Western Norway
University of
Applied Sciences

DAT152 – Advanced Web Applications

Web Services II

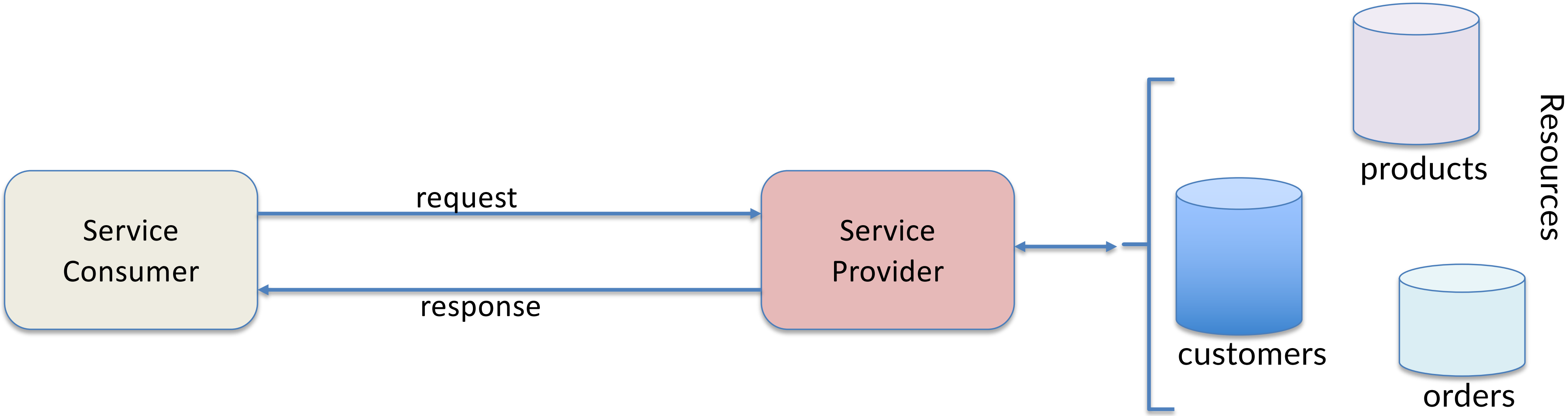
Agenda

- Resource Modelling & REST API Design
- Building RESTful APIs for elibrary service

Resource: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

Resource & REST API Modeling

- When building RESTful APIs, we need to organize the API design around **resources**
- What are the services to render and what are the resources the services will provide?



High-Level Web API Design Goals

- A well-designed web API should aim to support:
  - Platform independence
    - Any client should be able to call the API, regardless of the internal implementation (standard protocol, format of data exchange)
  - Service evolution
    - The web API should be able to evolve and add functionality independently from client applications.
    - Existing application should continue to function without modification
    - All functionality should be discoverable

Resource Modeling & REST API design

- Let’s start with the rules and later apply them to constructing a RESTful API.
- Organize the API design around resources
- A resource must have a name
  - Plural noun for collection (e.g. /customers)
  - Singular noun for document (e.g. /customers/customer
  - Verb for controller (e.g. /alerts/245743/resend
- A resource must have a unique URI path
- A path takes the form:
  - /resource/identifier
  - /resource/identifier/resource
  - collection/document/collection

Resource Modeling & REST API design

Recommendations

- A resource does not have to be based on a single physical data item
  - e.g. an **order** resource might be implemented internally as several tables but presented to the client as a **single entity**
- Avoid creating APIs that simply mirror the internal structure of a database

Resource Modeling & REST API design

- Organize URIs for collections and documents into a hierarchy
  - e.g. ***/customers*** (Path to the customers collection)
  - ***/customers/5*** (Path to the customer with id = 5)
- Consider the relationships between different types of resources and how you might expose these associations
  - e.g. ***URI: /customers/5/orders***
  - Or ***URI: /orders/99***

Resource Modeling & REST API design

- Avoid URI design deeper than
  - *resource/identifier/resource*
  - *collection/{id}/collection*

This is a recommendation not a hard rule!

collection/{id}/collection/{id}

- It can be tempting to provide URI that allows client to navigate through levels of relationships
  - e.g. /customers/1/orders/99/products
- Problems:
  - Difficult to maintain and inflexible if relationships between resources change in the future
- Can be simplified into 2 URIs:
  - /customers/1/orders
  - /orders/99/products

Resource Modeling & REST API design

- Web requests impose a load on the web server,
  - The more requests, the bigger the load.
- Therefore,
- Avoid "chatty" web APIs that expose a large number of small resources.
- May combine related information into bigger resources that can be retrieved with a single request
  - Concern: Latency and bandwidth costs?

Resource Modeling & REST API design

Example of a “chatty” web APIs

- Implementing a single logical operation as a series of HTTP requests

```
public class AppUser {

    private String username;
    private String firstname;
    private String lastname;
    private String gender;
    private Date dateOfBirth;
    ...
}
```

Resource URI Path	**HTTP Method**
/users/{id}/username	GET
/users/{id}/gender	GET
/users/{id}/dateofbirth	GET

Solution: /users/{id}

Resource Modeling & REST API Design

Resource	**POST**	**GET**	**PUT**	**DELETE**
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

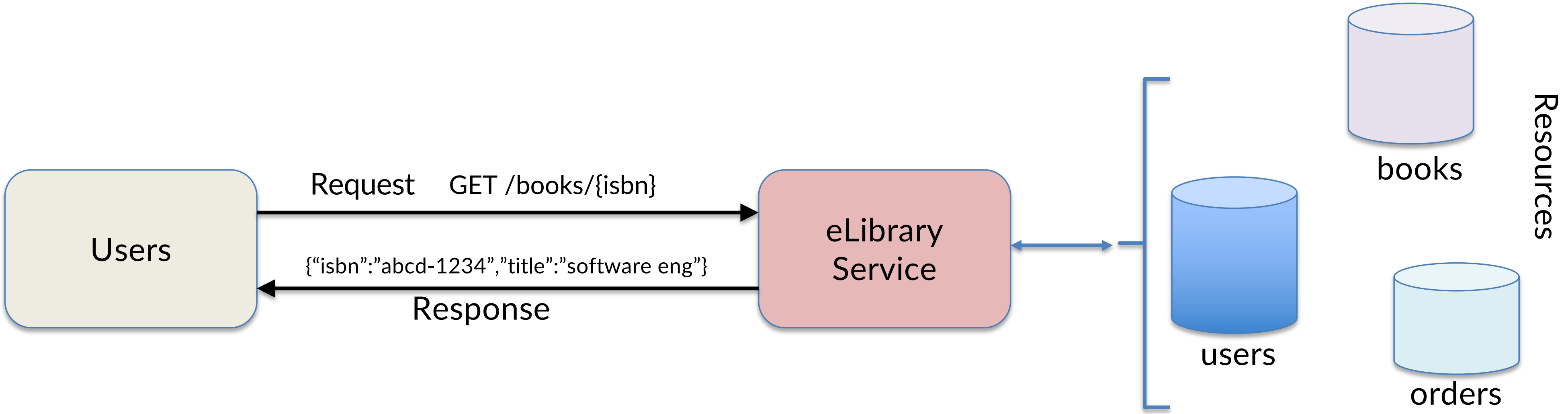
REST APIs for eLibrary Service

The REST API should provide the following services:

- Produce a list of all the books with their authors.
- Provide details of each book with its author.
- Provide support for creating, updating and deleting a book.
- Produce a list of all the authors with the books they published.
- Provide details of each author with their published books.
- Provide the support for creating and updating an author.
- Produce a list of all the library users.
- Provide details of each library user.
- Provide support for creating, updating and deleting library users.
- Provide support for users to order and return books.
- Produce a list of all books ordered (borrowed) by a user.

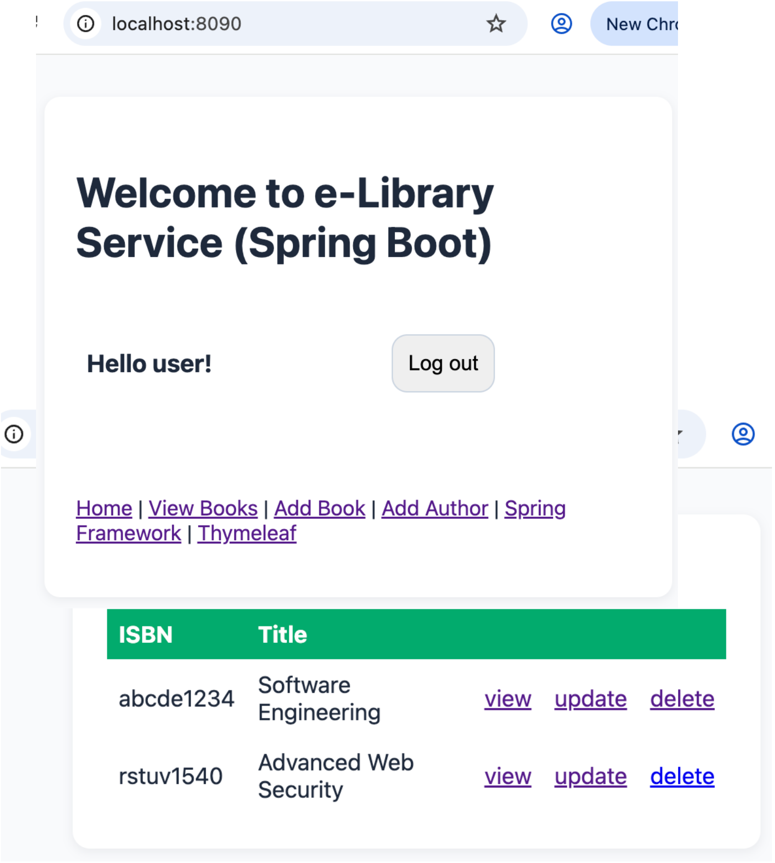
REST APIs for eLibrary Service

- Entities


Request GET /books/{isbn}
Users
eLibrary Service
{"isbn":"abcd-1234","title":"software eng"}
Response
books
users
orders
Resources

REST APIs for eLibrary Service

- Services (some)
  - createBook
  - updateBook
  - getBooks
  - getBook
  - deleteBook
  - createAuthor
  - updateAuthor
  - getAuthors
  - createUser
  - updateUser
  - deleteUser
  - getUsers
  - getUser
  - borrowBook
  - returnBook



Welcome to e-Library Service (Spring Boot)

Hello user!

Home | View Books | Add Book | Add Author | Spring Framework | Thymeleaf

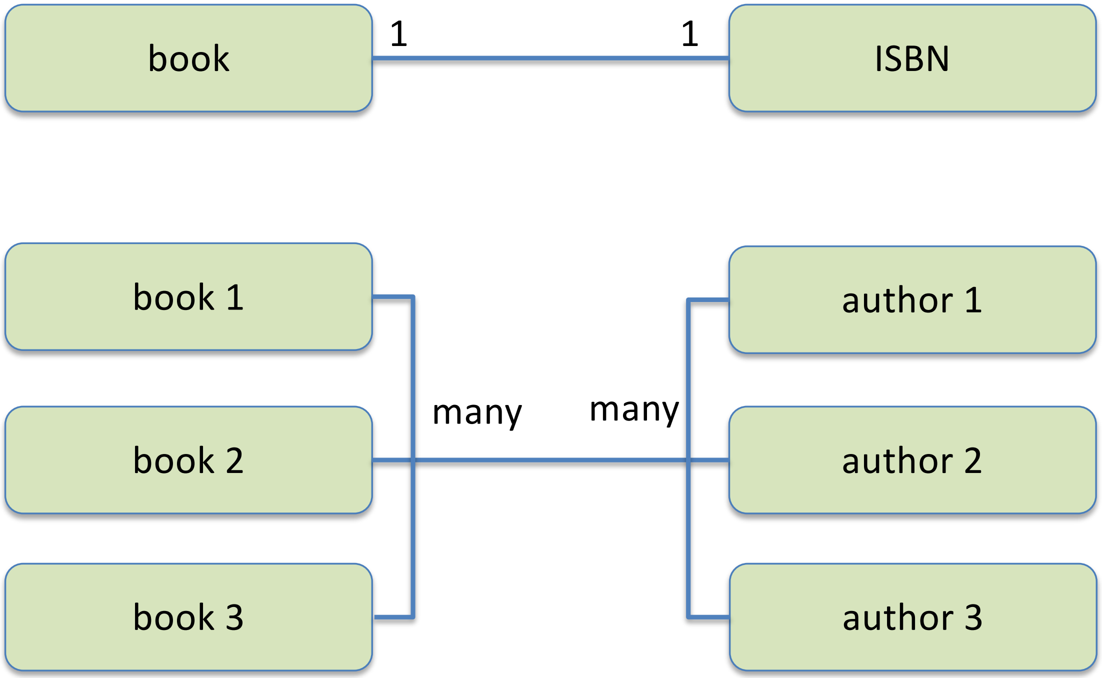
ISBN	Title			
abcde1234	Software Engineering	view	update	delete
rstuv1540	Advanced Web Security	view	update	delete

View

```
{
        "id": 1,
        "isbn": "abcde1234",
        "title": "Software Engineering",
        "authors": [
                {
                "authorId": 1,
                "firstname": "Shari",
                "lastname": "Pfleeger"
                }
        ]
}
```

REST format (json)

Book-Author Model



book 1 — 1 ISBN

book 1

book 2

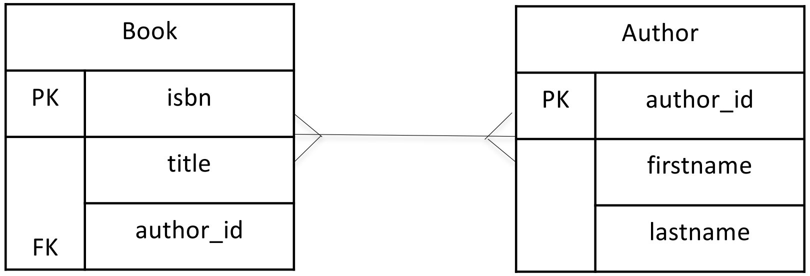
book 3

many — many

author 1

author 2

author 3



Book

PK isbn

title

FK author_id

Author

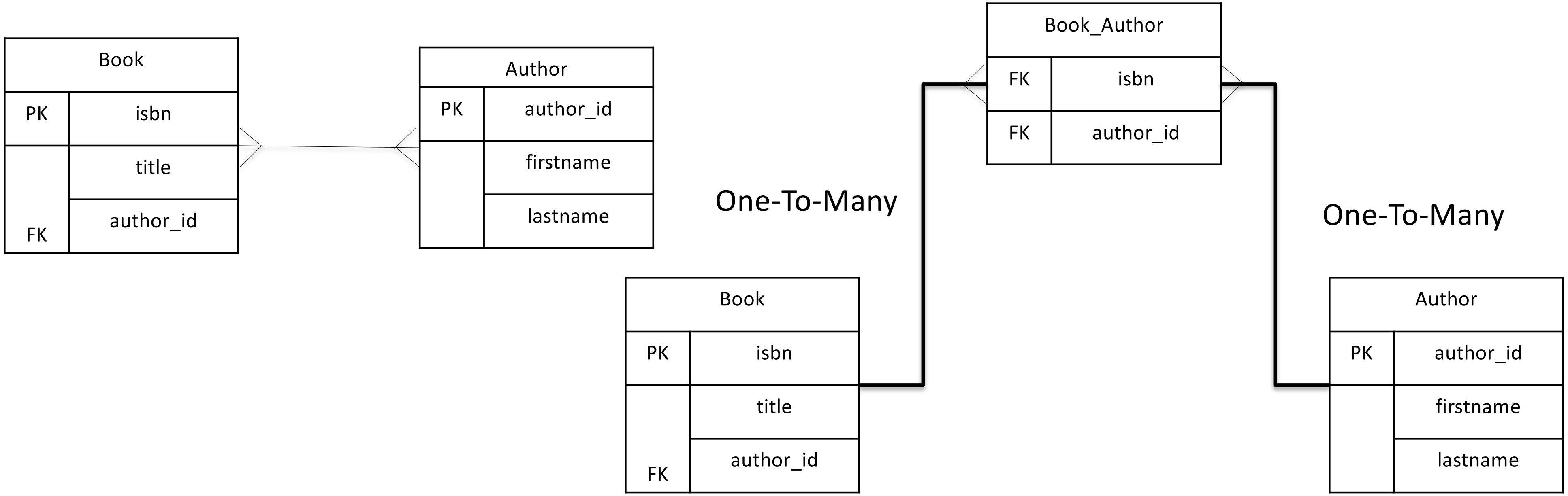
PK author_id

firstname

lastname

Book-Author Model

- Many-To-Many relationship



Book	
PK	isbn
	title
FK	author_id

Author	
PK	author_id
	firstname
	lastname

Book_Author	
FK	isbn
FK	author_id

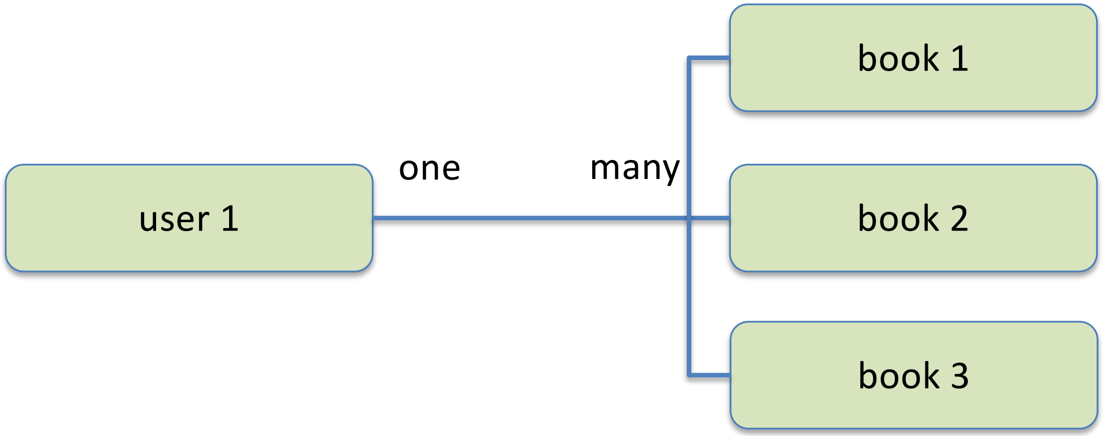
One-To-Many

One-To-Many

Book	
PK	isbn
	title
FK	author_id

Author	
PK	author_id
	firstname
	lastname

User-Book Model



user 1

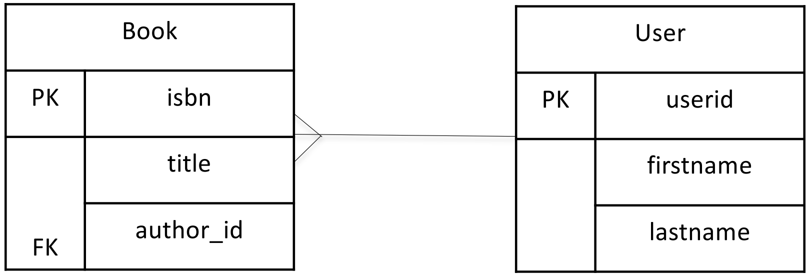
one

many

book 1

book 2

book 3

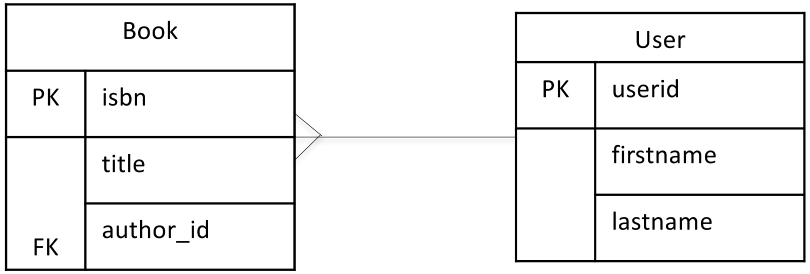


Book	
PK	isbn
	title
FK	author_id

User	
PK	userid
	firstname
	lastname

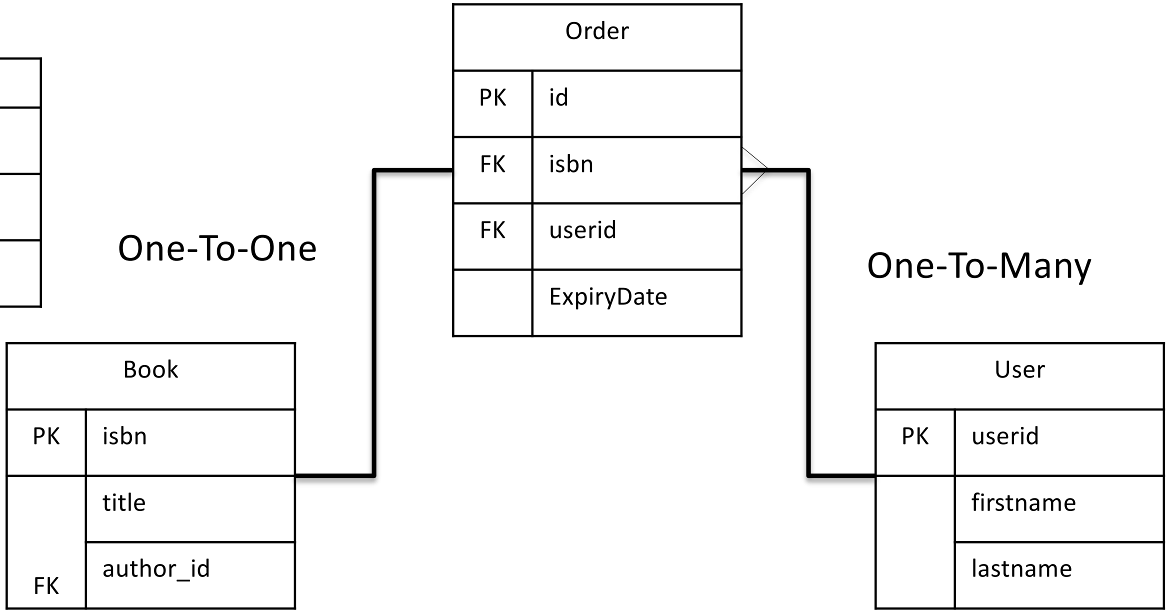
User-Book Model

- One-To-Many relationship



Book	
PK	isbn
	title
FK	author_id

User	
PK	userid
	firstname
	lastname



Order	
PK	id
FK	isbn
FK	userid
	ExpiryDate

Book	
PK	isbn
	title
FK	author_id

User	
PK	userid
	firstname
	lastname

eLibrary REST API Endpoints

Service	**API Method**	**Resource Endpoint (URI Path)**	**HTTP Method**
List all books	getBooks	/books	GET
List of borrowed books by user	borrowedBooksByUserId	/users/{userid}/orders /users/{userid}/books	GET
borrow book	borrowBooks	~~/books~~ /orders	POST
All borrowed books	getAllBorrowedBooks	/orders	GET
Return/cancel book by a user	returnBook	/orders/{id}	DELETE

eLibrary REST API Endpoints

Service	**API Method**	**Resource Endpoint (URI Path)**	**HTTP Method**
List of authors	getAuthors	/authors	GET

eLibrary REST API Endpoints

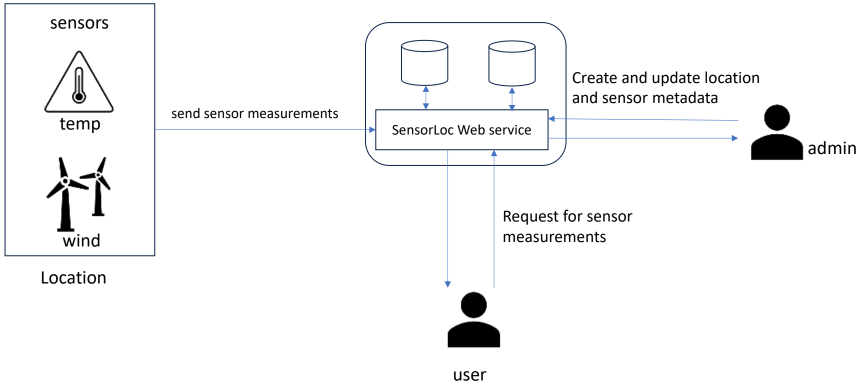
Service	**API Method**	**Resource Endpoint (URI Path)**	**HTTP Method**
Create a borrow book order	borrowBook	/orders	POST

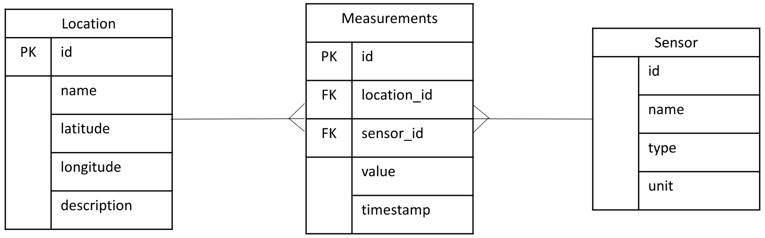
eLibrary REST API Endpoints

Service	**API Method**	**Resource Endpoint (URI Path)**	**HTTP Method**
List all borrow orders filtered by expiry date	getBorrowOrders	/orders?query=expirydate	GET

Filtering:
Query by expirydate

Exercises (Exam June 2024)


sensors
temp
wind
Location
send sensor measurements
SensorLoc Web service
Create and update location and sensor metadata
admin
Request for sensor measurements
user



The above figure shows a simple sensor location (SensorLoc) web service and the entity model diagram. A location (e.g., Kronstad) can contain many sensors of different types (e.g., temperature and wind sensors). An admin can create, update or delete a location and sensor metadata via REST API endpoints. Each sensor can send its measurement (data) to the SensorLoc web service via REST API endpoints. An external user can request for different sensor measurements for different locations via REST API endpoints.

Build a REST API endpoints for this service. The REST API should provide the following services:

- ○ Provide support to create, update, and delete location metadata
- ○ Provide details of each location
- ○ Produce a list of all locations
- ○ Provide support to create, update and delete sensor metadata
- ○ Provide details of each sensor
- ○ Produce a list of all sensors
- ○ Produce the list of all measurements for all sensors in a location.
- ○ Produce measurements for a sensor in a location.

Use the table below to structure your answers (example in the first row).

Note that you can have up to four levels (e.g., collection/identifier/collection/identifier)

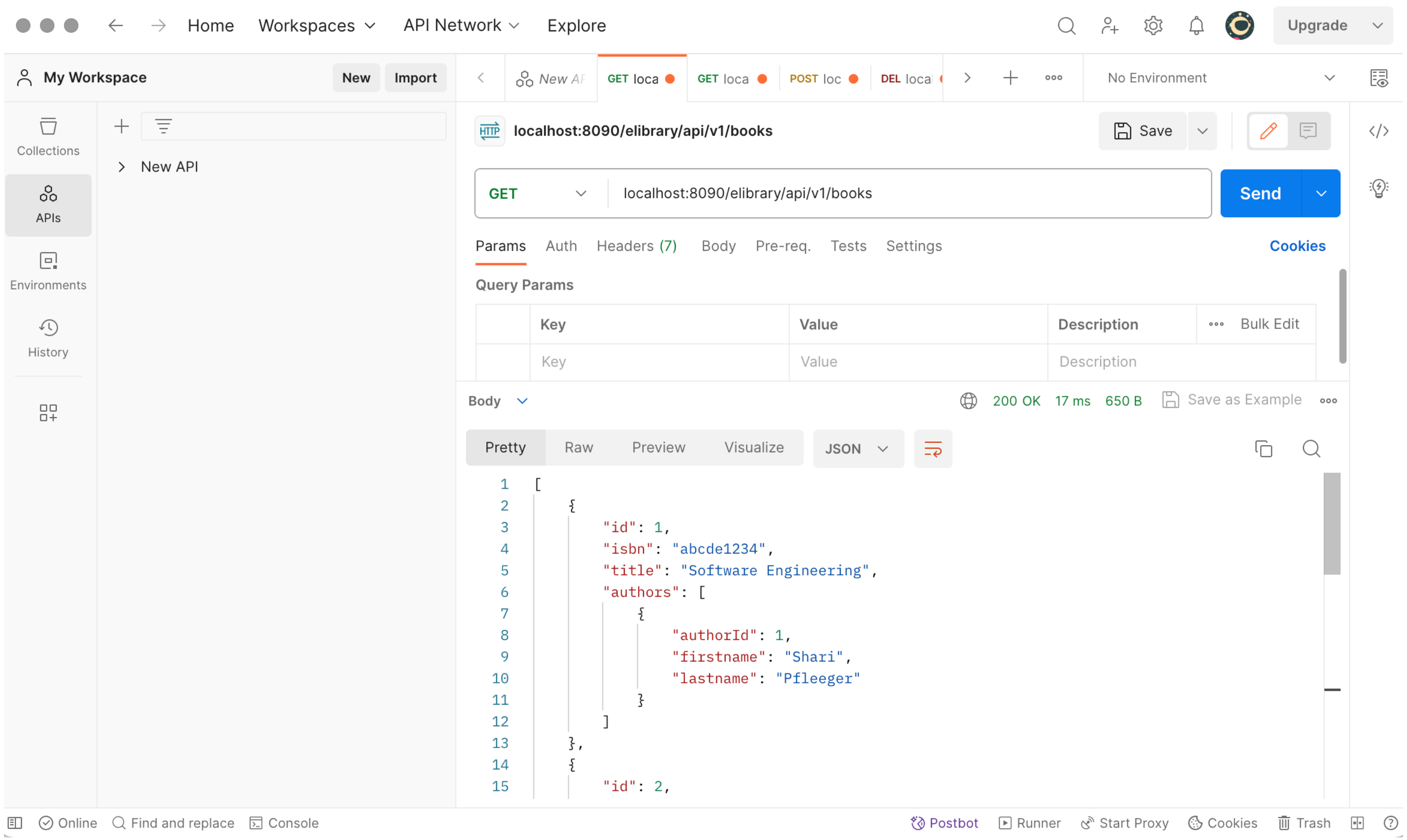
Service	**API Method**	**Endpoint (URI)**	**HTTP Method**
List all locations	getLocations	/locations	GET

Truly RESTful API

- Discoverability & Self-descriptive
  - It should be possible to navigate the entire set of resources without prior knowledge of the URI scheme
- HTTP GET request should
  - return information necessary to find the resources related to the requested object through hyperlinks
  - provide operations possible on each of these resources
- This principle is known as HATEOAS
  - Hypertext(media) as the Engine of Application State

Next lecture, we'll look at the Richardson Maturity Model

Postman for testing REST APIs



localhost:8090/elibrary/api/v1/books

GET localhost:8090/elibrary/api/v1/books

Key	Value	Description
Key	Value	Description

Body 200 OK 17 ms 650 B

```
[
    {
        "id": 1,
        "isbn": "abcde1234",
        "title": "Software Engineering",
        "authors": [
            {
                "authorId": 1,
                "firstname": "Shari",
                "lastname": "Pfleeger"
            }
        ]
    },
    {
        "id": 2,
```

Lab – REST Services

- Build on the previous library service
  - Create a RESTful web service for the library model
  - Spring Framework + REST
- You have a startcode posted on Canvas

- Task Descriptions also on Canvas