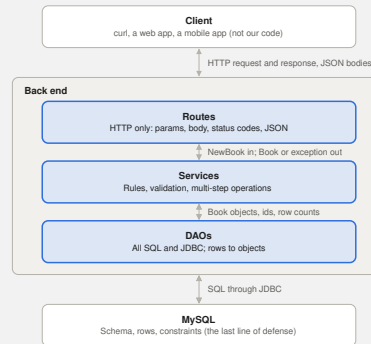


The Story So Far

- layered back end
 - each layer only calls the layer below it
- domain objects
 - Java record for each entity
- DAOs
 - one for each domain object
 - contains all SQL and JDBC
- services – the application's rules
 - validate, enforce policies, orchestrate multi-step operations
 - work with domain objects
 - one service class per domain area, one service method per use case



57

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

58

DTOs

- what is sent between client and server may not be the whole domain object

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

- client creating a book doesn't send `bookId` (autogenerated) or `available` (defaults to true)
- a *DTO* (data transfer object) defines exactly what is sent
 - makes the rule structural – only the fields in the DTO are sent

```
public record NewBook(String isbn, String title, String author,
    LocalDate publishedOn, BigDecimal replacementCost) { }
```

Routes With Javalin

- a *route* is the connection between an endpoint and the code that handles it
 - term is often used to refer to both the act of registering such a component and the handler itself
- the *routes layer* consists of the classes that hold and register handlers

```
public class BookRoutes {
    private final BookService service;

    public BookRoutes(BookService service) {
        this.service = service;
    }

    public void register(JavalinConfig config) {
        config.routes.get("/books", ctx -> list(ctx));
        config.routes.get("/books/{id}", ctx -> getOne(ctx));
        config.routes.post("/books", ctx -> create(ctx));
    }

    ...
}
```

Routes With Javalin

```
public class BookRoutes {
    ...
    private void create(Context ctx) throws SQLException {
        NewBook body = ctx.bodyAsClass(NewBook.class);
        try {
            Book created = service.create(body);
            ctx.status(201);
            ctx.header("Location", "/books/" + created.bookId());
            ctx.json(created);
        } catch (IllegalArgumentException e) {
            ctx.status(400).result(e.getMessage());
        }
    }
    ...
}
```

- accessing elements of the request
 - bodyAsClass for the JSON body
 - pathParam for /books/{id}
 - queryParams for ?author=
- the route chooses the status code and Location header

61

Routes With Javalin

```
public class BookRoutes {
    ...
    private void getOne(Context ctx) throws SQLException {
        int id = Integer.parseInt(ctx.pathParam("id"));
        Optional<Book> book = service.findById(id);
        if (book.isPresent()) {
            ctx.json(book.get());
        } else {
            ctx.status(404);
        }
    }
    private void list(Context ctx) throws SQLException {
        String author = ctx.queryParam("author");
        List<Book> books = (author == null) ? service.findAll() :
            service.findByAuthor(author);
        ctx.json(books);
    }
}
```

CPSC 343: Database Theory and Practice • Fall 2026

62

Main Program

```
public static void main(String[] args) {
    BookDao bookDao = new BookDao();
    BookService bookService = new BookService(bookDao);
    BookRoutes bookRoutes = new BookRoutes(bookService);

    // Javalin 7: all routes are registered up front, inside the config block
    Javalin.create(config -> {
        config.jetty.port = 7000;
        bookRoutes.register(config);
    }).start();
}
```

```
curl -i localhost:7000/books/1
curl -i "localhost:7000/books?author=Douglas%20Adams"
curl -i -X POST localhost:7000/books -H "Content-Type: application/json" \
-d '{"isbn": "9780141036144", \
  "title": "Nineteen Eighty-Four", "author": "George Orwell"}'
```

CPSC 343: Database Theory and Practice • Fall 2026

63

Putting It All Together – Basic Application

element	does
Route	parses the path id, query parameters, and JSON body; calls one service method; turns the service's failures into status codes
	sets the success status, any headers (such as Location after a create), and the JSON body
Service	enforces the library's rules (a book must exist and be available) and runs the steps of a use case in order, across DAOs
DAO	runs the SQL through JDBC and maps rows to domain objects
MySQL	stores the data and enforces the constraints (UNIQUE, foreign keys, NOT NULL): the last line of defense if a check above was missed

CPSC 343: Database Theory and Practice • Fall 2026

64

Error Handling

POST /books (an ISBN that already exists)

```
500 Internal Server Error
java.sql.SQLException: Duplicate entry '9780345391803'
for key 'BOOKS.isbn'
  at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:...)
  ...
  at library.dao.BookDao.insert(BookDao.java:...)
  at library.service.BookService.create(BookService.java:...)
  at library.routes.BookRoutes.create(BookRoutes.java:...)
  at library.routes.BookRoutes.lambda$register$2(BookRoutes.java:...)
  at io.javalin.router.Endpoint.handle(...)
```

- whose fault is it? – to decide the status code
 - the client's (bad input, clash with existing data) → 4xx
 - the server's (a bug, the database is down) → 5xx
- who is the audience? – to decide what the message is
 - the client – what went wrong, in domain terms – in the response
 - the server – full details, stack trace, exception error message – in the server log
 - never put internal DB details in the response – gives an attacker tools

Whose Fault?

code	meaning	library examples
400 Bad Request	malformed or invalid input	no title GET /books/abc
401 Unauthorized	not logged in	missing or expired token
403 Forbidden	logged in, not allowed	a member tries POST /books (requires a librarian)
404 Not Found	no such resource (or not yours)	GET /books/9999
409 Conflict	clashes with the current state	duplicate ISBN checking out a book that's already out
500 Internal Server Error	a server bug	a NullPointerException
503 Service Unavailable	server is temporarily unable to satisfy the request	the database is unreachable

- 4xx – the client's fault
 - the client can fix it by changing the request
- 5xx – the server's fault
 - nothing the client sends will help

Sources of Errors

Three sources of errors –

- service rules – e.g. no title, book already out
 - generated within the back end – we control the exception type
- constraint violations
- other SQLExceptions
 - generated by the database – must translate the error
 - decide by code, never by message text – vendor code precisely identifies the reason; message text can vary from one MySQL version to the next

MySQL error	vendor code / SQLState	response
Duplicate key (UNIQUE, primary key)	1062	409
Parent row missing (foreign key on insert/update)	1452	404
Row still referenced (ON DELETE RESTRICT)	1451	409
NOT NULL or CHECK violated	1048, 3819	400
Can't reach the database	SQLState class 08	503
Anything else		500

Defining Exception Types

```
public class AppException extends RuntimeException {
    public AppException(String message) { super(message); }
    public AppException(String message, Throwable cause) { super(message, cause); }
}

public class ValidationException extends AppException { ... } // bad input
public class UnauthorizedException extends AppException { ... } // not logged in
public class ForbiddenException extends AppException { ... } // logged in, not allowed
public class NotFoundException extends AppException { ... } // no such thing (or not yours)
public class ConflictException extends AppException { ... } // clashes with current state
public class UnavailableException extends AppException { ... } // temporarily can't do it
```

- each exception class names a kind of failure, not a specific HTTP code
- unchecked (extend RuntimeException) so they pass through layers without requiring throws clauses
 - allows handling of errors thrown in any layer in one place (at the top)
 - standard practice – the alternative (checked exceptions, with the required throws declarations) couples the layers, makes change expensive, and clashes with lambdas

Translating SQL Errors

- the DAO is the only layer that knows SQL, so it is the one that translates SQL errors
 - DAO and service methods no longer declare throws SQLException

```
public final class SqlErrors {  
    // define a helper class to encapsulate the translations  
    public static RuntimeException translate(SQLException e) {  
        String state = e.getSQLState();  
        if (state != null && state.startsWith("08")) {  
            return new UnavailableException("Database unavailable", e);  
        }  
        int code = e.getErrorCode();  
        if (code == 1062) {  
            return new ConflictException("Duplicate value", e);  
        } else if (code == 1451) {  
            return new ConflictException("Still referenced by other records", e);  
        } else if (code == 1452) {  
            return new NotFoundException("Referenced record does not exist", e);  
        } else if (code == 1048 || code == 3819) {  
            return new ValidationException("Invalid value", e);  
        } else {  
            return new AppException("Database error", e);  
        }  
    }  
}
```

keep the original SQLException as the cause – allows for recording the detailed error in the server log

Translating SQL Errors

```
// BookDao: POST /books → returns the new book_id  
public int insert(Book b) {  
    String sql = "INSERT INTO BOOKS (isbn, title, author, published_on, " +  
        "replacement_cost) VALUES (?, ?, ?, ?, ?)";  
    try (Connection conn = Db.getConnection();  
        PreparedStatement ps =  
            conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {  
        ...  
    } catch (SQLException e) {  
        if (e.getErrorCode() == 1062) {  
            throw new ConflictException("A book with ISBN " + b.isbn() +  
                " already exists", e);  
        }  
        throw SqlErrors.translate(e);  
    }  
}
```

no longer throws SQLException

handles 1062 in order to provide a more specific error message than what is defined in SqlErrors

Designing Error Responses

```
public record ErrorResponse(int status, String message) {}  
  
private static void send(Context ctx, int status, String message) {  
    ctx.status(status).json(new ErrorResponse(status, message));  
}
```

build the error response: set the HTTP status line (ctx.status(...)) and set the response body to contain the status and message (ctx.json(...))

- every error goes to the client in the same format

```
{  
  "status": 409,  
  "message": "A book with ISBN 9780345391803 already exists"  
}
```

 - allows the client to have one piece of code that handles every error – read status, show message
- never include stack traces, SQL, table or column names, driver messages in responses – log on the server instead
 - not meaningful to a user
 - useful to an attacker

Setting Up Error Handlers

```
public final class ErrorHandler {  
    private static final Logger LOG = Logger.getLogger(ErrorHandler.class.getName());  
  
    public static void register(JavalinConfig config) {  
        config.routes.exception(ValidationException.class,  
            (e, ctx) -> send(ctx, 400, e.getMessage()));  
        config.routes.exception(UnauthorizedException.class,  
            (e, ctx) -> send(ctx, 401, e.getMessage()));  
        config.routes.exception(ForbiddenException.class,  
            (e, ctx) -> send(ctx, 403, e.getMessage()));  
        config.routes.exception(NotFoundException.class,  
            (e, ctx) -> send(ctx, 404, e.getMessage()));  
        config.routes.exception(ConflictException.class,  
            (e, ctx) -> send(ctx, 409, e.getMessage()));  
        config.routes.exception(UnavailableException.class, (e, ctx) -> {  
            LOG.log(Level.SEVERE, "Database unavailable", e);  
            send(ctx, 503, "Service temporarily unavailable");  
        });  
        config.routes.exception(Exception.class, (e, ctx) -> {  
            LOG.log(Level.SEVERE, "Unhandled error on " +  
                ctx.method() + " " + ctx.path(), e);  
            send(ctx, 500, "Internal server error");  
        });  
    }  
  
    private static void send(Context ctx, int status, String message) {  
        ctx.status(status).json(new ErrorResponse(status, message));  
    }  
}
```

utilizes Java's logging framework

each handler serves all routes

for 4xx, the message is safe to show – send to client

for 5xx, send a generic message to the client and log the details

default handler to catch exceptions due to bugs

Setting Up Error Handlers

- main

```
// in main
Javalin.create(config -> {
    config.jetty.port = 7000;
    ErrorHandler.register(config); ← do the setup
    bookRoutes.register(config);
    // ... the other routes
}).start();
```

Result: Simplified Routes

```
// BookRoutes
private void getOne(Context ctx) {
    ctx.json(service.get(pathId(ctx)));
}
private void create(Context ctx) {
    Book created = service.create(ctx.bodyAsClass(NewBook.class));
    ctx.status(201).header("Location",
        "/books/" + created.bookId()).json(created);
}
static int pathId(Context ctx) {
    try {
        return Integer.parseInt(ctx.pathParam("id"));
    } catch (NumberFormatException e) {
        throw new ValidationException("id must be a whole number");
    }
}

// BookService
public Book get(int id) {
    Optional<Book> book = dao.findById(id);
    if (book.isEmpty()) {
        throw new NotFoundException("No book with id " + id);
    }
    return book.get();
}
```

routes only describe the success path
– failures are exceptions, handled by the configured error handlers

name change (findById to get) reflects the change in contract making a missing book an error – convention is "find" for "might not find it" and "get" for "it must exist"

with a central handler available to turn exceptions into responses, the service can treat a missing book as an error instead of handing an Optional back to routes to deal with

Comparison

```
// BookRoutes
private void getOne(Context ctx) throws SQLException {
    int id = Integer.parseInt(ctx.pathParam("id"));
    Optional<Book> book = service.findById(id);
    if (book.isPresent()) {
        ctx.json(book.get());
    } else {
        ctx.status(404);
    }
}
private void create(Context ctx) throws SQLException {
    NewBook body = ctx.bodyAsClass(NewBook.class);
    try {
        Book created = service.create(body);
        ctx.status(201);
        ctx.header("Location", "/books/" + created.bookId());
        ctx.json(created);
    } catch (IllegalArgumentException e) {
        ctx.status(400).result(e.getMessage());
    }
}

// BookService
public Optional<Book> findById(int id) throws SQLException {
    return dao.findById(id);
}
```

check is now done by the service

error is now handled by the registered exception handler

error is now handled by the registered exception handler

Putting It All Together – With Error Handling

element	does	can fail with
Route	parses the path id, query parameters, and JSON body; calls one service method	400 (bad id or body), 403
	sets the success status, any headers (such as Location after a create), and the JSON body	on success: 200, 201, 204
Service	enforces the library's rules (a book must exist and be available) and runs the steps of a use case in order, across DAOs	400, 404, 409
DAO	runs the SQL through JDBC, maps rows to domain objects, and translates any SQLException into one of our exceptions	409, 404, 400, 503
MySQL	stores the data and enforces the constraints (UNIQUE, foreign keys, NOT NULL): the last line of defense if a check above was missed	(reported through the DAO)
ErrorHandler	catches any exception thrown above, chooses the status code, logs the details of 5xx failures, and sends the error JSON	500 for anything unexpected