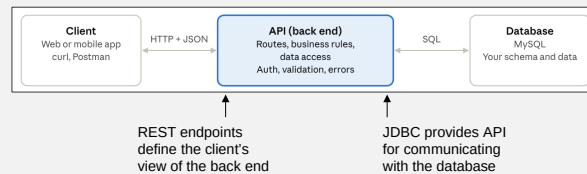


Back End



- the back end is a *service* – it starts up and continues running, waiting for requests
- Javalin is a small web framework for Java
 - contains an embedded web server
- application main program structure
 - create a Javalin instance
 - register handlers for each endpoint
 - Javalin turns each incoming HTTP request into a Context object passed to the handler
 - start the service – listen on a port

40

Back End With Javalin

```

Javalin app = Javalin.create();           // the server object
app.post("/books", ctx -> {               // register handler for GET /books
    Map<String, Object> body = ctx.bodyAsClass(Map.class);
    String title = (String) body.get("title");
    if (title == null || title.isBlank()) {
        ctx.status(400).result("title is required");
        return;
    }
    try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
        PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO BOOKS (isbn, title, author) VALUES (?, ?, ?)",
            Statement.RETURN_GENERATED_KEYS)) {
        ps.setString(1, (String) body.get("isbn"));
        ps.setString(2, title);
        ps.setString(3, (String) body.get("author"));
        ps.executeUpdate();
        // ... read the key, build a response map, ctx.status(201).json(...)
    }
});

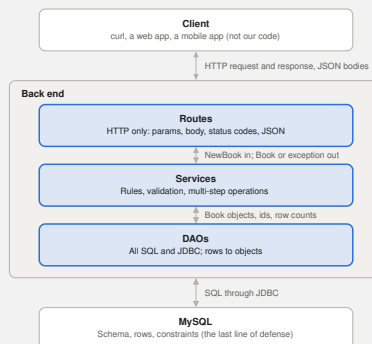
...
app.start(7070);                          // start listening
    
```

CPSC 343: Database Theory and Practice • Fall 2026

41

Layered Back End

- routes – HTTP in and out
 - process request – parse the path, query string and body
 - call a service
 - produce response – set the status, write JSON
- services – the application's rules
 - validate, enforce policies, orchestrate multi-step operations
- DAOs
 - all SQL and JDBC
- each layer only calls the layer below it



CPSC 343: Database Theory and Practice • Fall 2026

42

Domain Objects

- domain objects* are the things the application works with
 - entities from the ER diagram
 - relationships from the ER diagram that have data (attributes) of their own beyond the entities connected
 - bookkeeping data the application needs to work with
 - e.g. sessions
- represented by a plain class whose fields mirror the table columns
 - no SQL, no JDBC, no HTTP – just data

```

public record Book ( Integer bookId, String isbn, String title, String author,
    LocalDate publishedOn, BigDecimal replacementCost,
    boolean available ) { }

public record Member ( Integer memberId, String username, String passwordHash,
    String role ) { }

public record Loan ( Integer loanId, int bookId, int memberId,
    LocalDate checkedOutOn, LocalDate dueOn,
    LocalDate returnedOn ) { }
    
```

43

Java record

- record is a shortcut for defining a final, immutable data-carrier class
 - private data fields
 - constructor
 - getters for each field
 - equals, hashCode
 - toString

The DAO Pattern

- the DAO (data access object) layer encapsulates the SQL
 - the rest of the application works only with Java objects
- one DAO class per domain object
 - e.g. domain objects Book, Loan, Member → DAO classes BookDao, LoanDao, MemberDao
- methods mirror CRUD
 - findById, findAll, insert, update, delete
- take and return domain objects, not ResultSets or JSON
 - all SQL and JDBC is (only) inside
 - e.g. Book
- handle “might not exist” by returning an Optional
- for now each method opens its own connection
 - will need a different mechanism to support atomic transactions where several statements need to succeed or fail together

BookDAO

- one method per kind of access needed by the endpoints

```
public class BookDao {  
    // GET /books  
    public List<Book> findAll() throws SQLException { ... }  
  
    // GET /books?author=...  
    public List<Book> findByAuthor(String author) throws SQLException { ... }  
  
    // GET /books/{id}  
    public Optional<Book> findById(int id) throws SQLException { ... }  
  
    // POST /books → returns the new book_id  
    public int insert(Book b) throws SQLException { ... }  
  
    // PUT /books/{id} → false if no such book  
    public boolean update(Book b) throws SQLException { ... }  
  
    // DELETE /books/{id} → false if no such book  
    public boolean delete(int id) throws SQLException { ... }  
}
```

Optional<T>

- Optional<T> is a container that holds either one value of type T or nothing
 - meant for return values – not to be used for instance variables, parameters, or collections
- advantages
 - the method signature reflects intent
 - an Optional<Book> return type tells you it might not find anything
 - a Book return type doesn't
 - separates “not found” from “failed”
 - a missing book is a normal outcome (Optional.empty()), but a lost connection is a failure (SQLException)
 - helps avoid bugs from ignored null returns
 - the caller has to unwrap the Optional so must think about the empty case

Mapping Results to Objects

- from a row to a Book

bookId is Integer rather than int because a book that hasn't been inserted into the database yet has no ID

```
public record Book(Integer bookId, String isbn, String title, String author,
    LocalDate publishedOn, BigDecimal replacementCost,
    boolean available) { }
```

```
public class BookDAO {
    ...
    private static Book map(ResultSet rs) throws SQLException {
        return new Book(
            rs.getInt("book_id"),
            rs.getString("isbn"),
            rs.getString("title"),
            rs.getString("author"),
            rs.getObject("published_on", LocalDate.class),
            rs.getBigDecimal("replacement_cost"),
            rs.getBoolean("available"));
    }
}
```

- mapping is encapsulated in a helper method – every query that returns books maps them in the same way

```
/**
 * Read one book. An id with no matching row is normal, so it's Optional.empty,
 * not an exception.
 */
public Optional<Book> findById(int id) throws SQLException {
    String sql = "SELECT " + COLUMNS + " FROM BOOKS WHERE book_id = ?";
    try (Connection conn = DB.getConnection(); PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setInt(1, id);
        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
                return Optional.of(map(rs));
            }
            return Optional.empty();
        }
    }
}

/** Read every book, ordered by title. */
public List<Book> findAll() throws SQLException {
    String sql = "SELECT " + COLUMNS + " FROM BOOKS ORDER BY title";
    List<Book> books = new ArrayList<>();
    try (Connection conn = DB.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            books.add(map(rs));
        }
    }
    return books;
}
```

Services

- services carry out the application's operations and enforce its rules
 - i.e. implement the use cases and enforce the policies
- organization
 - one service per domain area, usually named for its main domain object
 - e.g. BookService, LoanService, MemberService
 - one service method per use case, named in domain terms
 - e.g. checkOut, returnBook
- each endpoint calls the service method for its use case
 - a service method may call several DAOs

Services – Library

use case	endpoint	service method	DAO calls
browse all books	GET /books	BookService.findAll	BookDao.findAll
find books by an author	GET /books?author=	BookService.findByAuthor	BookDao.findByAuthor
add a new book	POST /books	BookService.create	BookDao.insert, findById
view a book's details	GET /books/{id}	BookService.findById	BookDao.findById
edit a book's details	PUT /books/{id}	BookService.update	BookDao.update
remove a book	DELETE /books/{id}	BookService.delete	BookDao.delete
list loans	GET /loans	LoanService.listLoans	LoanDao.findAll
view one loan	GET /loans/{id}	LoanService.findLoan	LoanDao.findById
check out a book	POST /loans	LoanService.checkOut	BookDao.findById, LoanDao.insert, BookDao.setAvailable
return a book	PUT /loans/{id}/return	LoanService.returnBook	LoanDao.findById, LoanDao.markReturned, BookDao.setAvailable

```

public class BookService {
    private final BookDao dao;
    public BookService(BookDao dao) {
        this.dao = dao;
    }
    public Optional<Book> findById(int id) throws SQLException {
        return dao.findById(id);
    }
    public Book create(NewBook nb) throws SQLException {
        if (nb.title() == null || nb.title().isBlank()) {
            throw new IllegalArgumentException("title is required");
        }
        Book book = new Book(null, nb.isbn(), nb.title(), nb.author(),
            nb.publishedOn(), nb.replacementCost(), true);
        int id = dao.insert(book);
        Optional<Book> created = dao.findById(id);
        if (created.isEmpty()) {
            throw new IllegalStateException("Book " + id +
                " was inserted but can't be found");
        }
        return created.get();
    }
}

```

DAO is passed through the constructor rather than being created by the service

NewBook reflects what the client actually sends – different from Book

- services deal with domain terms – Book, Optional, exceptions
 - no SQL, no status codes

2

Services

```

// LoanService
public Loan checkOut(int memberId, int bookId) throws SQLException {
    Optional<Book> book = bookDao.findById(bookId);
    if (book.isEmpty()) {
        throw new NoSuchElementException("No book " + bookId);
    }
    if (!book.get().available()) {
        throw new IllegalStateException("Book " + bookId + " is checked out");
    }
    LocalDate today = LocalDate.now();
    int loanId = loanDao.insert(new Loan(null, bookId, memberId,
        today, today.plusDays(LOAN_DAYS), null));
    bookDao.setAvailable(bookId, false);
    Optional<Loan> created = loanDao.findById(loanId);
    if (created.isEmpty()) {
        throw new IllegalStateException("Loan " + loanId +
            " was inserted but can't be found");
    }
    return created.get();
}

```

- not yet safe regarding atomicity and concurrency
 - failure partway through or simultaneous requests may leave in an inconsistent state

3

Validation and DB Constraints

- do both
- the service validates first
 - can provide a clear error message
 - rejects bad input before any SQL runs
- the database is the last line of defense
 - should not be the first feedback the user sees
 - catches what the service can't
 - e.g. interference from simultaneous requests
 - e.g. other clients may bypass the service

Objects ↔ JSON

```

Book b = new Book(42, "9780345391803", "The Hitchhiker's Guide to the Galaxy",
    "Douglas Adams", LocalDate.of(1979, 10, 12),
    new BigDecimal("12.99"), true);

```

```

{
  "bookId": 42,
  "isbn": "9780345391803",
  "title": "The Hitchhiker's Guide to the Galaxy",
  "author": "Douglas Adams",
  "publishedOn": "1979-10-12",
  "replacementCost": 12.99,
  "available": true
}

```

} JSON

- Javalin uses the Jackson library to turn objects into JSON (serialize) and JSON into objects (deserialize)
 - in `ctx.bodyAsClass(...)`, `ctx.json(...)`
- JSON field names come from Java (Book class), not from the table
 - JSON format is the API's contract with clients – doesn't have to match the database schema

8