The EpicBook! - Complete Project Documentation

Table of Contents

---

1. Project Overview

The EpicBook! is a full-stack e-commerce web application for an online bookstore built using Node.js, Express, MySQL, and deployed on AWS EC2 with Nginx as a reverse proxy.

Features:

- Browse book catalog
- View book details (title, author, price)
- Add books to shopping cart
- Process checkout orders

Architecture Type:

Monolithic - All components (frontend, backend, database) run on a single EC2 instance.

---

2. Technology Stack

Frontend

Technology	Purpose	Why Used
HTML	Web page structure	Standard markup language for web pages
CSS	Styling and layout	Makes the application visually appealing
JavaScript	Client-side interactivity	Handles cart functionality, form validation
EJS	Template engine	Dynamically generates HTML from data

Backend

Node.js (v17.9.1)

What: JavaScript runtime for server-side execution

Why Used:

- **JavaScript Everywhere:** Same language for frontend and backend
- **Non-Blocking I/O:** Handles multiple users simultaneously
- **Fast Execution:** V8 engine compiles JavaScript to machine code
- **NPM Ecosystem:** Access to 1.5M+ packages
- **Lightweight:** Works well on small servers (t2.micro)

Role in Project:

- Runs the backend server
- Processes business logic
- Handles HTTP requests/responses

Express.js (v4.18.0)

What: Web application framework for Node.js

Why Used:

- **Simple Routing:** Easy to define API endpoints
- **Middleware Support:** Modular request processing
- **Fast Development:** Minimal boilerplate code

Role in Project:

- Defines routes (`/books` , `/cart` , `/checkout`)
- Handles HTTP methods (GET, POST, PUT, DELETE)
- Manages sessions and cookies

---

mysql2 (v2.3.3)

What: MySQL client for Node.js

Why Used:

- Connects Node.js to MySQL database
- Supports Promises for cleaner async code
- Better performance than legacy mysql package

Role in Project:

- Establishes database connection
- Executes SQL queries
- Returns results to Node.js

Database

MySQL 5.7

What: Relational database management system

Why Used:

- **Structured Data:** Books, authors, orders have defined relationships
- **ACID Compliance:** Ensures data integrity during transactions
- **SQL Language:** Standardized, widely-known query language
- **Performance:** Optimized for read-heavy operations (browsing books)
- **Reliability:** Battle-tested for 25+ years

Role in Project:

- Stores book catalog (titles, authors, prices)
- Maintains relationships (authors → books → orders)
- Ensures data integrity with foreign keys

Web Server

Nginx (v1.20.1)

What: High-performance web server and reverse proxy

Why Used:

- **Standard Port:** Allows access via port 80 (http://ip instead of http://ip:8080)
- **Reverse Proxy:** Hides backend architecture
- **Security Layer:** Protects Node.js from direct exposure
- **Performance:** Efficient static file serving
- **Scalability:** Can add load balancing in future

Role in Project:

- Receives requests on port 80
- Forwards to Node.js on port 8080
- Returns responses to users

Without Nginx:

```
❌ Users must access: http://54.123.45.67:8080
❌ Node.js directly exposed to internet
❌ No SSL termination point
```

With Nginx:

```
 Users access: http://54.123.45.67
 Node.js hidden behind proxy
 Can add SSL (HTTPS) easily
```

Cloud Infrastructure

AWS EC2 (t2.micro)

What: Virtual server in the cloud

Why Used:

- **On-Demand:** Pay only for usage
- **Scalable:** Easy to upgrade resources
- **Reliable:** 99.99% uptime SLA
- **Free Tier:** First 12 months free (750 hours/month)

Specifications:

- **vCPUs:** 1
- **Memory:** 1 GB RAM
- **Storage:** 8 GB EBS
- **OS:** Amazon Linux 2

AWS Security Groups

What: Virtual firewall for EC2 instances

Why Used:

- Controls inbound/outbound traffic
- Instance-level security
- Stateful (return traffic auto-allowed)

Configuration:

```
Inbound Rules:
- Port 22 (SSH): Your IP only → Secure remote access
- Port 80 (HTTP): 0.0.0.0/0 → Public web access
- Port 8080 (Node.js): Localhost only → Internal communication

Outbound Rules:
- All traffic allowed
```

Version Control

Git/GitHub

What: Distributed version control system

Why Used:

- Track code changes
- Collaborate with team
- Clone repository to any server
- Version history backup

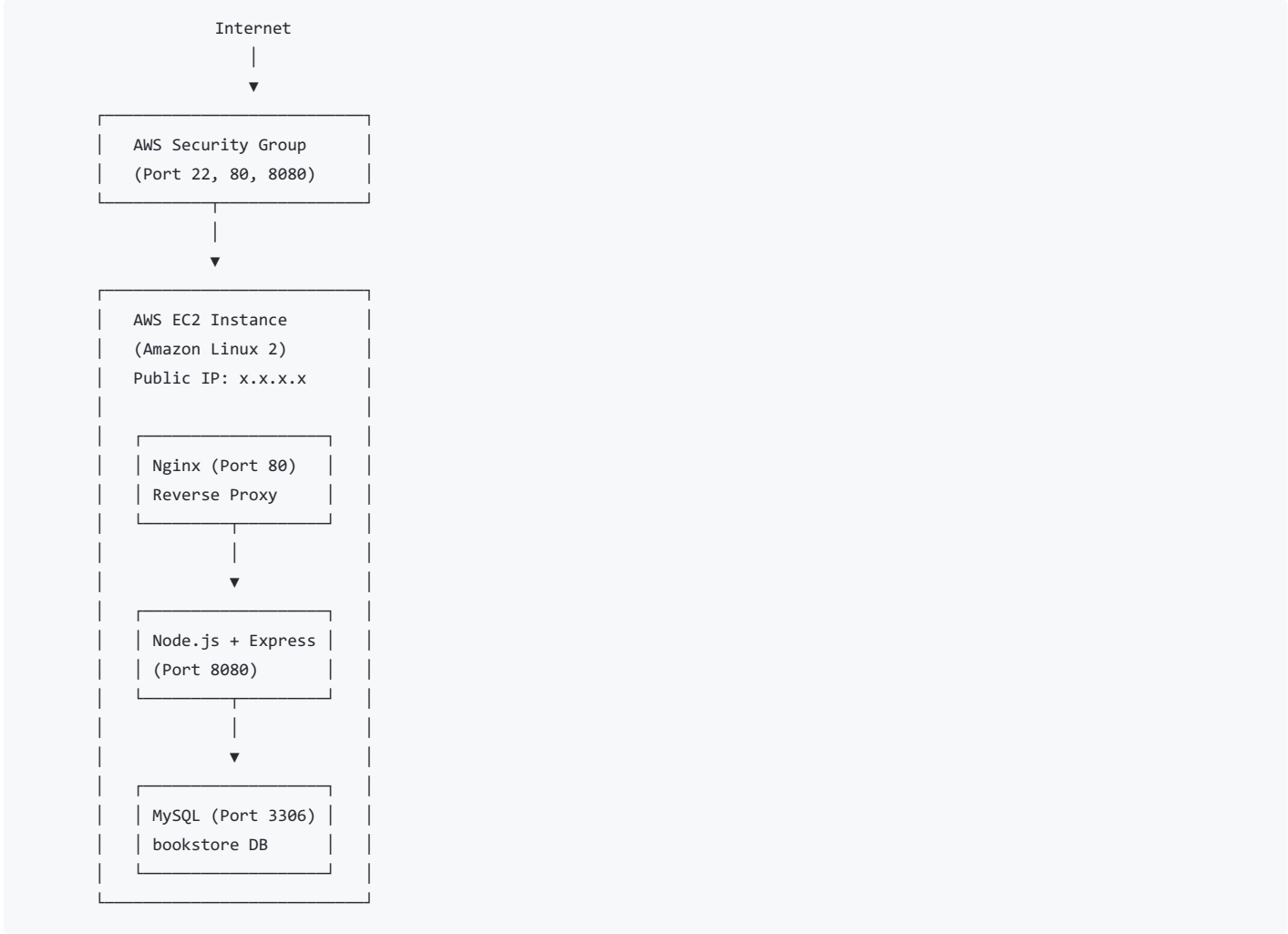
Role in Project:

- Download application source code
- Access database schema files
- Get project dependencies list

---

3. Project Architecture

High-Level Architecture



Database Schema

```
authors
├── author_id (PK)
├── author_name
└── bio

books
├── book_id (PK)
├── title
├── author_id (FK → authors)
├── price
├── isbn
├── description
└── stock_quantity

orders
├── order_id (PK)
├── customer_name
├── customer_email
├── total_amount
├── order_status
└── order_date

order_items
├── item_id (PK)
├── order_id (FK → orders)
├── book_id (FK → books)
├── quantity
└── price
```

4. Setup and Installation

Step 1: Launch EC2 Instance

```
1. AWS Console → EC2 → Launch Instance
2. Name: theepicbook
3. AMI: Amazon Linux 2
4. Instance Type: t2.micro
5. Key Pair: Create new (Linux-VM.pem)
6. Security Group:
   - SSH (22): Your IP
   - HTTP (80): 0.0.0.0/0
   - Custom TCP (8080): 0.0.0.0/0
7. Storage: 8 GB gp3
8. Launch Instance
```

Step 2: Connect to EC2

```
# Secure key file
chmod 400 Linux-VM.pem

# Connect via SSH
ssh -i Linux-VM.pem ec2-user@54.123.45.67
```

Step 3: Update System

```
sudo yum update -y
```

What it does: Updates all system packages to latest versions

Step 4: Install Git

```
sudo yum install git -y
```

What it does: Installs Git version control tool

Step 5: Clone Repository

```
git clone https://github.com/pravinmishraaws/theepicbook
cd theepicbook
```

What it does:

- Downloads application source code
- Navigates into project directory

Step 6: Install MySQL Server 5.7

```
# Add MySQL repository
sudo yum install -y https://dev.mysql.com/get/mysql57-community-release-el7-11.noarch.rpm

# Import GPG key
sudo rpm --import https://repo.mysql.com/RPM-GPG-KEY-mysql-2022

# Disable MySQL 8.0, enable MySQL 5.7
sudo yum-config-manager --disable mysql80-community
sudo yum-config-manager --enable mysql57-community

# Install MySQL Server
sudo yum install -y mysql-community-server

# Start MySQL
sudo systemctl start mysqld

# Enable auto-start on boot
sudo systemctl enable mysqld
```

What it does:

- Installs MySQL 5.7 database server
- Starts MySQL service
- Configures to start automatically on reboot

Step 7: Configure MySQL

```
# Get temporary password
sudo grep 'temporary password' /var/log/mysqld.log

# Login to MySQL
mysql -u root -p
# Enter temporary password

# Change root password
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'NewPassword123!';

# Create database
mysql> CREATE DATABASE bookstore;

# Exit
mysql> EXIT;
```

What it does:

- Secures root account with strong password
- Creates empty bookstore database

Step 8: Load Database Schema and Data

```
# Load schema (creates table structure)
mysql -u root -p bookstore < db/BuyTheBook_Schema.sql

# Load authors (sample data)
mysql -u root -p bookstore < db/author_seed.sql

# Load books (sample data)
mysql -u root -p bookstore < db/books_seed.sql
```

What it does:

- Creates 4 tables: authors, books, orders, order_items
- Inserts 10 sample authors
- Inserts 20+ sample books

Step 9: Install Node.js

```
# Install NVM (Node Version Manager)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash

# Load NVM
source ~/.nvm/nvm.sh

# Install Node.js v17
nvm install v17

# Verify
node -v  # v17.9.1
npm -v   # 8.11.0
```

What it does:

- Installs NVM for managing Node.js versions
- Installs Node.js v17 and npm

Step 10: Install Dependencies

```
npm install
```

What it does:

- Reads package.json
- Downloads and installs all required packages (express, mysql2, ejs, etc.)
- Creates node_modules/ folder with 125+ packages

Step 11: Configure Database Connection

```
vi config/config.json
```

Update password:

```
{
  "development": {
    "username": "root",
    "password": "NewPassword123!",
    "database": "bookstore",
    "host": "localhost",
    "port": 3306,
    "dialect": "mysql"
  }
}
```

What it does: Tells Node.js how to connect to MySQL database

Step 12: Install Nginx

```
# Install EPEL repository
sudo yum install -y epel-release

# Install Nginx
sudo yum install -y nginx

# Start Nginx
sudo systemctl start nginx

# Enable auto-start
sudo systemctl enable nginx
```

What it does:

- Installs Nginx web server
- Starts Nginx service
- Configures to start on boot

Step 13: Configure Nginx Reverse Proxy

```
sudo vi /etc/nginx/conf.d/theepicbooks.conf
```

Add configuration:

```
server {
    listen 80;
    server_name 54.123.45.67;  # Your EC2 public IP

    location / {
        proxy_pass http://localhost:8080;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Test and reload:

```
sudo nginx -t
sudo systemctl reload nginx
```

What it does:

- Configures Nginx to listen on port 80
- Forwards all requests to Node.js on port 8080

Step 14: Start Application

```
node server.js
```

Output:

```
Connected to MySQL database
Server running on port 8080
```

What it does: Starts Node.js application server

Step 15: Access Application

Open browser and navigate to:

```
http://54.123.45.67
```

You should see: The EpicBook homepage with book listings!

---

5. Code Explanation

File Structure

```
theepicbook/
├── server.js              # Main application entry point
├── package.json           # Dependencies and project metadata
├── config/
│   └── config.json        # Database connection settings
├── db/
│   ├── BuyTheBook_Schema.sql   # Database structure
│   ├── author_seed.sql         # Sample authors
│   └── books_seed.sql          # Sample books
├── routes/
│   ├── books.js           # Book-related routes
│   ├── cart.js            # Cart routes
│   └── orders.js          # Order routes
├── models/
│   ├── Book.js            # Book data model
│   ├── Author.js          # Author data model
│   └── Order.js           # Order data model
├── views/
│   ├── index.ejs          # Homepage template
│   ├── books.ejs          # Book listing page
│   └── cart.ejs           # Shopping cart page
└── public/
    ├── css/               # Stylesheets
    ├── js/                # Frontend JavaScript
    └── images/            # Images
```

---

server.js (Main Application File)

```
// ============================================
// 1. Import Dependencies
// ============================================
const express = require('express');
const mysql = require('mysql2');
const bodyParser = require('body-parser');
const session = require('express-session');

// ============================================
// 2. Create Express Application
// ============================================
const app = express();

// ============================================
// 3. Configure Middleware
// ============================================

// Parse JSON request bodies
app.use(bodyParser.json());

// Parse URL-encoded request bodies (form data)
```

```
app.use(bodyParser.urlencoded({ extended: true }));

// Session management for shopping cart
app.use(session({
    secret: 'bookstore-secret-key',  // Encryption key
    resave: false,                   // Don't save unchanged sessions
    saveUninitialized: true,         // Create session for new visitors
    cookie: { maxAge: 3600000 }      // Session expires in 1 hour
}));

// Set EJS as template engine
app.set('view engine', 'ejs');

// Serve static files (CSS, JS, images)
app.use(express.static('public'));

// ============================================
// 4. Database Connection
// ============================================
const config = require('./config/config.json');
const dbConfig = config.development;

const connection = mysql.createConnection({
    host: dbConfig.host,           // localhost
    user: dbConfig.username,       // root
    password: dbConfig.password,   // NewPassword123!
    database: dbConfig.database,   // bookstore
    port: dbConfig.port            // 3306
});

// Connect to MySQL
connection.connect((error) => {
    if (error) {
        console.error('Database connection failed:', error.message);
        process.exit(1);  // Exit if can't connect
    }
    console.log('✅ Connected to MySQL database');
});

// ============================================
// 5. Define Routes
// ============================================

// Homepage Route
app.get('/', (req, res) => {
    res.render('index');
});

// Get All Books Route
app.get('/books', (req, res) => {
    const sql = 'SELECT * FROM books';

    connection.query(sql, (error, results) => {
        if (error) {
            console.error('Query error:', error);
            return res.status(500).json({ error: 'Database error' });
        }

        // Render books.ejs template with data
        res.render('books', { books: results });
    });
});

// Get Single Book Route
app.get('/books/:id', (req, res) => {
```

```
    const bookId = req.params.id;
    const sql = 'SELECT * FROM books WHERE book_id = ?';

    connection.query(sql, [bookId], (error, results) => {
        if (error) {
            return res.status(500).json({ error: 'Database error' });
        }

        if (results.length === 0) {
            return res.status(404).json({ error: 'Book not found' });
        }

        res.render('book-detail', { book: results[0] });
    });
});

// Add to Cart Route
app.post('/api/cart/add', (req, res) => {
    const { book_id, quantity } = req.body;

    // Initialize cart if doesn't exist
    if (!req.session.cart) {
        req.session.cart = [];
    }

    // Add item to cart
    req.session.cart.push({
        book_id: book_id,
        quantity: quantity
    });

    res.json({
        success: true,
        message: 'Book added to cart',
        cartCount: req.session.cart.length
    });
});

// View Cart Route
app.get('/cart', (req, res) => {
    const cart = req.session.cart || [];

    if (cart.length === 0) {
        return res.render('cart', { items: [], total: 0 });
    }

    // Get book details for cart items
    const bookIds = cart.map(item => item.book_id);
    const sql = 'SELECT * FROM books WHERE book_id IN (?)';

    connection.query(sql, [bookIds], (error, results) => {
        if (error) {
            return res.status(500).json({ error: 'Database error' });
        }

        // Calculate total
        let total = 0;
        const cartItems = cart.map(cartItem => {
            const book = results.find(b => b.book_id === cartItem.book_id);
            const itemTotal = book.price * cartItem.quantity;
            total += itemTotal;

            return {
                ...book,
                quantity: cartItem.quantity,
                itemTotal: itemTotal
```

```
                itemTotal: itemTotal
            };
        });

        res.render('cart', { items: cartItems, total: total });
    });
});

// Checkout Route
app.post('/api/checkout', (req, res) => {
    const { customer_name, customer_email } = req.body;
    const cart = req.session.cart || [];

    if (cart.length === 0) {
        return res.status(400).json({ error: 'Cart is empty' });
    }

    // Get book details for pricing
    const bookIds = cart.map(item => item.book_id);
    const sql = 'SELECT * FROM books WHERE book_id IN (?)';

    connection.query(sql, [bookIds], (error, books) => {
        if (error) {
            return res.status(500).json({ error: 'Database error' });
        }

        // Calculate total
        let total = 0;
        cart.forEach(cartItem => {
            const book = books.find(b => b.book_id === cartItem.book_id);
            total += book.price * cartItem.quantity;
        });

        // Insert order
        const orderSql = `
            INSERT INTO orders (customer_name, customer_email, total_amount)
            VALUES (?, ?, ?)
        `;

        connection.query(orderSql, [customer_name, customer_email, total],
        (error, orderResult) => {
            if (error) {
                return res.status(500).json({ error: 'Order failed' });
            }

            const orderId = orderResult.insertId;

            // Insert order items
            const itemPromises = cart.map(cartItem => {
                return new Promise((resolve, reject) => {
                    const book = books.find(b => b.book_id === cartItem.book_id);
                    const itemSql = `
                        INSERT INTO order_items (order_id, book_id, quantity, price)
                        VALUES (?, ?, ?, ?)
                    `;

                    connection.query(itemSql,
                        [orderId, cartItem.book_id, cartItem.quantity, book.price],
                        (error, result) => {
                            if (error) reject(error);
                            else resolve(result);
                        }
                    );
                });
            });
```

```
            // Wait for all items to be inserted
            Promise.all(itemPromises)
                .then(() => {
                    // Clear cart
                    req.session.cart = [];

                    res.json({
                        success: true,
                        order_id: orderId,
                        message: 'Order placed successfully'
                    });
                })
                .catch(error => {
                    res.status(500).json({ error: 'Failed to save order items' });
                });
        });
    });
});

// ============================================
// 6. Start Server
// ============================================
const PORT = process.env.PORT || 8080;

app.listen(PORT, () => {
    console.log(`🚀 Server running on port ${PORT}`);
});
```

Code Breakdown

1. Import Dependencies

```
const express = require('express');
const mysql = require('mysql2');
```

What it does: Loads required Node.js packages

---

2. Create Express App

```
const app = express();
```

What it does: Creates an Express application instance

---

3. Middleware

```
app.use(bodyParser.json());
```

What it does: Parses incoming JSON requests

```
app.use(session({ ... }));
```

What it does: Manages user sessions for shopping cart

```
app.set('view engine', 'ejs');
```

What it does: Sets EJS as template engine for rendering HTML

```
app.use(express.static('public'));
```

What it does: Serves static files (CSS, JS, images) from public/ folder

---

4. Database Connection

```
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'NewPassword123!',
    database: 'bookstore'
});

connection.connect((error) => {
    if (error) {
        console.error('Database connection failed');
        process.exit(1);
    }
    console.log('Connected to MySQL');
});
```

What it does:

- Creates connection to MySQL
- Verifies connection works
- Exits application if connection fails

---

5. Routes

Homepage Route:

```
app.get('/', (req, res) => {
    res.render('index');
});
```

What it does: Renders homepage when user visits http://ip/

Books Listing Route:

```
app.get('/books', (req, res) => {
    const sql = 'SELECT * FROM books';

    connection.query(sql, (error, results) => {
        if (error) {
            return res.status(500).json({ error: 'Database error' });
        }
        res.render('books', { books: results });
    });
});
```

What it does:

1. Receives GET request to /books
2. Queries database for all books
3. Renders books.ejs template with data
4. Returns HTML to browser

Add to Cart Route:

```
app.post('/api/cart/add', (req, res) => {
    const { book_id, quantity } = req.body;

    if (!req.session.cart) {
        req.session.cart = [];
    }

    req.session.cart.push({ book_id, quantity });

    res.json({ success: true });
});
```

What it does:

1. Receives POST request with book_id and quantity
2. Initializes cart in session if doesn't exist
3. Adds item to cart
4. Returns success response

Checkout Route:

```
app.post('/api/checkout', (req, res) => {
    // 1. Get customer info from request
    const { customer_name, customer_email } = req.body;

    // 2. Calculate total from cart
    // 3. Insert order into orders table
    // 4. Insert items into order_items table
    // 5. Clear cart
    // 6. Return success response
});
```

What it does:

1. Receives checkout request
2. Creates order in database
3. Saves order items
4. Clears shopping cart
5. Returns order confirmation

---

6. Start Server

```
app.listen(8080, () => {
    console.log('Server running on port 8080');
});
```

What it does:

- Starts HTTP server on port 8080
- Listens for incoming requests
- Logs confirmation message

Database Schema Files

BuyTheBook_Schema.sql

```
 -- Create authors table
CREATE TABLE authors (
    author_id INT PRIMARY KEY AUTO_INCREMENT,
    author_name VARCHAR(100) NOT NULL,
    bio TEXT
);

-- Create books table
CREATE TABLE books (
    book_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(200) NOT NULL,
    author_id INT,
    price DECIMAL(10,2) NOT NULL,
    isbn VARCHAR(13) UNIQUE,
    description TEXT,
    stock_quantity INT DEFAULT 0,

    FOREIGN KEY (author_id) REFERENCES authors(author_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);

-- Create orders table
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_name VARCHAR(100) NOT NULL,
    customer_email VARCHAR(100) NOT NULL,
    total_amount DECIMAL(10,2) NOT NULL,
    order_status ENUM('pending', 'processing', 'shipped', 'delivered') DEFAULT 'pending',
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create order_items table
CREATE TABLE order_items (
    item_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT NOT NULL,
    book_id INT NOT NULL,
    quantity INT NOT NULL,
    price DECIMAL(10,2) NOT NULL,

    FOREIGN KEY (order_id) REFERENCES orders(order_id)
        ON DELETE CASCADE,
    FOREIGN KEY (book_id) REFERENCES books(book_id)
        ON DELETE RESTRICT
);
```

What it does:

- Creates 4 tables with proper structure
- Defines relationships using foreign keys
- Sets constraints for data integrity

---

author_seed.sql

```
 INSERT INTO authors (author_name, bio) VALUES
('J.K. Rowling', 'British author of Harry Potter'),
('George R.R. Martin', 'American novelist'),
('Stephen King', 'Master of horror fiction');
```

What it does: Inserts sample authors into database

---

books_seed.sql

```
INSERT INTO books (title, author_id, price, isbn, description, stock_quantity) VALUES
('Harry Potter Complete Collection', 1, 99.99, '9781408856772', 'All seven books', 50),
('Game of Thrones', 2, 34.99, '9780553103540', 'First novel in series', 80),
('The Shining', 3, 19.99, '9780307743657', 'Horror classic', 100);
```

What it does: Inserts sample books into database

EJS Template Example

books.ejs

```
<!DOCTYPE html>
<html>
<head>
    <title>Books - The EpicBook</title>
    <link rel="stylesheet" href="/css/styles.css">
</head>
<body>
    <header>
        <h1>The EpicBook!</h1>
        <nav>
            <a href="/">Home</a>
            <a href="/books">Books</a>
            <a href="/cart">Cart (<span id="cart-count">0</span>)</a>
        </nav>
    </header>

    <main>
        <h2>Our Book Collection</h2>

        <div class="books-grid">
            <% books.forEach(book => { %>
                <div class="book-card">
                    <h3><%= book.title %></h3>
                    <p class="author">by <%= book.author_name %></p>
                    <p class="price">$<%= book.price %></p>
                    <p class="stock">
                        <%= book.stock_quantity > 0 ? 'In Stock' : 'Out of Stock' %>
                    </p>
                    <button onclick="addToCart(<%= book.book_id %>)">
                        Add to Cart
                    </button>
                </div>
            <% }); %>
        </div>
    </main>

    <script src="/js/main.js"></script>
</body>
</html>
```

What it does:

- Loops through books array
- Displays each book in a card
- Shows title, price, stock status
- Provides "Add to Cart" button

Nginx Configuration

/etc/nginx/conf.d/theepicbooks.conf

```
server {
    listen 80;
    server_name 54.123.45.67;

    location / {
        proxy_pass http://localhost:8080;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_cache_bypass $http_upgrade;
    }
}
```

What each line does:

Line	Purpose
`listen 80;`	Listen on port 80 (HTTP)
`server_name 54.123.45.67;`	Respond to this IP address
`proxy_pass http://localhost:8080;`	Forward requests to Node.js
`proxy_set_header Host $host;`	Pass original host to Node.js
`proxy_set_header X-Real-IP $remote_addr;`	Pass client's IP to Node.js

---

6. Complete Request-Response Flow

Scenario: User Browses Books

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 1: User Action                                         │
└─────────────────────────────────────────────────────────────┘

User opens browser
User types: http://54.123.45.67/books
User presses Enter

Browser creates HTTP GET request:
GET /books HTTP/1.1
Host: 54.123.45.67
User-Agent: Mozilla/5.0 (Chrome)
Accept: text/html


┌─────────────────────────────────────────────────────────────┐
│ STEP 2: Request Travels Over Internet                       │
└─────────────────────────────────────────────────────────────┘

User's Computer (IP: 203.0.113.50)
        │
        │ HTTP Request
        │ Destination: 54.123.45.67:80
        │
        ▼
Internet Routers
        │
```

```
        ▼
AWS Region (us-east-1)
        │
        ▼
EC2 Instance (Public IP: 54.123.45.67)


┌─────────────────────────────────────────────────────────────┐
│ STEP 3: Security Group Check                                │
└─────────────────────────────────────────────────────────────┘

AWS Security Group:
- Check: Is port 80 allowed? → YES ✓
- Check: Is source IP blocked? → NO ✓
- Action: Allow request through


┌─────────────────────────────────────────────────────────────┐
│ STEP 4: Nginx Receives Request (Port 80)                    │
└─────────────────────────────────────────────────────────────┘

Nginx Web Server (listening on port 80)
        │
        ├─ Receives: GET /books HTTP/1.1
        ├─ Logs request to: /var/log/nginx/access.log
        │   203.0.113.50 - - [02/Feb/2026:10:30:00] "GET /books HTTP/1.1"
        │
        ├─ Checks configuration: /etc/nginx/conf.d/theepicbooks.conf
        │   location / {
        │       proxy_pass http://localhost:8080;
        │   }
        │
        └─ Decision: Forward to Node.js on port 8080


┌─────────────────────────────────────────────────────────────┐
│ STEP 5: Nginx Forwards to Node.js (Port 8080)               │
└─────────────────────────────────────────────────────────────┘

Nginx → Node.js (localhost:8080)

Modified request (Nginx adds headers):
GET /books HTTP/1.1
Host: 54.123.45.67
X-Real-IP: 203.0.113.50              ← Added by Nginx
X-Forwarded-For: 203.0.113.50        ← Added by Nginx
X-Forwarded-Proto: http              ← Added by Nginx
Connection: upgrade


┌─────────────────────────────────────────────────────────────┐
│ STEP 6: Node.js Receives Request                            │
└─────────────────────────────────────────────────────────────┘

Node.js (Express) Application

server.js:
app.get('/books', (req, res) => {
    // This function executes
});

What Node.js does:
1. Express router matches: GET /books
2. Finds corresponding route handler
3. Executes the callback function
```

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 7: Node.js Queries MySQL Database                      │
└─────────────────────────────────────────────────────────────┘
```

Node.js creates SQL query:

```
const sql = 'SELECT * FROM books';

connection.query(sql, (error, results) => {
    // This will execute after MySQL responds
});
```

```
Node.js → MySQL (localhost:3306)
        │
        ├─ Opens TCP connection to port 3306
        ├─ Sends SQL query: SELECT * FROM books
        └─ Waits for response (asynchronous)
```

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 8: MySQL Processes Query                               │
└─────────────────────────────────────────────────────────────┘
```

MySQL Database Server (Port 3306)

Receives: SELECT * FROM books

Processing steps:
1. Parse SQL query
2. Check user permissions (does 'root' have SELECT privilege?)
3. Access database: bookstore
4. Access table: books
5. Read data from disk: /var/lib/mysql/bookstore/books.ibd
6. Apply any WHERE filters (none in this case)
7. Load all rows into memory
8. Return results

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 9: MySQL Returns Data to Node.js                       │
└─────────────────────────────────────────────────────────────┘
```

MySQL → Node.js

Returns data in this format (JSON-like):

```
[
    {
        book_id: 1,
        title: 'Harry Potter',
        author_id: 1,
        price: 99.99,
        isbn: '9781408856772',
        description: 'Complete collection...',
        stock_quantity: 50
    },
    {
        book_id: 2,
        title: 'Game of Thrones',
        author_id: 2,
        price: 34.99,
        isbn: '9780553103540',
        description: 'First novel in...',
        stock_quantity: 80
    },
```

```
    // ... more books
]

Connection closes (or returns to pool)



┌─────────────────────────────────────────────────────────────┐
│ STEP 10: Node.js Processes Data                             │
└─────────────────────────────────────────────────────────────┘


Node.js callback function executes:

connection.query(sql, (error, results) => {
    if (error) {
        console.error('Database error:', error);
        return res.status(500).json({ error: 'Database error' });
    }

    // results = array of book objects from MySQL

    // Option A: Send JSON (for API)
    res.json(results);

    // Option B: Render HTML (for web page)
    res.render('books', { books: results });
});

What Node.js does here:
1. Receives results from MySQL
2. Checks for errors
3. Processes/formats data if needed
4. Generates response (HTML or JSON)



┌─────────────────────────────────────────────────────────────┐
│ STEP 11: Node.js Generates HTML Response                    │
└─────────────────────────────────────────────────────────────┘


If using template engine (EJS):

res.render('books', { books: results });

EJS template (books.ejs):
<!DOCTYPE html>
<html>
<head>
    <title>Books - The EpicBook</title>
</head>
<body>
    <h1>Our Books</h1>
    <div class="books-grid">
        <% books.forEach(book => { %>
            <div class="book-card">
                <h3><%= book.title %></h3>
                <p>Price: $<%= book.price %></p>
                <button>Add to Cart</button>
            </div>
        <% }); %>
    </div>
</body>
</html>

Node.js generates final HTML:
<!DOCTYPE html>
<html>
<head>
```

```
<head>
    <title>Books - The EpicBook</title>
</head>
<body>
    <h1>Our Books</h1>
    <div class="books-grid">
        <div class="book-card">
            <h3>Harry Potter</h3>
            <p>Price: $99.99</p>
            <button>Add to Cart</button>
        </div>
        <div class="book-card">
            <h3>Game of Thrones</h3>
            <p>Price: $34.99</p>
            <button>Add to Cart</button>
        </div>
        <!-- More books... -->
    </div>
</body>
</html>
```

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 12: Node.js Sends Response to Nginx                    │
└─────────────────────────────────────────────────────────────┘

Node.js → Nginx

HTTP Response:
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 5432
Date: Sun, 02 Feb 2026 10:30:01 GMT

<!DOCTYPE html>
<html>
...
</html>
```

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 13: Nginx Forwards Response to Browser                 │
└─────────────────────────────────────────────────────────────┘

Nginx receives response from Node.js

Nginx may add headers:
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 5432
Server: nginx/1.20.1              ← Added by Nginx
X-Powered-By: Express             ← From Node.js
Date: Sun, 02 Feb 2026 10:30:01 GMT

<!DOCTYPE html>
<html>
...
</html>

Nginx → User's Browser
```

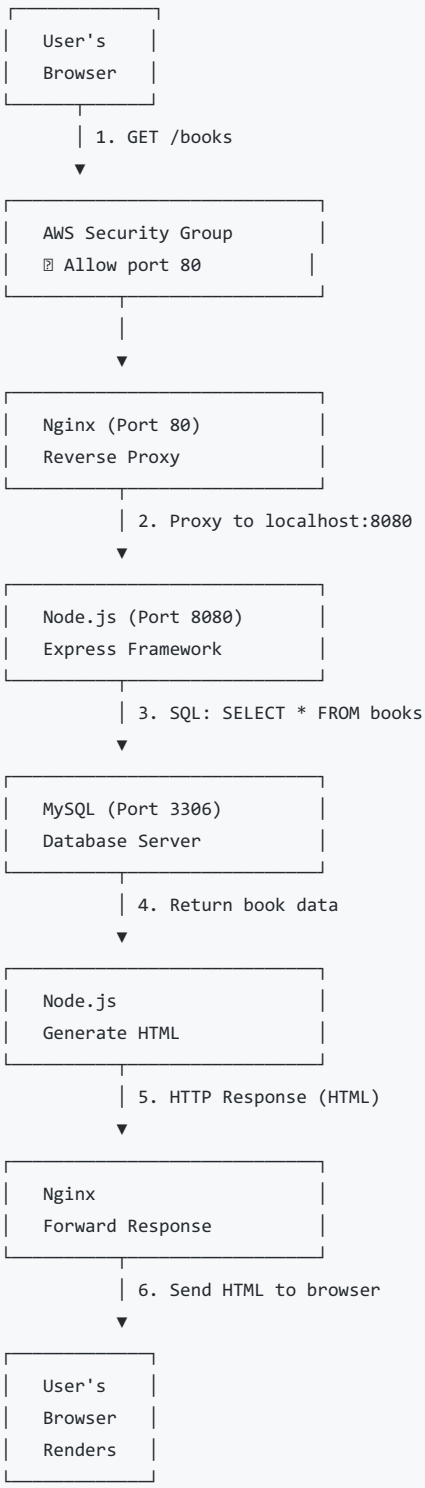
```
┌─────────────────────────────────────────────────────────────┐
│ STEP 14: Browser Receives and Renders Page                  │
└─────────────────────────────────────────────────────────────┘
```

```
User's Browser receives HTML

Browser processing:
1. Parses HTML structure
2. Builds DOM (Document Object Model)
3. Requests additional resources:
   - CSS files: GET /css/styles.css
   - JavaScript files: GET /js/main.js
   - Images: GET /images/logo.png
4. Applies CSS styling
5. Executes JavaScript
6. Renders final page

User sees: Beautiful bookstore page with books displayed! 
```

Simplified Visual Flow



Summary

Technologies Used:

- **Node.js:** Backend runtime
- **Express.js:** Web framework
- **MySQL:** Database
- **Nginx:** Reverse proxy
- **AWS EC2:** Cloud server
- **Git:** Version control

Key Concepts:

1. **Monolithic Architecture:** All components on one server
2. **Reverse Proxy:** Nginx forwards requests to Node.js
3. **MVC Pattern:** Models (data), Views (templates), Controllers (routes)

4. **Session Management:** Shopping cart stored in memory
5. **Database Relationships:** Foreign keys link tables

Request Flow:

```
User → Nginx (80) → Node.js (8080) → MySQL (3306)
                                          ↓
User ← Nginx (80) ← Node.js (8080) ← MySQL data
```

✅ Project Complete!

This documentation covers all essential aspects of The EpicBook project, from technology choices to complete implementation details and request-response flow.