BookVerse – Day 4 Documentation Report

This report documents the Day 4 enhancements to the BookVerse React application, focused on form handling, validation, and Flux architecture implementation. The Day 4 update introduces the ability to add new books using Formik and Yup, managing data flow via a Flux pattern (Dispatcher → Actions → Store → Components).

1. Project Overview

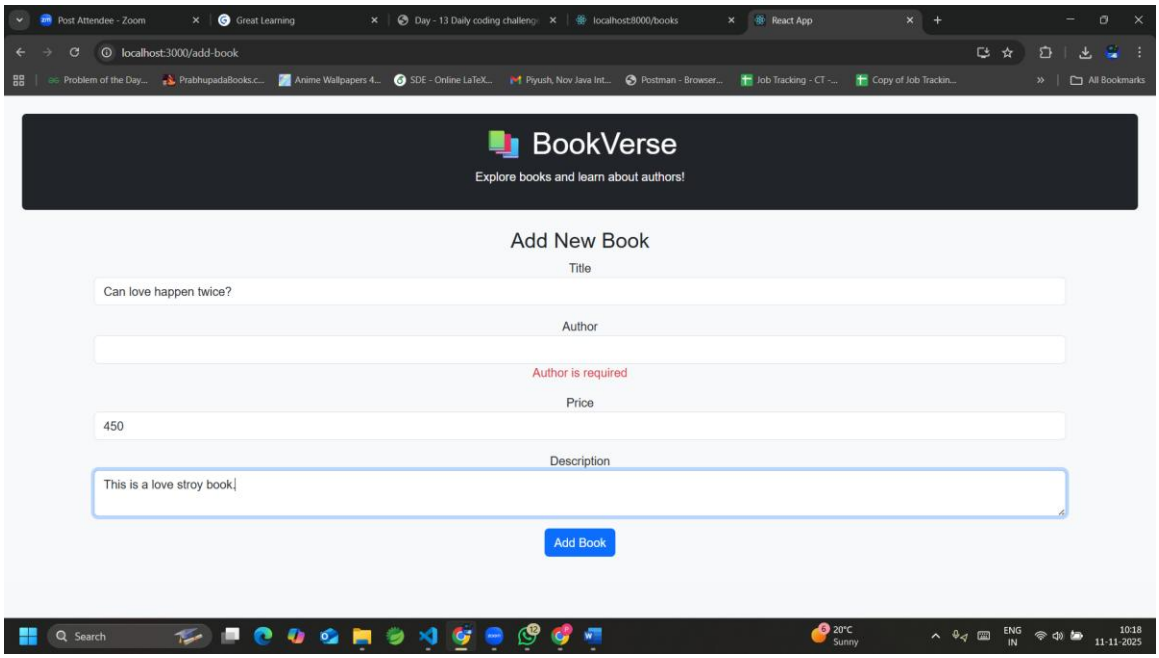
Title: BookVerse – Add New Book and Manage Data Flow

Objective: Implement a Flux architecture with form handling for adding books, ensuring unidirectional data flow and SPA behavior without page reloads.

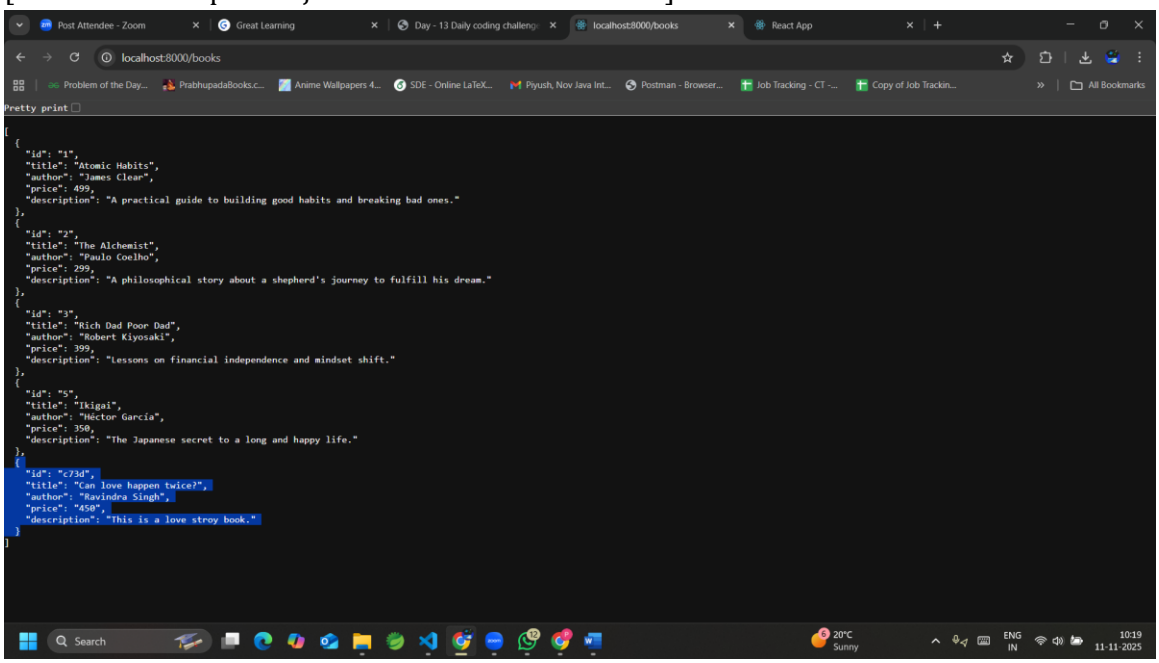
Key Additions:
- Formik and Yup for form handling and validation
- Flux pattern with Dispatcher, Store, and Actions
- Dependency Injection for modular store management
- SPA lifecycle and smooth data flow between views

2. Output Screenshots (Placeholders)

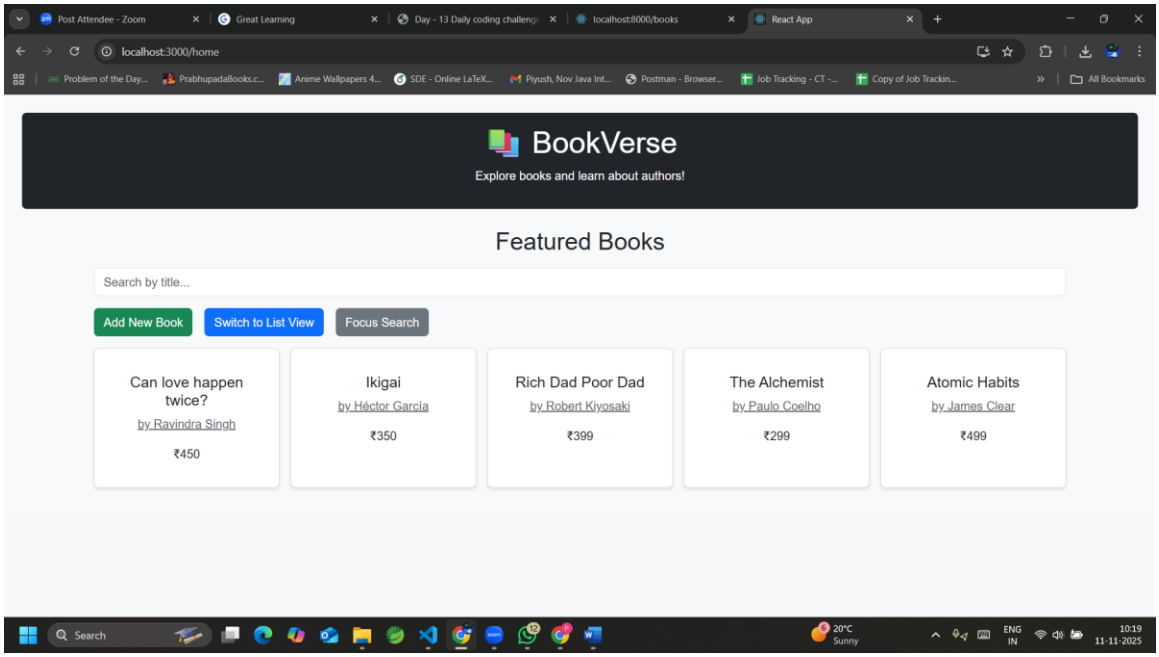
[Screenshot 1: Add Book Form Placeholder]



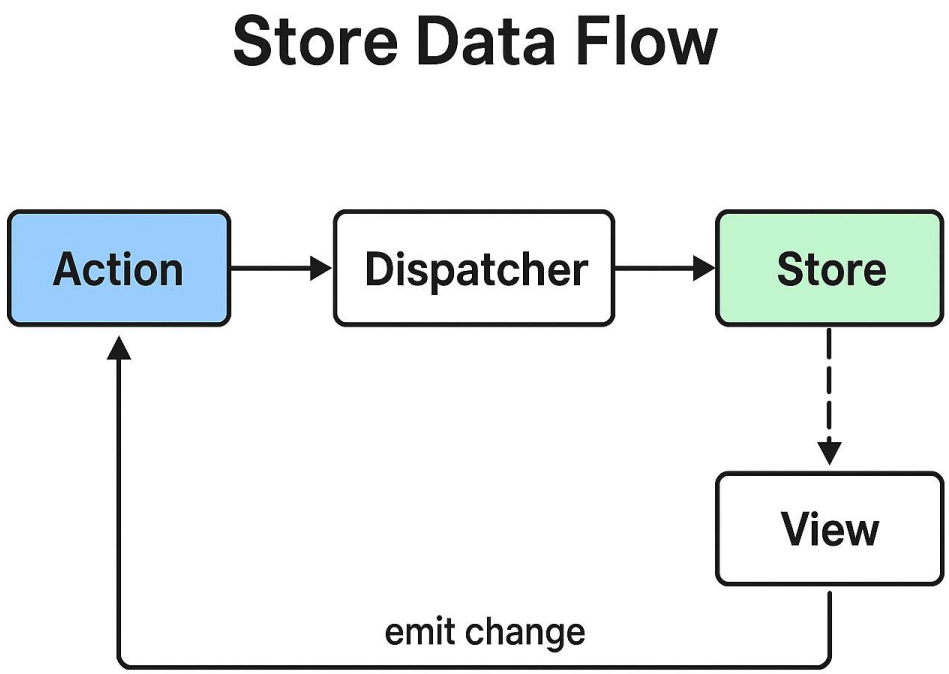
[Screenshot 2: Updated json-server Data Placeholder]



[Screenshot 3: Home Page After Adding Book Placeholder]



[Screenshot 4: Store Data Flow Diagram Placeholder]



3. Updated & New Components (with Explanations)

3.1 App.jsx – Updated Routing

New routes `/add-book` and `/edit/:id` have been added to handle the book form for adding and editing.

```
// App.jsx - updated to include Day 4 routes (Add/Edit)
import React, { useState } from 'react';
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-
router-dom';
import BookList from './components/BookList';
import BookDetails from './components/BookDetails';
import AuthorInfo from './components/AuthorInfo';
import BookForm from './components/BookForm';
import 'bootstrap/dist/css/bootstrap.min.css';
import './App.css';

const App = () => {
  const [loading, setLoading] = useState(false);

  return (
    <Router>
      <div className="App bg-light p-4 fade-in">
```

```
        <header className="bg-dark text-white p-3 rounded mb-4">
          <h1>📚 BookVerse</h1>
          <p>Explore books and learn about authors!</p>
        </header>

        <Routes>
          <Route path="/" element={<Navigate to="/home" />} />
          <Route path="/home" element={<BookList loading={loading}
setLoading={setLoading} />} />
          <Route path="/book/:id" element={<BookDetails loading={loading}
setLoading={setLoading} />} />
          <Route path="/author/:authorName" element={<AuthorInfo />} />
          <Route path="/add-book" element={<BookForm mode="add" />} />
          <Route path="/edit/:id" element={<BookForm mode="edit" />} />
        </Routes>
      </div>
    </Router>
  );
};

export default App;
```

Explanation: Adds navigation for the new form component to enable Add and Edit functionality.

3.2 BookForm.jsx – Formik + Yup Form Handling

The BookForm component handles both 'Add' and 'Edit' modes. Formik manages form state while Yup enforces validation.

```
// BookForm.jsx - Add/Edit book form using Formik + Yup
import React from 'react';
import { Formik, Form, Field, ErrorMessage } from 'formik';
import * as Yup from 'yup';
import { useNavigate, useParams } from 'react-router-dom';
import apiClient from '../utils/apiClient';
import { addBook } from '../flux/actions';
import dispatcherInstance from '../flux/di'; // DI singleton

const BookSchema = Yup.object().shape({
    title: Yup.string().required('Title is required'),
    author: Yup.string().required('Author is required'),
    price: Yup.number().typeError('Price must be a
number').required('Price is required'),
    description: Yup.string(),
});
```

```
const BookForm = ({ mode = 'add' }) => {
    const navigate = useNavigate();
    const { id } = useParams();

    const initialValues = { title: '', author: '', price: '', description:
'' };

    // If edit mode, load current book data
    const [initial, setInitial] = React.useState(initialValues);

    React.useEffect(() => {
        if (mode === 'edit' && id) {
            apiClient.get(`/books/${id}`).then((data) => {
                setInitial({
                    title: data.title || '',
                    author: data.author || '',
                    price: data.price || '',
                    description: data.description || '',
                });
            }).catch((err) => {
                console.error(err);
            });
        }
    }, [mode, id]);

    return (
        <div className="container mt-4">
            <h3>{mode === 'add' ? 'Add New Book' : 'Edit Book'}</h3>
            <Formik
                enableReinitialize
                initialValues={initial}
                validationSchema={BookSchema}
                onSubmit={async (values, { setSubmitting, resetForm,
setErrors }) => {
                    try {
                        if (mode === 'add') {
                            // send to backend and dispatch action
                            const created = await
addBook(dispatcherInstance, values);
                            setSubmitting(false);
                            resetForm();
                            navigate('/home');
                        } else {
                            // edit book using API
```

```
                            const updated = await
apiClient.put(`/books/${id}`, { ...values, id: Number(id) });
                            // Optionally you could dispatch an UPDATE
action - skipping for brevity
                            setSubmitting(false);
                            navigate('/book/' + id);
                        }
                    } catch (err) {
                        console.error(err);
                        setSubmitting(false);
                        setErrors({ submit: 'Failed to save book' });
                    }
                }}
            >
                {({ isSubmitting }) => (
                    <Form>
                        <div className="mb-3">
                            <label>Title</label>
                            <Field name="title" className="form-control"
/>
                            <div className="text-danger"><ErrorMessage
name="title" /></div>
                        </div>
                        <div className="mb-3">
                            <label>Author</label>
                            <Field name="author" className="form-control"
/>
                            <div className="text-danger"><ErrorMessage
name="author" /></div>
                        </div>
                        <div className="mb-3">
                            <label>Price</label>
                            <Field name="price" className="form-control"
/>
                            <div className="text-danger"><ErrorMessage
name="price" /></div>
                        </div>
                        <div className="mb-3">
                            <label>Description</label>
                            <Field as="textarea" name="description"
className="form-control" />
                        </div>
                        <button type="submit" className="btn btn-primary"
disabled={isSubmitting}>
                            {mode === 'add' ? 'Add Book' : 'Save Changes'}
```

```
                        </button>
                        <div className="text-danger mt-2"><ErrorMessage
name="submit" component="div" /></div>
                    </Form>
                )}
            </Formik>
        </div>
    );
};

export default BookForm;
```

Explanation: Uses Formik to simplify input handling and Yup for client-side validation. On submission, the dispatcher triggers a Flux action to update the store.

3.3 Dispatcher.js – Central Action Dispatcher

Dispatcher acts as the hub for all actions in the Flux architecture. It broadcasts actions to registered stores.

```
// Dispatcher.js - Simple Flux dispatcher
export default class Dispatcher {
    constructor() {
        this._callbacks = {};
        this._lastId = 1;
    }

    register(callback) {
        const id = 'ID_' + this._lastId++;
        this._callbacks[id] = callback;
        return id;
    }

    unregister(id) {
        delete this._callbacks[id];
    }

    dispatch(action) {
        Object.values(this._callbacks).forEach((cb) => cb(action));
    }
}
```

Explanation: The dispatcher ensures that all stores receive actions, maintaining unidirectional data flow.

3.4 actions.js – Flux Action Creators

Defines actions for initializing and adding books. Dispatches API results to the store.

```
// actions.js - Action creators for BookVerse
import apiClient from '../utils/apiClient';

const Actions = {
    INIT_BOOKS: 'INIT_BOOKS',
    ADD_BOOK: 'ADD_BOOK',
    ADD_BOOK_SUCCESS: 'ADD_BOOK_SUCCESS',
    ADD_BOOK_FAIL: 'ADD_BOOK_FAIL',
    // other actions could be listed here
};

const addBook = async (dispatcher, bookData) => {
    dispatcher.dispatch({ type: Actions.ADD_BOOK, payload: bookData });
    try {
        const created = await apiClient.post('/books', bookData);
        dispatcher.dispatch({ type: Actions.ADD_BOOK_SUCCESS, payload:
created });
        return created;
    } catch (err) {
        dispatcher.dispatch({ type: Actions.ADD_BOOK_FAIL, payload: {
error: err } });
        throw err;
    }
};

const initBooks = (dispatcher, books) => {
    dispatcher.dispatch({ type: Actions.INIT_BOOKS, payload: books });
};

export { Actions, addBook, initBooks };
```

Explanation: Action creators encapsulate business logic and API interaction before notifying the store.

3.5 Store.js – Flux Data Store

The store maintains application state and listens to actions from the dispatcher. When updated, it notifies components.

```
// Store.js - Flux store implementation with dependency injection
import { Actions } from './actions';

export default class Store {
    constructor({ dispatcher, apiBase = 'http://localhost:8000' } = {}) {
        if (!dispatcher) throw new Error('Dispatcher instance required');
        this._dispatcher = dispatcher;
```

```
        this._books = [];
        this._listeners = [];
        this._dispatchToken =
this._dispatcher.register(this._onDispatch.bind(this));
        this.apiBase = apiBase;
    }

    _onDispatch(action) {
        switch (action.type) {
            case Actions.INIT_BOOKS:
                this._books = Array.isArray(action.payload) ?
action.payload : [];
                this.emitChange();
                break;
            case Actions.ADD_BOOK:
                // optimistic update could be added
                break;
            case Actions.ADD_BOOK_SUCCESS:
                this._books = [action.payload, ...this._books];
                this.emitChange();
                break;
            case Actions.ADD_BOOK_FAIL:
                // could set error state
                break;
            default:
                break;
        }
    }

    getBooks() {
        return this._books.slice();
    }

    addChangeListener(cb) {
        if (typeof cb === 'function') this._listeners.push(cb);
    }

    removeChangeListener(cb) {
        this._listeners = this._listeners.filter(l => l !== cb);
    }

    emitChange() {
        this._listeners.forEach((l) => {
            try { l(); } catch (e) { console.error(e); }
        });
```

```
    }
}
```

Explanation: The store updates its internal data when a new book is added and triggers subscribed components to re-render.

3.6 BookList.jsx – Store Subscription

BookList subscribes to store changes, re-rendering when the store updates after an action.

```
import React, { useState, useEffect, useRef } from "react";
import BookCard from "./BookCard";
import SearchBox from "./SearchBox";
import RenderMessage from "./RenderMessage";
import withLoader from "./withLoader";
import "bootstrap/dist/css/bootstrap.min.css";
import { storeInstance } from '../flux/di';
import { Link } from 'react-router-dom';

const BookList = ({ loading, setLoading }) => {
    const [books, setBooks] = useState([]);
    const [layout, setLayout] = useState("grid");
    const [searchTerm, setSearchTerm] = useState("");
    const searchRef = useRef(null);

    useEffect(() => {
        // initialize from current store contents
        setBooks(storeInstance.getBooks());
        // subscribe to store changes
        const onChange = () => setBooks(storeInstance.getBooks());
        storeInstance.addChangeListener(onChange);

        // fetch initial data from backend (and populate store)
        setLoading(true);
        fetch("http://localhost:8000/books")
            .then((res) => res.json())
            .then((data) => {
                // dispatch INIT via direct store injection for simplicity
                // we don't import actions here to avoid circular deps;
use store directly
                if (Array.isArray(data)) {
                    // mutate store by dispatching INIT through its
dispatcher
                    // storeInstance will be updated through Store API;
but simplest is to call storeInstance._books = data;
storeInstance.emitChange();
```

```
                    storeInstance._books = data.slice().reverse(); //
latest first
                    storeInstance.emitChange();
                }
                setLoading(false);
            }).catch((err) => {
                console.error('Error fetching books', err);
                setLoading(false);
            });

        return () => {
            storeInstance.removeChangeListener(onChange);
        };
    }, [setLoading]);

    const toggleLayout = () => setLayout((l) => (l === "grid" ? "list" :
"grid"));
    const focusSearch = () => searchRef.current &&
searchRef.current.focus();

    const filteredBooks = books.filter((book) =>
        book.title.toLowerCase().includes(searchTerm.toLowerCase())
    );

    return (
        <div className="container">
            <h2 className="mt-4">Featured Books</h2>
            <SearchBox searchTerm={searchTerm}
setSearchTerm={setSearchTerm} inputRef={searchRef} />
            <div className="d-flex gap-2 mb-3">
                <Link to="/add-book" className="btn btn-success me-2">Add
New Book</Link>
                <button className="btn btn-primary"
onClick={toggleLayout}>
                    Switch to {layout === "grid" ? "List" : "Grid"} View
                </button>
                <button className="btn btn-secondary ms-2"
onClick={focusSearch}>Focus Search</button>
            </div>


            {filteredBooks.length === 0 && (
                <RenderMessage render={() => "No books found. Try a
different search!"} />
            )}
```

```
            <div className={`book-list ${layout === "grid" ? "grid" : "list"}`}>
                {filteredBooks.map((book) => (
                    <BookCard
                        key={book.id}
                        id={book.id}
                        title={book.title}
                        author={book.author}
                        price={book.price}
                        layout={layout}
                    />
                ))}
            </div>
        </div>
    );
};

export default withLoader(BookList);
```

Explanation: Subscribes to Flux store updates and displays the latest books list in real-time.

4. Data Flow (Flux Pattern)

1. **BookForm** collects data → validates via Formik/Yup → triggers `*addBook*` action.
2. **Action** dispatches event → performs API POST → dispatches success.
3. **Dispatcher** broadcasts `*ADD_BOOK_SUCCESS*` → Store listens.
4. **Store** updates `*_books*` → emits change.
5. **BookList** hears the change → React re-renders instantly.
6. Navigation returns to `*/home*` showing the new book (SPA behavior).

5. Summary

The Day 4 enhancement implements complete Flux-based data management, form validation, and modular dependency injection. This ensures predictable data flow, centralized state management, and smooth SPA interactions.