Python-based Web Application for Library Management Using Docker, and Kubernetes, Deployed on Google Cloud

Group FE04: Mansi Patel, Vaishali Jadon, Atiya Azeez, Astha Patel

TABLE OF CONTENTS

Microservice Architecture

Similar architecture to 692

OUR MICROSERVICES

Customer/Manager authentication	Handles the authentication and authorization of customers and managers. It is responsible for verifying the identities of customers and managers
Catalog Management	This service manages the catalog collection of books and materials
Notification Service	This microservice handles the sending of email notifications to customers and library staff.
Account Management	This service allows users and managers to update their profile information, add, or delete an account.
Search books	This microservice provides the users with a set of filters to set and search for books.
Reservation	Books on hold and waitlist and the queue for each item
Review and Rating (future work)	Users can leave reviews and ratings under each item
My Library	Users can view their current and pending reservations, their position in the queue, and saved wishlist items

Technology Stack

Frontend, Backend, Middleware, Database and Deployment.

TECHNOLOGY STACK

DEVELOPMENT	TECHNOLOGY
FRONTEND	HTML, CSS, JAVASCRIPT, JINJA2
BACKEND	PYTHON, FASTAPI
MIDDLEWARE	CUSTOM FASTAPI MIDDLEWARE, INGRESS
DATABASE	MONGODB
DEPLOYMENT	DOCKER, KUBERNETES, GOOGLE CLOUD

Frontend

HTML, CSS, and Javascript

HTML

- Create static views.
- Organize the content of each page.
- Used Bootstrap framework.

CSS

- Custom styling of elements (buttons, borders, and images).
- Create a cohesive aesthetic of our LMS.

Javascript

- Dynamic content.
- Creating custom functions for events.
- Retrieving data to display from backend.

Backend + Middleware

Python, Fast API and Custom Middleware

FASTAPI

- Create backend logic function
- Develop API endpoints for routing and database queries

MongoDB

- Generate Pydantic BaseModels for database tables
- Perform CRUD operations using MongoDB and Python queries

Ingress

- Routing between microservices internally
- Microservices are exposed internally as ClusterIPs

Python

What is Python?

- Python is a high-level, interpreted language
- Simple and Readable Syntax
- Open Source & Cross-Platform

Why Python for Library Management?

- Ease of Development
- Efficient Backend Development with Python
- Efficient Database Management
- Security & Authentication

Python

- Uses a clean and concise syntax
- Interpreted language – No need for compilation, just run the script!
- Supports both SQL (MySQL, PostgreSQL) and NoSQL (MongoDB, Firebase) databases.
- Python is lightweight and flexible
- Easier to deploy using Heroku, AWS, or Docker.

VS

Java

- Uses verbose syntax, requiring more boilerplate code
- Compiled language – Requires writing, compiling, which slows down the development process.
- Java also supports databases, but JDBC requires complex configurations.
- Java frameworks are heavier and slower to get started.
- Java applications are heavier and require more memory & CPU.

FAST API IMPLEMENTATION

Implement required packages for FastAPI

Create a ‘router’ and implement each microservice

Connect all microservice API together in an ‘app’

Run the Fast API server: ‘uvicorn catalog_main:app’

```
from fastapi import
FastAPI, APIRouter
```

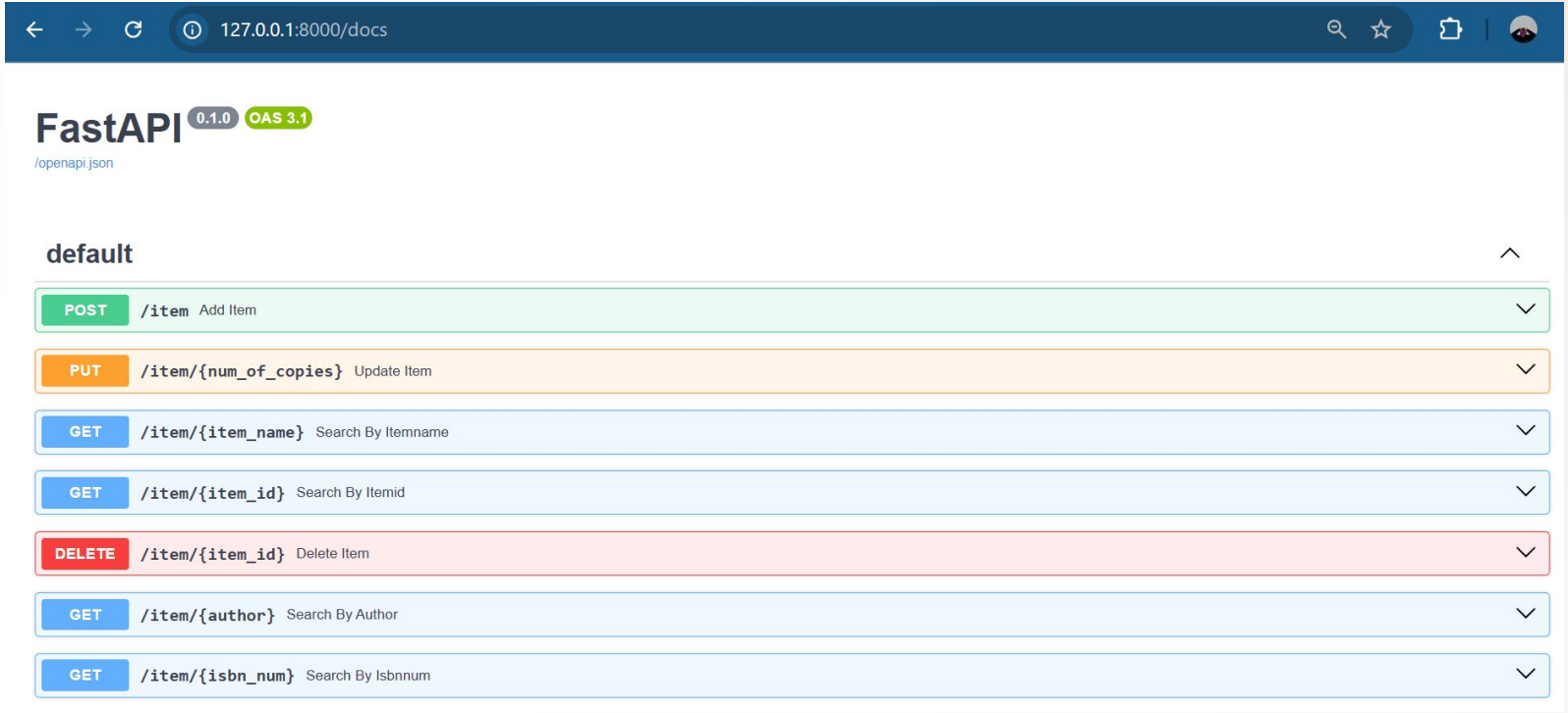
```
router = APIRouter()
# routes
# create an item
@router.post("/item",
response_model=catalog.CatalogItem)
def add_item(item: catalog.CatalogItem):
    catalog_db.add_item_db(item)
    return item
```

```
from fastapi import FastAPI
import uvicorn
from controller.catalog_controller import router
app = FastAPI()
app.include_router(catalog_controller.router,
prefix = "/catalogManagement")
```

```
uvicorn catalog_main:app
--reload
```

FAST API INTERACTIVE UI

127.0.0.1:8000/docs



Advantages of using FAST API for Backend

Automatic Validation

- Using **Pydantic** and **Python type hints**, Fast API is able to validate the type of data sent to the APIs
- Saves time from writing extra code to validate input types

Auto-Generated API Documentation

- Creates interactive doc for all API methods created
- Test and explore API easily
- Updates the interactive doc as the code updates

Built-in Middleware

- Provides built-in middleware for common tasks without needing extra libraries
- **CORS** middleware allows APIs to work with frontend and prevents getting blocked in your browsers

High Performance

- Built on **Asynchronous Server Gateway Interface (ASGI)** , able to process multiple requests
- Handles **real-time** requests
- Manages **a lot** of users

MIDDLEWARE

Middleware is a checkpoint that every request and response passes through. It allows us to modify requests before they reach the main logic and modify responses before they are sent back to the client.

CORS MIDDLEWARE

- Controls who (which domains) can access your API
- Cross-Origin Resource Sharing (CORS) middleware is used when your API needs to be access by other domains (microservices for our case).

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://127.0.0.1:8001","http://127.0.0.1:8002"], # Allow all
origins
    allow_credentials=True,
    allow_methods=["GET", "POST", "PUT", "DELETE"], # Allow all HTTP methods
    allow_headers=[""], # Allow all headers
)
```

FAST API'S CUSTOM MIDDLEWARE

- Controls how requests & responses behave inside FastAPI.

```python
class CustomMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request: Request, call_next):
        print("Custom middleware: Before request processing")
        response = await call_next(request)
        print("Custom middleware: After request processing")
        return response
app.add_middleware(CustomMiddleware)
```

```
Custom middleware: Before request processing
Custom middleware: After request processing
INFO:     127.0.0.1:51193 - "GET /search/popular HTTP/1.1" 200 OK
Custom middleware: After request processing
INFO:     127.0.0.1:51189 - "GET /search/newest HTTP/1.1" 200 OK
Custom middleware: Before request processing
```

Database Selection: NoSQL vs SQL

SQL

- Use structured schemas with predefined tables consisting of rows and columns
- Requires a fixed schema defined before data insertion.
- Best for complex relationships, as it uses JOIN operations to link tables.

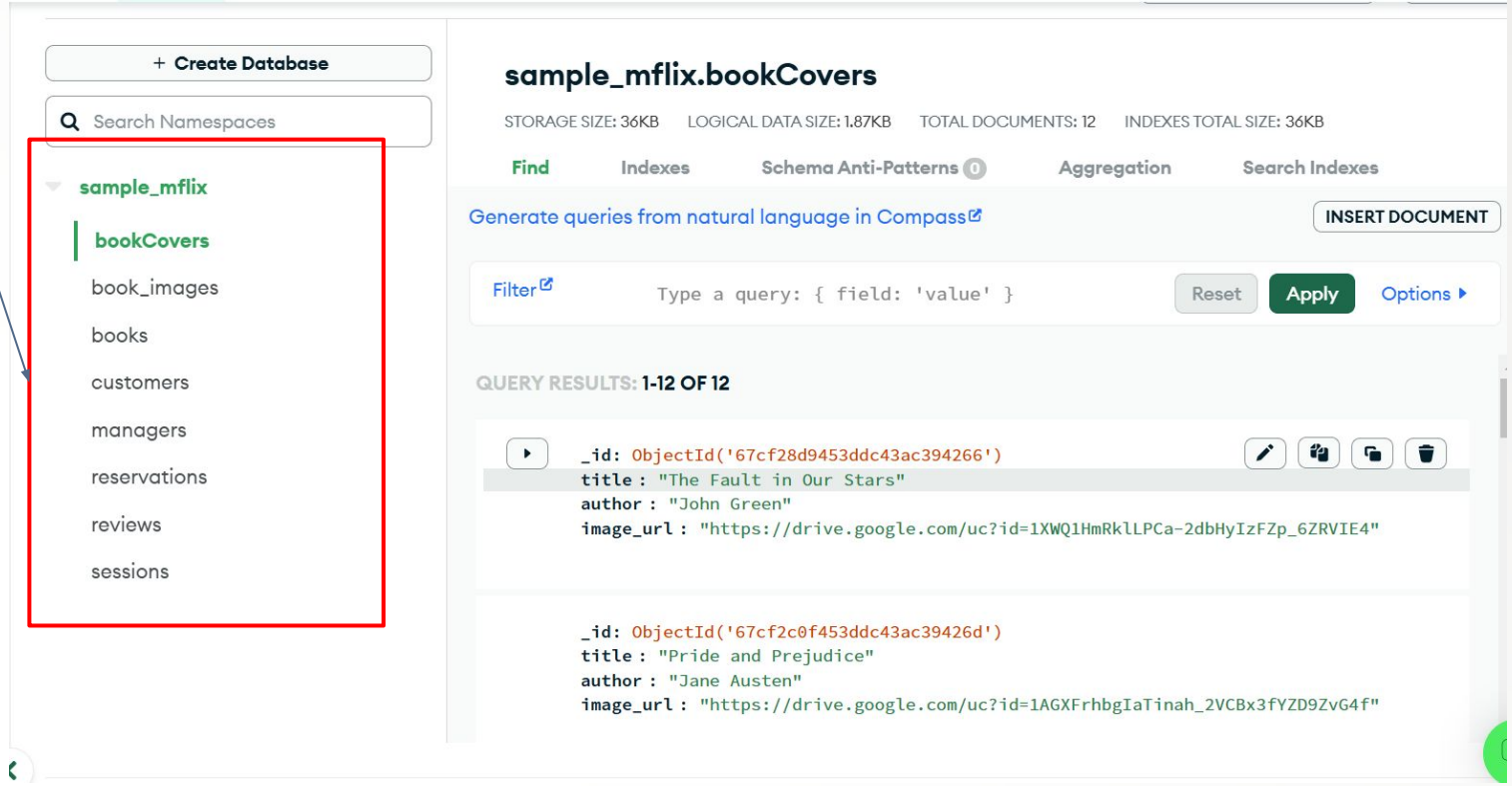
NoSQL

- Use flexible, schema-less designs to store data in various formats like key-value pairs, documents, graphs, or wide-columns
- Schema-less or dynamic schema, allowing easier changes
- Focuses on simplicity and performance

Database : MongoDB

Example: Dummy Database

Collection Tables



How to Connect MongoDB to VSCode

Link to our database and collections

Deployment

Docker, Kubernetes, Ingress, Google Cloud

Deployment

Docker, Kubernetes, Ingress and Google Cloud

Ingress.yaml file

```yaml
- path: /auth
  pathType: Prefix
  backend:
    service:
      name: customer-auth
      port:
        number: 8001
- path: /catalog
  pathType: Prefix
  backend:
    service:
      name: catalog
      port:
        number: 8002
```

Prefix for each microservice used for routing client-side.

Port for each microservice used for routing server-side

```
C:\Windows\SysWOW64>kubectl get pods
NAME                                READY   STATUS
catalog-6c8c7cbfbf-52xzp            1/1     Running
customer-auth-558cd66978-zjw8f      1/1     Running
mylibrary-6f795bcf6c-nsgmb          1/1     Running
reservations-9787b9f8-8tw9m         1/1     Running
search-d667ccd78-v2h4g              1/1     Running
```

Microservices run inside "pods" on a Kubernetes cluster

```
C:\Windows\SysWOW64>kubectl get svc
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
catalog         ClusterIP   34.118.236.156   <none>        8002/TCP
customer-auth   ClusterIP   34.118.233.147   <none>        8001/TCP
kubernetes      ClusterIP   34.118.224.1     <none>        443/TCP
reservations    ClusterIP   34.118.237.28    <none>        8004/TCP
search          ClusterIP   34.118.235.113   <none>        8003/TCP
```

Each microservice has an internal IP address and unique port

Customer Authentication

Authentication and authorization of customers and managers.

Front End View

- Use of Bootstrap form control for the login and create account forms.

```html
<form id="loginForm" action="/auth/login" method="POST">
  <div class="row">
    <div class="col-2"></div>
    <div class="col-8">
      <div class="form-group p-3">
        <label for="email">Email</label>
        <input type="email" name="email" class="form-control rounded px-10 mb-2" id="email" placeholder="Enter your email" required/>
        <label for="password">Password</label>
        <input type="password" name="pword" class="form-control rounded" id="password" placeholder="Enter your password" required/>
        <div class="d-flex justify-content-
          <a href="/auth/forgot-password" c
                                                Password?</a>
        </div>
        <div id="errorMessage" class="fs-6 text-danger p-0 m-0"></div>
        <button type="submit" class="btn btn-custom-size w-100 mt-3 mb-2">Login</button>
      </div>
      <div class="col-2"></div>
    </div>
  </div>
</form>
```

Route to the login method in routers.py.

Form group for login page.

Select type for input fields.

Using bootstrap button with custom CSS.

```css
/* Custom button size */
.btn-custom-size {
    padding: 6px 12px;
    border-radius: 15px;
    font-size: 16px;
    cursor: pointer;
    background-color: #162d3a;
    color: white;
    transition: background-color 0.3s;
}
```

Front End View

- Use Javascript for error handling.

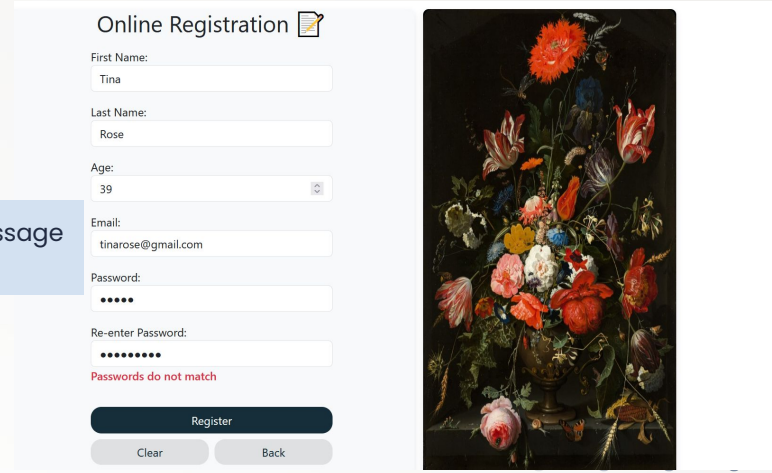
```javascript
// Get the values of the new password and re-entered password fields
const new_password = document.getElementById('password').value;
const re_entered_password = document.getElementById('re-password').value;

// Get the error message container
const passErrorMessage = document.getElementById('passwordErrorMessage');
passErrorMessage.style.display = 'none';
passErrorMessage.textContent = '';

// Check if fields are empty
if (new_password === '' || re_entered_password === '') {
    passErrorMessage.textContent = 'Enter required fields';
    passErrorMessage.style.display = 'block';
    return;
}
else if (new_password !== re_entered_password) {
    passErrorMessage.textContent = 'Passwords do not match';
    passErrorMessage.style.display = 'block';
    return;
}
```

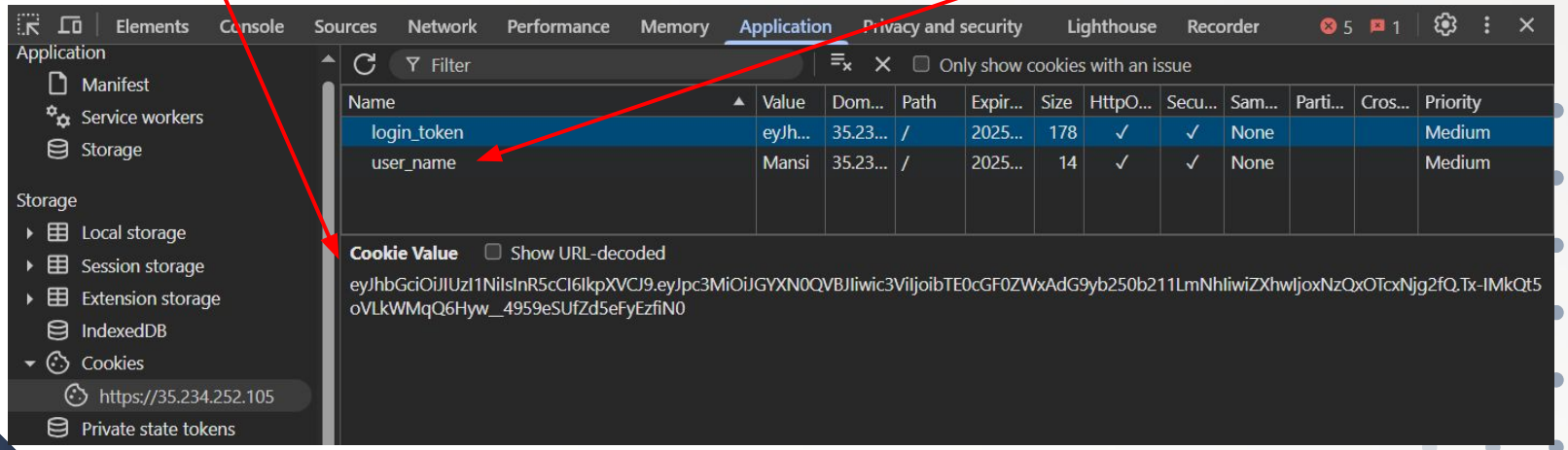
Get the error message element.

Dynamically display the error message.



Backend Logic

- The login_token stores the user's email address in the payload that is encoded as a JWT token. This token is set as a cookie on the user's browser
- The user_name cookies stores the user's first name for user experience



Database Queries + Collections

```
_id: ObjectId('6787541dea7503b89fa5cb19')
email : "johndoe@example.com"
password : "john"
▸ address : Object
firstName : "John"
lastName : "Doe"
age : 12
▸ wishlist : Object
```

```
_id: ObjectId('67874f926021fa95d522ea01')
managerID : "MGR00001"
firstName : "Alice"
lastName : "Johnson"
email : "alice.johnson@example.com"
phoneNumber : "6475550101"
▸ address : Object
passwordHash : "MGR00001"
```

```python
@app.get("/customers/{email}", response_model=Customer)
def get_user(email: str):
    customer = db["customers"].find_one({"email": email})
    if not customer:
        return None
    customer["_id"] = str(customer["_id"])
    if isinstance(customer["age"], dict) and "$numberInt" in customer["age"]:
        customer["age"] = int(customer["age"]["$numberInt"])

    return Customer(**customer)
```

Queries the customers for a customer with the provided email.

```python
@app.get("/customers/{firstName}", response_model=Customer)
async def get_user_by_fname(firstName: str):
    customer = await get_db["customers"].find_one({"firstName": firstName})
    if not customer:
        raise HTTPException(status_code=404, detail="User not found")
    return Customer(**customer)
```

Queries the database for a customer with the provided firstName.

```python
@app.post("/customers/", response_model=dict)
def create_user(customer: Customer):
    customer_dict = customer.dict()
    result = db["customers"].insert_one(customer_dict)
    return {"id": str(result.inserted_id)}
```

This endpoint creates a new customer record in the MongoDB database

```python
@app.put("/customers/{email}/password", response_model=dict)
def change_password(email: str, new_password: str):
    customer = get_user(email)
    if not customer:
        raise HTTPException(status_code=404, detail="User not found")
    result = db["customers"].update_one(
        {"email": email},
        {"$set": {"password": new_password}}
    )
    if result.modified_count == 0:
        raise HTTPException(status_code=400, detail="Password update failed")
    return {"message": "Password updated successfully"}
```

Updates the password of a customer based on the provided email

Catalogue Management

Manages the catalogue collection of books and materials

Front End View

```html
<div class="col-md-6">
  <label for="status" class="form-label">Status:</label>
  <select id="status" name="status" class="form-select" required>
    <option selected disabled>Choose...</option>
    <option>Available</option>
    <option>Not Available</option>
  </select>
</div>
<div class="col-12">
    <div class="mb-3">
      <label for="imageUpload" class="form-label">Upload Book Cover:</label>
      <input type="file" class="form-control" id="imageUpload" name="imageUpload" accept="image/*" onchange="previewImage(event)">
    </div>
    <div class="mb-3">
      <img id="imagePreview" src="#" alt="Image Preview" style="display: none; width: 100%; max-width: 300px;"/>
    </div>
</div>
```

Use Bootstrap to create different input field types to create visually appealing forms.

Use Javascript to send the form information in JSON format to API endpoint.

```javascript
const formData = new FormData(document.getElementById("addBookForm"));
const data = Object.fromEntries(formData.entries());

fetch('/catalog/add-item', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
})
```

Front End View

Library Management System

My Dashboard | Catalogue View | Account | Log Out

Add Book to Inventory

Title:

This field is required

ISBN:

This field is required

Author:

This field is required

Genre:

This field is required

Rating:

This field is required

Kid Friendly:

Choose...

This field is required

Description:

Different input fields.

Image upload field for book covers.

This field is required

Book Format:

This field is required

Page Number:

This field is required

Number of Copies:

This field is required

Publisher:

This field is required

Status:

Choose...

This field is required

Upload Book Cover:

Browse... No file selected.

Submit | Clear | Cancel

Backend Logic

```python
def handle_add_book(title: str, isbn: str, author: str, genre: str, rating: float,
                    kidFriendly: bool, description: str, format: str, pageNumber: int,
                    numCopies: int, publisher: str, status: str):
    book = get_book(isbn)
    if book is None:
        return create_book(Book(title=title, isbn=isbn, author=author, genre=genre, rating=rati
                kidFriendly=kidFriendly, description=description, format=format, pageNumber=pag
                numCopies=numCopies, publisher=publisher, status=status))
    return "Error"

def handle_modify_book(title: str, isbn: str, author: str, genre: str, numCopies: int, descript
    book = get_book(isbn)
    if book is not None:
        updates = {"title": title, "isbn": isbn, "author": author, "genre": genre, "numCopies":
                   "description": description, "kidFriendly": kidFriendly, "format": format,
                   "publisher": publisher, "status": status}
        update_occurred = False
        for field, new in updates.items():
            if getattr(book, field) != new:
                update_method = globals().get(f"update_{field}")
                if update_method:
                    response = update_method(book.isbn, new)
                    print(f"\n\n{response}\n\n")
                    if response:
                        update_occurred = True
        return update_occurred
```

- The Controller layer includes functions for adding, deleting and modifying book data

A new database query called "update_method" is referenced for every field of the Book table

Checks if the field has changed, and updates the book information in the database

Database Queries + Collections

```python
# Get book information
@app.get("/books/{isbn}", response_model=Book)
def get_book(isbn: str):
    book = db["books"].find_one({"isbn": isbn})
    if not book:
        return None
    book["_id"] = str(book["_id"])

    if isinstance(book["pageNumber"], dict) and "$numberInt" in book["pageNumber"]:
        book["pageNumber"] = int(book["pageNumber"]["$numberInt"])
    if isinstance(book["rating"], dict) and "$numberInt" in book["rating"]:
        book["rating"] = float(book["rating"]["$numberInt"])
    if isinstance(book['kidFriendly'], dict) and "$numberInt" i
        book['kidFriendly'] = bool(book['kidFriendly']["$number

    return Book(**book)
```

To retrieve a book document from the MongoDB database using its ISBN and return the book details as a response.

```
_id: ObjectId('67881101ea7503b89fa5e3bc')
title : "Harry Potter and the Sorcerer's Stone"
author : "J.K. Rowling"
genre : "Fantasy"
rating : 5
kidFriendly : true
description : "A young wizard discovers his magical heritage and attends the
              Hogwarts…"
format : "Audio"
pageNumber : 0
```

```python
# Update book fields
@app.put("/catalog/update-isbn/{isbn}", response_model=dict)
def update_isbn(isbn: str, new_isbn: str):
    result = db["books"].update_one(
        {"isbn": isbn},
        {"$set": {"isbn": new_isbn}}
    )
    return result.modified_count > 0

@app.put("/catalog/update-title/{isbn}", response_model=dict)
def update_title(isbn: str, new_title: str):
    result = db["books"].update_one(
        {"isbn": isbn},
        {"$set": {"title": new_title}}
    )
    return result.modified_count > 0

@app.put("/catalog/update-description/{isbn}", response_model=dict)
def update_description(isbn: str, new_description: str):
    result = db["books"].update_one(
        {"isbn": isbn},
        {"$set": {"description": new_description}}
    )
    return result.modified_count > 0
```

Updates the isbn field of a book in the database

Updates the title field of a book in the database.

Updates the description field of a book in the database.

Reservations Management

Holds a book for a user for two weeks.

Front End View

```html
<table class="table table-bordered">
<thead>
    <tr>
    <th></th>
    <th>#</th>
    <th>Title</th>
    <th>ISBN</th>
    <th>Book ID</th>
    <th>User Information</th>
    <th>Hold Date</th>
    <th>Due Date</th>
    <th>Status</th>
    </tr>
</thead>
<tbody id="book-table">
    <!-- Rows will be added dynamically here -->
</tbody>
</table>
```

Dynamically fill table with rows through javascript.

```javascript
const row = document.createElement('tr');
row.innerHTML = `
    <td><input type="checkbox" class="selectRow"></td>
    <th scope="row">${i}</th>
    <td>${title}</td>
    <td>${book.isbn}</td>
    <td>${book.book_id}</td>
    <td>${book.user_email}</td>
    <td>${formatDate(book.reservation_date)}</td>
    <td>${formatDate(book.expiration_date)}</td>
    <td>${book.status}</td>
`;
table.appendChild(row);
i++;
```

```html
<li class="list-group-item">
    <div class="form-check">
        <input class="form-check-input" type="checkbox" value="user" id="filter-user">
        <label class="form-check-label" for="filter-user"> User </label>
    </div>
</li>
```

Create the filter option in HTML.

```js
else if (filterOpt === 'user'){
    const filteredBooks = [];
    books.forEach(book => {
        if (!filteredBooks.includes(book.user) && book.user.toLowerCase().includes(query)){
            filteredBooks.push(book.user)
        }
    });
    console.log(filteredBooks);
    displayUser(filteredBooks);
}
```

Checked reservation list for entered users name and add to filtered list.

Call function to display the filtered results as buttons.

```js
function displayUser(usersToDisplay) {
    const searchList = document.getElementById('bookList');
    searchList.innerHTML = ''; // Clear the existing list
    if (usersToDisplay.length === 0) {
        searchList.innerHTML = '<li class="list-group-item">No entries found.</li>';
    } else {
        usersToDisplay.forEach(user => {
            const listItem = document.createElement('li');
            listItem.classList.add('list-group-item');

            // Create a button for each book
            const itemButton = document.createElement('button');
            itemButton.classList.add('btn', 'btn-custom-search', 'w-100');
            itemButton.innerHTML = `<p align="left">${user}</p>`;

            itemButton.addEventListener('click', () => {
                filterTable(user, 'user');
                clearSearchResults();
            });
            listItem.appendChild(itemButton);
            searchList.appendChild(listItem);
        });
    }
}
```

Filter table rows by the search results.

```js
tableRows.forEach(row => {
    const cells = row.querySelectorAll('td');
    console.log(cells);
    if (filter === 'user'){
        shouldDisplay = cells[4].textContent.toLowerCase().includes(query);
        console.log(cells[4].textContent.toLowerCase());
        row.style.display = shouldDisplay ? '' : 'none';
    }
```

Front End View

Library Management System

My Dashboard | Catalogue View | Account | Log Out

Hold Requests List

Filter | Clear | john

- [ ] Book ID
- [ ] ISBN
- [x] User
- [ ] Hold Date
- [ ] Due Date

	#	Title	ISBN	Book ID		Hold Date	Due Date	Status
[ ]	1	The Hobbit	33333HZNDU	TH-1937	.com	Feb 6, 2025	Feb 13, 2025	no longer valid
[ ]	2	To Kill a Mockingbird	22222YNFAE	TKM-00	.com	Feb 7, 2025	Feb 12, 2025	no longer valid
[ ]	3	Harry Potter and the Sorcerer's Stone	11111LBXRN	HP-001	.com	Mar 8, 2025	Mar 13, 2025	complete
[ ]	4	To Kill a Mockingbird	22222YNFAE	TKM-003	johndoe@example.com	Mar 10, 2025	Mar 20, 2025	complete
[ ]	5	Harry Potter and the Sorcerer's Stone	11111LBXRN	HP-002	johndoe@example.com	Mar 10, 2025	Mar 20, 2025	pending
[ ]	6	Harry Potter and the Sorcerer's Stone	11111LBXRN	HP-003	johndoe@example.com	Mar 12, 2025	Mar 17, 2025	pending

Extend | Cancel | Back

Backend Logic

```javascript
let books = [];

async function fetchHolds() {
  try {
    const response = await fetch("/reservations/list-holds/");
    if (response.ok) {
      const booksData = await response.json();
      books = [...booksData];
      console.log(books);

      // Update books status
      for (const book of books) {
        const isbn = book.isbn; // Assuming isbn is a field in
        const book_id = book.book_id; // Assuming book_id is a
```

Uses existing API endpoints to get a list of books from the database and gets a JSON response

A new row is created for each reservation that is retrieved from the database

```javascript
async function createTable(filteredBooks = books) {
    let i = 1;
    const table = document.getElementById('book-table');
    table.innerHTML = '';

    for (const book of filteredBooks) {
        let title = book.title;
        if (!title) {
            title = await fetchBooksISBN(book.isbn) || "Unknown Title";
        }

        const row = document.createElement('tr');
        row.innerHTML = `
            <td><input type="checkbox" class="selectRow"></td>
            <th scope="row">${i}</th>
            <td>${title}</td>
            <td>${book.isbn}</td>
            <td>${book.book_id}</td>
            <td>${book.user_email}</td>
            <td>${formatDate(book.reservation_date)}</td>
            <td>${formatDate(book.expiration_date)}</td>
            <td>${book.status}</td>
        `;
        table.appendChild(row);
        i++;
```

Database Queries + Collections

```python
@app.get("/holds/", response_model=List[Reservations])
def list_holds():
    db = get_db()
    holds = list(db["reservations"].find())
    for hold in holds:
        hold["_id"] = str(hold["_id"])
        hold["reservation_id"] = str(hold["reservation_id"]) if isinstance(hold["reservation_id"], ObjectId) else hold["reservation_id"]
        hold["user_id"] = str(hold["user_id"]) if isinstance(hold["user_id"], ObjectId) else hold["user_id"]
    return [Reservations(**hold) for hold in holds]
```

```python
@app.get("/extendHold/")
def extend_hold(isbn: str, book_id: str):
    reservation = db["reservations"].find_one({"isbn": isbn, "book_id": book_id})
    if not reservation:
        return JSONResponse(status_code=404, content={"message": "Hold not found"})

    # Extend the due date by 5 days
    new_due_date = extend_due_date(reservation["expiration_date"])
    if new_due_date is None:
        return JSONResponse(status_code=400, content={"message": "Invalid due date format"})

    update_result = db["reservations"].update_one({"_id": reservation["_id"]}, {"$set": {"expiration_date": new_due_date}})
    if update_result.modified_count == 0:
        return JSONResponse(status_code=500, content={"message": "Failed to update the due date"})

    return JSONResponse(status_code=200, content={"message": "Hold successfully extended"})
```

Retrieve and return a list of all reservation (hold) records from the reservations collection in the database.

To extend the due date of a reservation for a specific book, identified by its isbn and book_id.

Future Plans

Complete pending microservices, finalize deployment and integration testing!

Thank you!

Thank you for your time and attention!