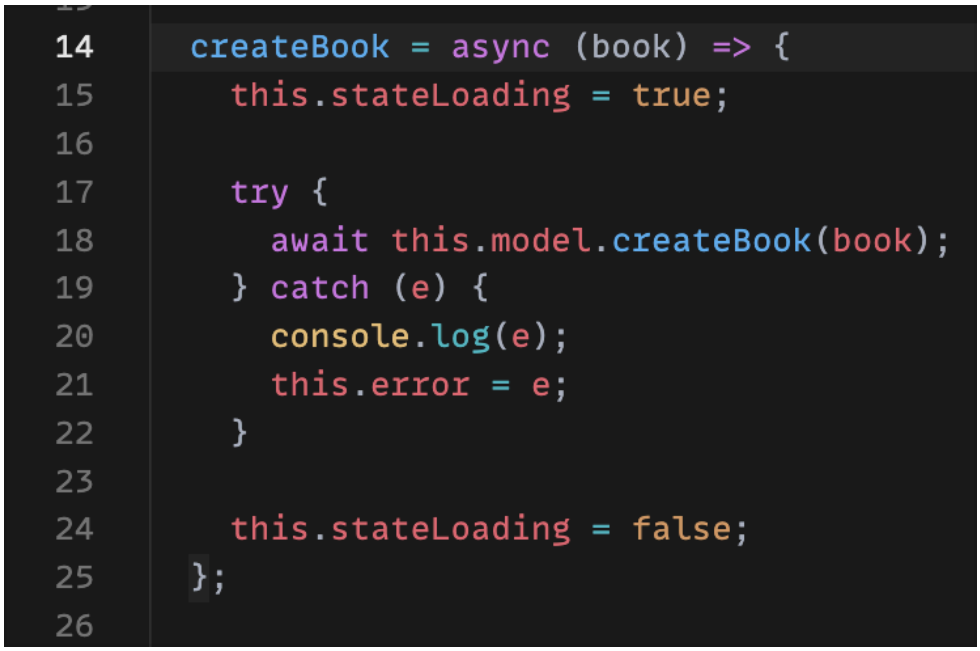
---

http file

- naming does not really explain the purpose of the file. api or ApiGateway would make much more sense, as we use it to make API calls

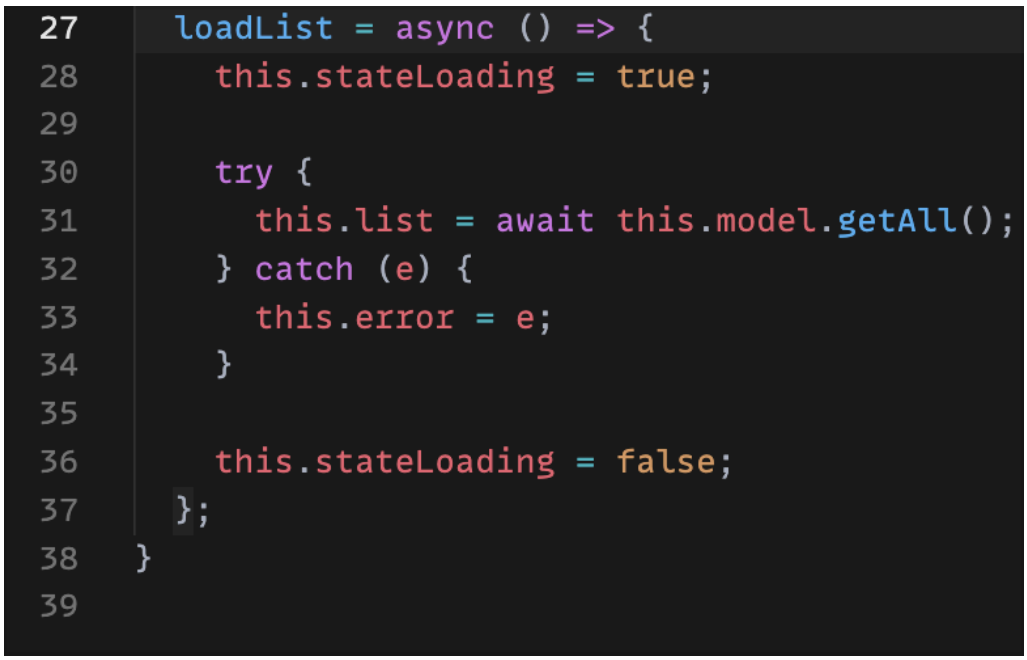
---

Books.ctrl.js



```
14    createBook = async (book) => {
15      this.stateLoading = true;
16
17      try {
18        await this.model.createBook(book);
19      } catch (e) {
20        console.log(e);
21        this.error = e;
22      }
23
24      this.stateLoading = false;
25    };
26
```

- according to guidelines, console.logs should be removed
- for async actions, runInAction should be used for next tick's mutations (in this case, lines 21 and 24 should be wrapped in runInAction)



```
27    loadList = async () => {
28      this.stateLoading = true;
29
30      try {
31        this.list = await this.model.getAll();
32      } catch (e) {
33        this.error = e;
34      }
35
36      this.stateLoading = false;
37    };
38  }
39
```

- runInAction should wrap lines 33 and 36 as well, according to guidelines

---

index.js

```
  const handleOnClick = async () => {
    await controller.createBook({
      id: 1914,
      ownerId: 2022,
      name: "The First World War",
      author: "People"
    });
    await controller.loadList();
  };
```

- considering we pass static data to create a book, we can move it to a controller to minimize code in App Component.
- we can seamlessly update client-side without additional request (loadList) — createBook returns { status: "ok" } if succeeds. knowing this, we can push new book's data to controller.list variable and only modify books list

```
  if (controller.stateLoading) {
    return <>Loading...</>;
  }

  if (controller.error) {
    return (
      <>
        <div>The error bellow has happend</div>
        <div className="error">{controller.error.toString()}</div>
      </>
    );
  }

  return (
    <div>
      {controller.list.map((book, i) => (
        <div key={i}>
          {book.author}: {book.name}
        </div>
      ))}
      <button onClick={handleOnClick}>Add</button>
    </div>
  );
}

```

- this logic should be separated, because as soon as the state changes, we

are not able to see the other parts of content
- index as key is not a good practice, use id or other book-related data instead
- semantically, lists should be presented as unordered lists, not divs