Task 1:

```
protoc --go_out=. --go_opt=paths=source_relative book.proto
```

- protoc: Protocol Buffers compiler.
- --go_out=.: Generates Go structs/messages for the proto definitions in the current directory.

```
import (
    "fmt"
    "log"

    // Import the generated protobuf code
    pb "book-catalog-grpc/proto"
    "google.golang.org/protobuf/proto"
)
```

- pb "book-catalog-grpc/proto": Imports generated protobuf code with alias pb (protocol buffer)

```go
func main() {
    // Create a Book instance
    book := &pb.Book{
        Id:            1,
        Title:         "The Go Programming Language",
        Author:        "Alan Donovan",
        Isbn:          "978-0134190440",
        Price:         39.99,
        Stock:         15,
        PublishedYear: 2015,
    }

    fmt.Printf("Book: %v\n", book)

    // Create DetailedBook with category and tags
    detailedBook := &pb.DetailedBook{
        Book:        book,
        Category:    pb.BookCategory_NONFICTION,
        Description: "A comprehensive introduction to Go programming.",
        Tags:        []string{"programming", "go", "technical"},
        Rating:      4.5,
    }

    fmt.Printf("\nDetailed Book: %v\n", detailedBook)
    fmt.Printf("Category: %s\n", detailedBook.Category)
    fmt.Printf("Tags: %v\n", detailedBook.Tags)
```

- Initializes a Book struct with fields like Id, Title, Author, ISBN, Price, Stock, and PublishedYear, then prints the struct using %v.
- Composes a DetailedBook containing a nested Book.
- Adds Category (enum), Description, Tags (string slice), and Rating.
- Prints the full struct, category, and tags.

```go
// Serialize to bytes
data, err := proto.Marshal(book)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("\nSerialized size: %d bytes\n", len(data))
```

- proto.Marshal converts Book to binary bytes; prints serialized size.

```
// Deserialize from bytes
newBook := &pb.Book{}
err = proto.Unmarshal(data, newBook)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("Deserialized book: %v\n", newBook)
```

- proto.Unmarshal reconstructs a new Book from bytes; prints deserialized struct.

```go
    // Create Author with multiple books
    author := &pb.Author{
        Id:    1,
        Name:  "Robert C. Martin",
        Bio:   "A renowned author in software development.",
        BirthYear: 1952,
        Books: []*pb.Book{
            {
                Id:            1,
                Title:         "Clean Code",
                Author:        "Robert C. Martin",
                Isbn:          "978-0132350884",
                Price:         29.99,
                Stock:         50,
                PublishedYear: 2008,
            },
            {
                Id:            2,
                Title:         "Clean Architecture",
                Author:        "Robert C. Martin",
                Isbn:          "978-0134494166",
                Price:         34.99,
                Stock:         30,
                PublishedYear: 2017,
            },
        },
    }

    fmt.Printf("\nAuthor: %s\n", author.Name)
    fmt.Printf("Books written: %d\n", len(author.Books))
    for i, b := range author.Books {
        fmt.Printf("  %d. %s\n", i+1, b.Title)
    }
}
```

- Initializes an Author struct with Id, Name, Bio, BirthYear.
- Assigns a slice of Book pointers to demonstrate a one-to-many relationship.
- Prints author name, number of books, and a numbered list of book titles.

Output:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab6\Task1> go run .\main.go
Book: id:1  title:"The Go Programming Language"  author:"Alan Donovan"  isbn:"978-0134190440"  price:39.99  stock:15  published_year:2015

Detailed Book: book:{id:1  title:"The Go Programming Language"  author:"Alan Donovan"  isbn:"978-0134190440"  price:39.99  stock:15  publis
hed_year:2015}  category:NONFICTION  description:"A comprehensive introduction to Go programming."  tags:"programming"  tags:"go"  tags:"te
chnical"  rating:4.5
Category: NONFICTION
Tags: [programming go technical]

Serialized size: 71 bytes
Deserialized book: id:1  title:"The Go Programming Language"  author:"Alan Donovan"  isbn:"978-0134190440"  price:39.99  stock:15  publishe
d_year:2015

Author: Robert C. Martin
Books written: 2
  1. Clean Code
  2. Clean Architecture
```

Task 2:

```
protoc --go_out=. --go-grpc_out=. calculator.proto
```

- protoc: Protocol Buffers compiler.
- --go_out=.: Generates Go structs/messages for the proto definitions in the current directory.
- --go-grpc_out=.: Generates Go gRPC service code (interfaces and stubs) in the current directory.

Server:

```
type calculatorServer struct {
    pb.UnimplementedCalculatorServer
    history []string
}
```

- Defines the server struct that implements the Calculator service
- pb.UnimplementedCalculatorServer: Embedding this ensures forward compatibility (if new methods are added to the proto, the code won't break)
- history []string: A slice to store calculation history in memory

```go
func (s *calculatorServer) Calculate(ctx context.Context, req *pb.CalculateRequest) (*pb.CalculateResponse, error) {
    log.Printf("Calculate: %.2f %s %.2f", req.A, req.Operation, req.B)

    var result float32

    switch req.Operation {
    case "add":
        result = req.A + req.B
    case "subtract":
        result = req.A - req.B
    case "multiply":
        result = req.A * req.B
    case "divide":
        if req.B == 0 {
            return nil, status.Errorf(codes.InvalidArgument, "cannot divide by zero")
        }
        result = req.A / req.B
    default:
        return nil, status.Errorf(codes.InvalidArgument, "unknown operation: %s", req.Operation)
    }

    entry := fmt.Sprintf("%.2f %s %.2f = %.2f", req.A, req.Operation, req.B, result)
    s.history = append(s.history, entry)

    return &pb.CalculateResponse{
        Result:    result,
        Operation: req.Operation,
    }, nil
}
```

- Takes context.Context (required for all gRPC methods) and a pointer to the request; returns a pointer to the response and an error
- Logs the incoming request with 2 decimal places formatting
- Switch statement handles different operations based on the operation field
- Creates a formatted string of the calculation then appends this entry to the server's history slice
- Returns the response with the calculation result and operation

```go
func (s *calculatorServer) SquareRoot(ctx context.Context, req *pb.SquareRootRequest) (*pb.SquareRootResponse, error) {
    log.Printf("SquareRoot: %.2f", req.Number)

    if req.Number < 0 {
        return nil, status.Errorf(codes.InvalidArgument,
            "cannot calculate square root of negative number: %.2f", req.Number)
    }

    result := float32(math.Sqrt(float64(req.Number)))

    entry := fmt.Sprintf("sqrt(%.2f) = %.2f", req.Number, result)
    s.history = append(s.history, entry)

    return &pb.SquareRootResponse{
        Result: result,
    }, nil
}
```

- Logs the incoming square root request
- Validates input: negative numbers don't have real square roots
- math.Sqrt() requires float64, so the number is cast to float64 for the calculation and the result is converted back to float32.

- Creates history entry and appends to history
- Returns the result wrapped in the response message

```go
func (s *calculatorServer) GetHistory(ctx context.Context, req *pb.HistoryRequest) (*pb.HistoryResponse, error) {
    log.Println("GetHistory called")
    return &pb.HistoryResponse{
        Calculations: s.history,
        Count:        int32(len(s.history)),
    }, nil
}
```

- Simple logging when history is requested and returning all stored calculations and the count.

```go
func main() {
    lis, err := net.Listen("tcp", ":50051")
    if err != nil {
        log.Fatalf("Failed to listen: %v", err)
    }

    grpcServer := grpc.NewServer()
    pb.RegisterCalculatorServer(grpcServer, &calculatorServer{})

    log.Println("🚀 Calculator gRPC server listening on :50051")

    if err := grpcServer.Serve(lis); err != nil {
        log.Fatalf("Failed to serve: %v", err)
    }
}
```

- Starts the gRPC server by creating a TCP listener on port 50051
- Initializing a new gRPC server, registering the calculatorServer implementation
- Logging that the server is running, and finally calling Serve to begin handling incoming gRPC requests.

Client:

```go
func main() {
    conn, err := grpc.Dial(
        "localhost:50051",
        grpc.WithTransportCredentials(insecure.NewCredentials()),
    )
    if err != nil {
        log.Fatalf("Failed to connect: %v", err)
    }
    defer conn.Close()

    client := pb.NewCalculatorClient(conn)

    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
```

Establish Connection

- grpc.Dial("localhost:50051", grpc.WithTransportCredentials(insecure.NewCredentials())): connects to the server over an insecure channel.
- Exits with fatal error if connection fails.
- defer conn.Close(): ensures the connection closes when the program ends.

Create Client & Context

- pb.NewCalculatorClient(conn): generates a Calculator service client stub.
- context.WithTimeout(..., 5*time.Second): creates a context with a 5-second RPC timeout.
- defer cancel(): cleans up the context when finished.

```go
// Test 1: Addition
fmt.Println("=== Test 1: Addition ===")
resp, err := client.Calculate(ctx, &pb.CalculateRequest{
    A:         10,
    B:         5,
    Operation: "add",
})
if err == nil {
    fmt.Printf("Result: %.2f + %.2f = %.2f\n", 10.0, 5.0, resp.Result)
}
```

- Prints header, sends Calculate request {A:10, B:5, Operation:"add"}, and prints the formatted result if no error.

```
// Test 2: Division
fmt.Println("\n=== Test 2: Division ===")
resp, err = client.Calculate(ctx, &pb.CalculateRequest{
    A:         20,
    B:         4,
    Operation: "divide",
})
if err == nil {
    fmt.Printf("Result: %.2f / %.2f = %.2f\n", 20.0, 4.0, resp.Result)
}
```

- Runs a valid division test (20 / 4), prints result, and uses the same context as previous calls.

```
// Test 3: Division by Zero
fmt.Println("\n=== Test 3: Division by Zero ===")
_, err = client.Calculate(ctx, &pb.CalculateRequest{
    A:         10,
    B:         0,
    Operation: "divide",
})
if err != nil {
    st, _ := status.FromError(err)
    fmt.Printf("Expected error: %s\n", st.Message())
}
```

- Sends a divide-by-zero request, expects an error, converts it to a gRPC status, and prints the server-returned message.

```
// Test 4: Square Root
fmt.Println("\n=== Test 4: Square Root ===")
sqrtResp, err := client.SquareRoot(ctx, &pb.SquareRootRequest{Number: 16})
if err == nil {
    fmt.Printf("Result: sqrt(16.00) = %.2f\n", sqrtResp.Result)
}
```

- Calls SquareRoot with 16, prints the result (expected 4.00) when successful.

```go
// Test 5: Negative sqrt
fmt.Println("\n=== Test 5: Negative Square Root ===")
_, err = client.SquareRoot(ctx, &pb.SquareRootRequest{Number: -4})
if err != nil {
    st, _ := status.FromError(err)
    fmt.Printf("Expected error: %s\n", st.Message())
}
```

- Sends a negative number to SquareRoot, receives a gRPC error, extracts and prints the error message.

```go
// Test 6: Get history
fmt.Println("\n=== Test 6: History ===")
hist, _ := client.GetHistory(ctx, &pb.HistoryRequest{})
fmt.Printf("Calculations: %d\n", hist.Count)
for i, h := range hist.Calculations {
    fmt.Printf("%d. %s\n", i+1, h)
}
```

- Calls GetHistory, prints the total count, and iterates through all stored calculation entries for display.

Output:

Server:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab6\Task2\server> go run .\main.go
2025/11/26 19:35:24 🚀 Calculator gRPC server listening on :50051
2025/11/26 19:35:39 Calculate: 10.00 add 5.00
2025/11/26 19:35:39 Calculate: 20.00 divide 4.00
2025/11/26 19:35:39 Calculate: 10.00 divide 0.00
2025/11/26 19:35:39 SquareRoot: 16.00
2025/11/26 19:35:39 SquareRoot: -4.00
2025/11/26 19:35:39 GetHistory called
```

Client:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU2126
7 - Lab6\Task2\client>                          go run .\main.go
=== Test 1: Addition ===
Result: 10.00 + 5.00 = 15.00

=== Test 2: Division ===
Result: 20.00 / 4.00 = 5.00

=== Test 3: Division by Zero ===
Expected error: cannot divide by zero

=== Test 4: Square Root ===
Result: sqrt(16.00) = 4.00

=== Test 5: Negative Square Root ===
Expected error: cannot calculate square root of negative number: -4
.00

=== Test 6: History ===
Calculations: 3
1. 10.00 add 5.00 = 15.00
2. 20.00 divide 4.00 = 5.00
3. sqrt(16.00) = 4.00
```

Task 3:

```
protoc --go_out=. --go-grpc_out=. book_service.proto
```

- protoc: Protocol Buffers compiler.
- --go_out=.: Generates Go structs/messages for the proto definitions in the current directory.
- --go-grpc_out=.: Generates Go gRPC service code (interfaces and stubs) in the current directory.

Server:

```go
type bookCatalogServer struct {
    pb.UnimplementedBookCatalogServer
    db *sql.DB
}
```

bookCatalogServer defines the gRPC server for the BookCatalog service.

- Embeds pb.UnimplementedBookCatalogServer for forward compatibility.
- Holds a *sql.DB connection shared by all RPC methods, enabling database CRUD operations.

```go
// ======================== GetBook ============================

func (s *bookCatalogServer) GetBook(ctx context.Context, req *pb.GetBookRequest) (*pb.GetBookResponse, error) {
    row := s.db.QueryRowContext(ctx,
        "SELECT id, title, author, isbn, price, stock, published_year FROM books WHERE id = ?",
        req.Id,
    )

    var book pb.Book
    err := row.Scan(&book.Id, &book.Title, &book.Author, &book.Isbn,
        &book.Price, &book.Stock, &book.PublishedYear)

    if err == sql.ErrNoRows {
        return nil, status.Errorf(codes.NotFound, "book not found: id=%d", req.Id)
    }
    if err != nil {
        return nil, err
    }

    return &pb.GetBookResponse{Book: &book}, nil
}
```

GetBook retrieves a single book by its ID.

- Executes a parameterized SQL query using QueryRowContext to safely fetch one row (ctx enables cancellation/timeout; ? prevents SQL injection).
- Scans the result into a pb.Book struct, matching the SELECT column order.
- If no row exists, returns a gRPC NotFound error; any other scan/query error is returned as-is.
- On success, wraps the populated Book in GetBookResponse and returns it.

```go
// ======================== CreateBook ============================

func (s *bookCatalogServer) CreateBook(ctx context.Context, req *pb.CreateBookRequest) (*pb.CreateBookResponse, error) {
    res, err := s.db.ExecContext(ctx,
        "INSERT INTO books (title, author, isbn, price, stock, published_year) VALUES (?, ?, ?, ?, ?, ?)",
        req.Title, req.Author, req.Isbn, req.Price, req.Stock, req.PublishedYear)

    if err != nil {
        return nil, err
    }

    id, _ := res.LastInsertId()

    return &pb.CreateBookResponse{
        Book: &pb.Book{
            Id:            int32(id),
            Title:         req.Title,
            Author:        req.Author,
            Isbn:          req.Isbn,
            Price:         req.Price,
            Stock:         req.Stock,
            PublishedYear: req.PublishedYear,
        },
    }, nil
}
```

CreateBook inserts a new book into the database.

- Uses ExecContext to run an INSERT statement (no returned rows).
- Fills the six ? placeholders with the request fields; id is omitted because it's auto-generated.
- Returns an error immediately if the insert fails.
- Retrieves the generated ID using LastInsertId and converts it to int32.
- Builds and returns a CreateBookResponse containing the newly created Book, including its assigned ID.

```go
// ======================== UpdateBook ============================

func (s *bookCatalogServer) UpdateBook(ctx context.Context, req *pb.UpdateBookRequest) (*pb.UpdateBookResponse, error) {
    res, err := s.db.ExecContext(ctx,
        `UPDATE books SET title=?, author=?, isbn=?, price=?, stock=?, published_year=? WHERE id=?`,
        req.Title, req.Author, req.Isbn, req.Price, req.Stock, req.PublishedYear, req.Id)

    if err != nil {
        return nil, err
    }

    rows, _ := res.RowsAffected()
    if rows == 0 {
        return nil, status.Errorf(codes.NotFound, "book not found: id=%d", req.Id)
    }

    return &pb.UpdateBookResponse{
        Book: &pb.Book{
            Id:            req.Id,
            Title:         req.Title,
            Author:        req.Author,
            Isbn:          req.Isbn,
            Price:         req.Price,
            Stock:         req.Stock,
            PublishedYear: req.PublishedYear,
        },
    }, nil
}
```

UpdateBook modifies an existing book by ID.

- Executes a parameterized UPDATE statement with seven placeholders (six fields + id).
- Returns any database errors immediately.
- Checks RowsAffected(); if zero, returns gRPC NotFound (book doesn't exist).
- On success, returns UpdateBookResponse containing the updated book, echoing the request values.

```go
// ======================== DeleteBook ============================

func (s *bookCatalogServer) DeleteBook(ctx context.Context, req *pb.DeleteBookRequest) (*pb.DeleteBookResponse, error) {
    res, err := s.db.ExecContext(ctx, "DELETE FROM books WHERE id=?", req.Id)
    if err != nil {
        return nil, err
    }

    rows, _ := res.RowsAffected()
    if rows == 0 {
        return &pb.DeleteBookResponse{
            Success: false,
            Message: "book not found",
        }, nil
    }

    return &pb.DeleteBookResponse{
        Success: true,
        Message: "book deleted successfully",
    }, nil
}
```

DeleteBook removes a book by ID.

- Executes a parameterized DELETE query and handles any database errors.
- Checks RowsAffected(); if zero, returns a response with Success: false and a "book not found" message (not an error).
- If a row was deleted, returns Success: true with a confirmation message.

```go
// ====================== ListBooks (Pagination) ==========================

func (s *bookCatalogServer) ListBooks(ctx context.Context, req *pb.ListBooksRequest) (*pb.ListBooksResponse, error) {
    if req.Page <= 0 {
        req.Page = 1
    }
    if req.PageSize <= 0 {
        req.PageSize = 5
    }

    offset := (req.Page - 1) * req.PageSize

    // Total count
    var total int32
    s.db.QueryRowContext(ctx, "SELECT COUNT(*) FROM books").Scan(&total)

    // Query with pagination
    rows, err := s.db.QueryContext(ctx,
        "SELECT id, title, author, isbn, price, stock, published_year FROM books LIMIT ? OFFSET ?",
        req.PageSize, offset)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    books := []*pb.Book{}
    for rows.Next() {
        var b pb.Book
        rows.Scan(&b.Id, &b.Title, &b.Author, &b.Isbn, &b.Price, &b.Stock, &b.PublishedYear)
        books = append(books, &b)
    }

    return &pb.ListBooksResponse{
        Books:    books,
        Total:    total,
        Page:     req.Page,
        PageSize: req.PageSize,
    }, nil
}
```

ListBooks returns a paginated list of books.

- Validates Page and PageSize, defaulting to 1 and 5 if invalid.
- Calculates offset = (Page-1) * PageSize to skip rows.
- Queries total book count for pagination info.
- Executes a parameterized SELECT with LIMIT and OFFSET to fetch the current page.

- Iterates over rows, scanning each book into a pb.Book and appending to a slice.
- Returns ListBooksResponse containing the current page's books, total count, page number, and page size.

```go
// ==================== DB Initialization ======================

func initDB() (*sql.DB, error) {
    db, err := sql.Open("sqlite3", "./books.db")
    if err != nil {
        return nil, err
    }

    _, err = db.Exec(`
        CREATE TABLE IF NOT EXISTS books (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT,
            author TEXT,
            isbn TEXT,
            price REAL,
            stock INTEGER,
            published_year INTEGER
        );
    `)
    if err != nil {
        return nil, err
    }

    // Seed only when empty
    var count int
    db.QueryRow("SELECT COUNT(*) FROM books").Scan(&count)
    if count == 0 {
        fmt.Println("Seeding sample books...")
        db.Exec(`
            INSERT INTO books (title, author, isbn, price, stock, published_year) VALUES
            ('The Go Programming Language','Alan Donovan','9780134190440',39.99,10,2015),
            ('Clean Code','Robert Martin','9780132350884',42.50,15,2008),
            ('Design Patterns','Erich Gamma','9780201633610',55.00,7,1994),
            ('Concurrency in Go','Katherine Cox','9781491941195',33.99,12,2017),
            ('Deep Work','Cal Newport','9781455586691',29.99,20,2016);
        `)
    }

    return db, nil
}
```

initDB initializes the SQLite database and returns a *sql.DB connection.

- Opens the SQLite database file (books.db), creating it if needed.

- Creates the books table if it doesn't exist, defining id as auto-increment primary key and other book fields.
- Checks if the table is empty; if so, seeds it with five sample books.
- Returns the database connection for use by the server.

```go
// ============================ main ============================

func main() {
    db, err := initDB()
    if err != nil {
        log.Fatal("DB init error:", err)
    }

    lis, err := net.Listen("tcp", ":50052")
    if err != nil {
        log.Fatal(err)
    }

    s := grpc.NewServer()
    pb.RegisterBookCatalogServer(s, &bookCatalogServer{db: db})

    fmt.Println("📚 Book Catalog gRPC server running on :50052")
    if err := s.Serve(lis); err != nil {
        log.Fatal(err)
    }
}
```

main starts the Book Catalog gRPC server.

- Initializes the database using initDB, logging a fatal error if it fails.
- Creates a TCP listener on port 50052 for incoming gRPC connections.
- Instantiates a new gRPC server and registers bookCatalogServer with the database connection.
- Prints a startup message and begins serving, blocking indefinitely until the server stops or an error occurs.

Client:

```
func main() {
    conn, err := grpc.Dial("localhost:50052",
        grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    client := pb.NewBookCatalogClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
```

- Connects to localhost:50052 using an insecure channel.
- Creates a BookCatalog client stub with a 5-second timeout context.

```
// --- List Books ---
fmt.Println("=== Test 1: List All Books ===")
list, _ := client.ListBooks(ctx, &pb.ListBooksRequest{Page: 1, PageSize: 10})
fmt.Println("Total books:", list.Total)
for i, b := range list.Books {
    fmt.Printf("%d. %s by %s - $%.2f\n", i+1, b.Title, b.Author, b.Price)
}
```

- Calls ListBooks (page 1, 10 items) and prints total books and a numbered list of title, author, and price.

```
// --- Get Book ---
fmt.Println("\n=== Test 2: Get Book ===")
book, _ := client.GetBook(ctx, &pb.GetBookRequest{Id: 1})
fmt.Printf("Book ID: %d\nTitle: %s\nAuthor: %s\nPrice: %.2f\n",
    book.Book.Id, book.Book.Title, book.Book.Author, book.Book.Price)
```

- Calls GetBook for ID=1 and prints detailed info (ID, title, author, price).

```
// --- Create Book ---
fmt.Println("\n=== Test 3: Create Book ===")
created, _ := client.CreateBook(ctx, &pb.CreateBookRequest{
    Title:         "Learning Go",
    Author:        "Jon Bodner",
    Isbn:          "9781492077213",
    Price:         31.50,
    Stock:         10,
    PublishedYear: 2021,
})
fmt.Println("Created book ID:", created.Book.Id)
```

- Calls CreateBook with book details; prints the server-generated ID.

```
// --- Update Book ---
fmt.Println("\n=== Test 4: Update Book ===")
updated, _ := client.UpdateBook(ctx, &pb.UpdateBookRequest{
    Id:            1,
    Title:         "The Go Programming Language (2nd Edition)",
    Author:        "Alan Donovan",
    Isbn:          "9780134190440",
    Price:         35.99,
    Stock:         8,
    PublishedYear: 2024,
})
fmt.Printf("Updated book: %s\nNew price: %.2f\n", updated.Book.Title, updated.Book.Price)
```

- Calls UpdateBook for ID=1 with new values; prints updated title and price.

```
// --- Delete Book ---
fmt.Println("\n=== Test 5: Delete Book ===")
del, _ := client.DeleteBook(ctx, &pb.DeleteBookRequest{Id: 6})
fmt.Println(del.Message)
```

- Calls DeleteBook for ID=6; prints the server message (deleted successfully or not found).

```
// --- Pagination ---
fmt.Println("\n=== Test 6: Pagination ===")
p1, _ := client.ListBooks(ctx, &pb.ListBooksRequest{Page: 1, PageSize: 3})
fmt.Println("Page 1:", len(p1.Books), "books")

p2, _ := client.ListBooks(ctx, &pb.ListBooksRequest{Page: 2, PageSize: 3})
fmt.Println("Page 2:", len(p2.Books), "books")
```

- Calls ListBooks with page size 3 for pages 1 and 2; prints number of books returned to demonstrate pagination.

Output:

Server:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU2126
7 - Lab6\Task3\server> go run main.go
Seeding sample books...
Book Catalog gRPC server running on :50052
```

Client:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU2126
7 - Lab6\Task3\client> go run main.go
=== Test 1: List All Books ===
Total books: 5
1. The Go Programming Language by Alan Donovan - $39.99
2. Clean Code by Robert Martin - $42.50
3. Design Patterns by Erich Gamma - $55.00
4. Concurrency in Go by Katherine Cox - $33.99
5. Deep Work by Cal Newport - $29.99

=== Test 2: Get Book ===
Book ID: 1
Title: The Go Programming Language
Author: Alan Donovan
Price: 39.99
=== Test 3: Create Book ===
Created book ID: 6

=== Test 4: Update Book ===
Updated book: The Go Programming Language (2nd Edition)
New price: 35.99

=== Test 5: Delete Book ===
book deleted successfully

=== Test 6: Pagination ===
Page 1: 3 books
Page 2: 2 books
```

Rows: 5

Filter 5 rows... Upgrade to PRO

	id	title	author	isbn	price	stock	publishe...
1	1	The Go Programming Language (2nd Edition)	Alan Donovan	9780134190440	35.9900016784668	8	2024
2	2	Clean Code	Robert Martin	9780132350884	42.5	15	2008
3	3	Design Patterns	Erich Gamma	9780201633610	55	7	1994
4	4	Concurrency in Go	Katherine Cox	9781491941195	33.99	12	2017
5	5	Deep Work	Cal Newport	9781455586691	29.99	20	2016

Task 4:

Server:

```go
// ======================== SearchBooks ============================
func (s *bookCatalogServer) SearchBooks(ctx context.Context, req *pb.SearchBooksRequest) (*pb.SearchBooksResponse, error) {
    logPrefix := "SearchBooks"
    logf := func(msg string, args ...interface{}) {
        log.Printf("%s: %s", logPrefix, fmt.Sprintf(msg, args...))
    }

    logf("query=%q, field=%q", req.Query, req.Field)

    // Validation
    if strings.TrimSpace(req.Query) == "" {
        return nil, status.Error(codes.InvalidArgument, "search query required")
    }

    field := strings.ToLower(strings.TrimSpace(req.Field))

    var sqlQuery string
    var args []interface{}
    searchPattern := "%" + req.Query + "%"

    switch field {
    case "title":
        sqlQuery = "SELECT id, title, author, isbn, price, stock, published_year FROM books WHERE title LIKE ?"
        args = []interface{}{searchPattern}
    case "author":
        sqlQuery = "SELECT id, title, author, isbn, price, stock, published_year FROM books WHERE author LIKE ?"
        args = []interface{}{searchPattern}
    case "isbn":
        // ISBN: use exact match or partial? requirement says exact match -> use '='
        sqlQuery = "SELECT id, title, author, isbn, price, stock, published_year FROM books WHERE isbn = ?"
        args = []interface{}{req.Query}
    case "all", "":
        sqlQuery = `SELECT id, title, author, isbn, price, stock, published_year
                    FROM books
                    WHERE title LIKE ? OR author LIKE ? OR isbn LIKE ?`
        args = []interface{}{searchPattern, searchPattern, searchPattern}
    default:
        return nil, status.Errorf(codes.InvalidArgument, "invalid field: %s", req.Field)
    }

    rows, err := s.db.QueryContext(ctx, sqlQuery, args...)
    if err != nil {
        return nil, status.Errorf(codes.Internal, "db query failed: %v", err)
    }
    defer rows.Close()
```

```
    books := []*pb.Book{}
    for rows.Next() {
        var b pb.Book
        if err := rows.Scan(&b.Id, &b.Title, &b.Author, &b.Isbn, &b.Price, &b.Stock, &b.PublishedYear); err != nil {
            return nil, status.Errorf(codes.Internal, "scan failed: %v", err)
        }
        books = append(books, &b)
    }

    return &pb.SearchBooksResponse{
        Books: books,
        Count: int32(len(books)),
        Query: req.Query,
    }, nil
}
```

SearchBooks – Full-Text Search

- SearchBooks allows searching books by different fields (title, author, isbn, or all).
- Starts with a custom logging function to print the query and field for easy tracking.
- Validates input: query cannot be empty, and field is normalized to lowercase for case-insensitive comparison.
- Builds a dynamic SQL query based on the field: title and author use LIKE with %, isbn uses exact match, all or empty searches all fields; invalid fields return an error.
- Executes the query, scans results into Book structs, handling scan errors properly.
- Returns SearchBooksResponse containing the matched books, count, and original query for confirmation.

```go
// ======================== FilterBooks ==========================
func (s *bookCatalogServer) FilterBooks(ctx context.Context, req *pb.FilterBooksRequest) (*pb.FilterBooksResponse, error) {
    log.Printf("FilterBooks: price[%.2f-%.2f], year[%d-%d]", req.MinPrice, req.MaxPrice, req.MinYear, req.MaxYear)

    // Validate
    if req.MinPrice < 0 || req.MaxPrice < 0 {
        return nil, status.Error(codes.InvalidArgument, "price cannot be negative")
    }
    // If both provided and min > max -> invalid
    if req.MaxPrice > 0 && req.MinPrice > req.MaxPrice {
        return nil, status.Error(codes.InvalidArgument, "min_price cannot be greater than max_price")
    }
    if req.MinYear != 0 && req.MaxYear != 0 && req.MinYear > req.MaxYear {
        return nil, status.Error(codes.InvalidArgument, "min_year cannot be greater than max_year")
    }

    query := "SELECT id, title, author, isbn, price, stock, published_year FROM books WHERE 1=1"
    var args []interface{}

    if req.MinPrice > 0 {
        query += " AND price >= ?"
        args = append(args, req.MinPrice)
    }
    if req.MaxPrice > 0 {
        query += " AND price <= ?"
        args = append(args, req.MaxPrice)
    }
    if req.MinYear != 0 {
        query += " AND published_year >= ?"
        args = append(args, req.MinYear)
    }
    if req.MaxYear != 0 {
        query += " AND published_year <= ?"
        args = append(args, req.MaxYear)
    }

    rows, err := s.db.QueryContext(ctx, query, args...)
    if err != nil {
        return nil, status.Errorf(codes.Internal, "db query failed: %v", err)
    }
    defer rows.Close()
```

```go
    books := []*pb.Book{}
    for rows.Next() {
        var b pb.Book
        if err := rows.Scan(&b.Id, &b.Title, &b.Author, &b.Isbn, &b.Price, &b.Stock, &b.PublishedYear); err != nil {
            return nil, status.Errorf(codes.Internal, "scan failed: %v", err)
        }
        books = append(books, &b)
    }

    return &pb.FilterBooksResponse{
        Books: books,
        Count: int32(len(books)),
    }, nil
}
```

FilterBooks – Range-Based Filtering

- FilterBooks filters books by price range (MinPrice–MaxPrice) and/or publication year (MinYear–MaxYear).
- Logs filter criteria for debugging.
- Validates inputs: prices and years cannot be negative, min cannot exceed max when both are provided.
- Builds a dynamic SQL query starting with WHERE 1=1 and appends conditions for price and year only if values are provided.

- Executes the query, scans results into Book structs, handling errors properly.
- Returns FilterBooksResponse containing filtered books and count.

```go
// ====================== GetStats ==========================
func (s *bookCatalogServer) GetStats(ctx context.Context, req *pb.GetStatsRequest) (*pb.GetStatsResponse, error) {
    log.Println("GetStats called")

    var totalBooks int32
    if err := s.db.QueryRowContext(ctx, "SELECT COUNT(*) FROM books").Scan(&totalBooks); err != nil {
        return nil, status.Errorf(codes.Internal, "failed to count books: %v", err)
    }

    var avgPrice sql.NullFloat64
    if err := s.db.QueryRowContext(ctx, "SELECT AVG(price) FROM books").Scan(&avgPrice); err != nil {
        return nil, status.Errorf(codes.Internal, "failed to compute average price: %v", err)
    }

    var totalStock sql.NullInt64
    if err := s.db.QueryRowContext(ctx, "SELECT SUM(stock) FROM books").Scan(&totalStock); err != nil {
        return nil, status.Errorf(codes.Internal, "failed to compute total stock: %v", err)
    }

    var earliest sql.NullInt64
    if err := s.db.QueryRowContext(ctx, "SELECT MIN(published_year) FROM books").Scan(&earliest); err != nil {
        return nil, status.Errorf(codes.Internal, "failed to get earliest year: %v", err)
    }
    var latest sql.NullInt64
    if err := s.db.QueryRowContext(ctx, "SELECT MAX(published_year) FROM books").Scan(&latest); err != nil {
        return nil, status.Errorf(codes.Internal, "failed to get latest year: %v", err)
    }
```

```go
    var earliest sql.NullInt64
    if err := s.db.QueryRowContext(ctx, "SELECT MIN(published_year) FROM books").Scan(&earliest); err != nil {
        return nil, status.Errorf(codes.Internal, "failed to get earliest year: %v", err)
    }
    var latest sql.NullInt64
    if err := s.db.QueryRowContext(ctx, "SELECT MAX(published_year) FROM books").Scan(&latest); err != nil {
        return nil, status.Errorf(codes.Internal, "failed to get latest year: %v", err)
    }

    resp := &pb.GetStatsResponse{
        TotalBooks:   totalBooks,
        AveragePrice: float32(0),
        TotalStock:   0,
        EarliestYear: 0,
        LatestYear:   0,
    }
    if avgPrice.Valid {
        resp.AveragePrice = float32(avgPrice.Float64)
    }
    if totalStock.Valid {
        resp.TotalStock = int32(totalStock.Int64)
    }
    if earliest.Valid {
        resp.EarliestYear = int32(earliest.Int64)
    }
    if latest.Valid {
        resp.LatestYear = int32(latest.Int64)
    }

    return resp, nil
}
```

GetStats – Aggregate Statistics

- GetStats computes aggregate statistics about the book collection.

- Retrieves total book count using COUNT(*).
- Computes average price and total stock using AVG(price) and SUM(stock) with sql.NullFloat64/sql.NullInt64 to handle NULLs when the table is empty.
- Finds the earliest and latest publication year with MIN and MAX, handling NULLs.
- Builds the response with safe defaults (0) and sets values only if valid, ensuring proper type conversions for protobuf.
- Returns GetStatsResponse containing total books, average price, total stock, earliest year, and latest year.

Client:

```go
func main() {
    conn, err := grpc.Dial("localhost:50052",
        grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    client := pb.NewBookCatalogClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
```

- Connects to localhost:50052 using an insecure credentials.
- Creates a BookCatalog client stub with a 5-second timeout context.
- Executes six test scenarios

Helper functions:

- searchAndCount – Searches books by field and prints count and titles.
- searchError – Searches books and prints any gRPC errors.
- filterAndCount – Filters books by price/year and prints count.
- filterError – Filters books and prints any gRPC errors.
- doStats – Retrieves and prints aggregate book statistics.
- printGrpcError – Prints gRPC errors in a readable format.

Output:

Server:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab6\Task4\server> go run main.go
 Book Catalog gRPC server running on :50052
2025/12/03 13:24:38 SearchBooks: query="go", field="title"
2025/12/03 13:24:38 SearchBooks: query="Martin", field="author"
2025/12/03 13:24:38 FilterBooks: price[20.00-45.00], year[0-0]
2025/12/03 13:24:38 FilterBooks: price[0.00-0.00], year[2010-0]
2025/12/03 13:24:38 GetStats called
2025/12/03 13:24:38 SearchBooks: query="", field="title"
2025/12/03 13:24:38 FilterBooks: price[50.00-20.00], year[0-0]
```

Client:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab6\Task4\client> go run .\main.go
=== Test 1: Search by Title ===
Searching for "go"...
Found 3 books:
- The Go Programming Language
- Concurrency in Go
- Learning Go

=== Test 2: Search by Author ===
Searching for "Martin"...
Found 1 books:
- Clean Code

=== Test 3: Filter by Price ===
Books between $20 and $45:
Found 5 books

=== Test 4: Filter by Year ===
Books published after 2010:
Found 4 books

=== Test 5: Get Statistics ===
Total books: 6
Average price: $38.83
Total stock: 74
Year range: 1994 - 2021

=== Test 6: Error Cases ===
Empty search query:
Error: search query required
Invalid price range:
Error: min_price cannot be greater than max_price
```

Task 5:

Book Service:

author_id (Foreign key linking book to author) added to SELECT and Scan

```go
// ====================== GetBooksByAuthor ========================
func (s *bookCatalogServer) GetBooksByAuthor(ctx context.Context, req *pb.GetBooksByAuthorRequest) (*pb.GetBooksByAuthorResponse, error) {
    log.Printf("GetBooksByAuthor: author_id=%d", req.AuthorId)

    rows, err := s.db.QueryContext(ctx,
        "SELECT id, title, author, isbn, price, stock, published_year, author_id FROM books WHERE author_id = ?",
        req.AuthorId,
    )
    if err != nil {
        return nil, status.Errorf(codes.Internal, "db error: %v", err)
    }
    defer rows.Close()

    books := []*pb.Book{}
    for rows.Next() {
        var b pb.Book
        if err := rows.Scan(&b.Id, &b.Title, &b.Author, &b.Isbn,
            &b.Price, &b.Stock, &b.PublishedYear, &b.AuthorId); err != nil {
            return nil, status.Errorf(codes.Internal, "scan error: %v", err)
        }
        books = append(books, &b)
    }

    return &pb.GetBooksByAuthorResponse{
        Books: books,
        Count: int32(len(books)),
    }, nil
}
```

- GetBooksByAuthor serves as an internal endpoint for other services (e.g., Author Service) to retrieve all books linked to a specific author_id.
- It logs the request, queries the database for all rows where author_id = ?, and iterates through the results, scanning each into a Book struct.
- All matching books are collected into a list, and the method returns a GetBooksByAuthorResponse containing the books and their total count.

```go
// ============================== main ==============================
func main() {
    db, err := initDB()
    if err != nil {
        log.Fatal("DB init error:", err)
    }

    lis, err := net.Listen("tcp", ":50051")
    if err != nil {
        log.Fatal(err)
    }

    s := grpc.NewServer()
    pb.RegisterBookCatalogServer(s, &bookCatalogServer{db: db})

    fmt.Println("📚 Book Catalog gRPC server running on :50051")
    if err := s.Serve(lis); err != nil {
        log.Fatal(err)
    }
}
```

- Initializes the database via initDB() and exits on failure.
- Opens a TCP listener on port :50051.

- Creates a gRPC server and registers the BookCatalog service with the database instance.

Author Service:

```
type authorCatalogServer struct {
    authorpb.UnimplementedAuthorCatalogServer
    db         *sql.DB
    bookClient bookpb.BookCatalogClient
}

func newServer(db *sql.DB, bookClient bookpb.BookCatalogClient) *authorCatalogServer {
    return &authorCatalogServer{
        db:         db,
        bookClient: bookClient,
    }
}
```

- bookClient: gRPC client to communicate with Book Service.
- Dependency injection pattern: Client is passed in during server creation.
- Constructor function that accepts both dependencies and clean way to initialize the server with injected dependencies.

```
func (s *authorCatalogServer) GetAuthor(ctx context.Context, req *authorpb.GetAuthorRequest) (*authorpb.GetAuthorResponse, error) {
    log.Printf("GetAuthor: id=%d", req.Id)

    var author authorpb.Author
    err := s.db.QueryRowContext(ctx,
        "SELECT id, name, bio, birth_year, country FROM authors WHERE id = ?",
        req.Id,
    ).Scan(&author.Id, &author.Name, &author.Bio, &author.BirthYear, &author.Country)

    if err == sql.ErrNoRows {
        return nil, status.Errorf(codes.NotFound, "author not found: id=%d", req.Id)
    }
    if err != nil {
        return nil, status.Errorf(codes.Internal, "database error: %v", err)
    }

    return &authorpb.GetAuthorResponse{
        Author: &author,
    }, nil
}
```

GetAuthor

- Log incoming request.
- Execute SELECT query to fetch author by ID.
- Scan result into Author struct.
- Return NotFound error if no row exists.
- Return internal error for other DB issues.

- Return GetAuthorResponse with the author if successful.

```go
func (s *authorCatalogServer) CreateAuthor(ctx context.Context, req *authorpb.CreateAuthorRequest) (*authorpb.CreateAuthorResponse, error) {
    log.Printf("CreateAuthor: name=%s", req.Name)

    // Validate input
    if req.Name == "" {
        return nil, status.Error(codes.InvalidArgument, "name is required")
    }

    // Insert author into database
    result, err := s.db.ExecContext(ctx,
        "INSERT INTO authors (name, bio, birth_year, country) VALUES (?, ?, ?, ?)",
        req.Name, req.Bio, req.BirthYear, req.Country,
    )
    if err != nil {
        return nil, status.Errorf(codes.Internal, "failed to insert author: %v", err)
    }

    // Get the last inserted ID
    id, err := result.LastInsertId()
    if err != nil {
        return nil, status.Errorf(codes.Internal, "failed to get author ID: %v", err)
    }

    return &authorpb.CreateAuthorResponse{
        Author: &authorpb.Author{
            Id:        int32(id),
            Name:      req.Name,
            Bio:       req.Bio,
            BirthYear: req.BirthYear,
            Country:   req.Country,
        },
    }, nil
}
```

CreateAuthor

- Validate that name field is not empty.
- Execute INSERT statement into authors table.
- Retrieve auto-generated ID from the DB.
- Construct Author object with new ID.
- Return response containing the created author.

```go
func (s *authorCatalogServer) ListAuthors(ctx context.Context, req *authorpb.ListAuthorsRequest) (*authorpb.ListAuthorsResponse, error) {
    log.Printf("ListAuthors: page=%d, page_size=%d", req.Page, req.PageSize)

    // Set defaults
    if req.Page <= 0 {
        req.Page = 1
    }
    if req.PageSize <= 0 {
        req.PageSize = 10
    }

    // Get total count
    var total int32
    err := s.db.QueryRowContext(ctx, "SELECT COUNT(*) FROM authors").Scan(&total)
    if err != nil {
        return nil, status.Errorf(codes.Internal, "failed to count authors: %v", err)
    }

    // Calculate offset
    offset := (req.Page - 1) * req.PageSize

    // Query authors with pagination
    rows, err := s.db.QueryContext(ctx,
        "SELECT id, name, bio, birth_year, country FROM authors LIMIT ? OFFSET ?",
        req.PageSize, offset,
    )
    if err != nil {
        return nil, status.Errorf(codes.Internal, "query failed: %v", err)
    }
    defer rows.Close()

    var authors []*authorpb.Author
    for rows.Next() {
        var author authorpb.Author
        err := rows.Scan(&author.Id, &author.Name, &author.Bio, &author.BirthYear, &author.Country)
        if err != nil {
            return nil, status.Errorf(codes.Internal, "scan failed: %v", err)
        }
        authors = append(authors, &author)
    }

    return &authorpb.ListAuthorsResponse{
        Authors: authors,
        Total:   total,
    }, nil
}
```

ListAuthors (Pagination)

- Log request and apply default page and page_size.
- Query total number of authors to calculate pagination.
- Compute offset based on current page.
- Fetch authors using LIMIT + OFFSET.
- Scan each row into Author struct.
- Return paginated list plus total count.

```go
func (s *authorCatalogServer) GetAuthorBooks(ctx context.Context, req *authorpb.GetAuthorBooksRequest) (*authorpb.GetAuthorBooksResponse, error) {
    log.Printf("GetAuthorBooks: author_id=%d", req.AuthorId)

    // Get author from database
    var author authorpb.Author
    err := s.db.QueryRowContext(ctx,
        "SELECT id, name, bio, birth_year, country FROM authors WHERE id = ?",
        req.AuthorId,
    ).Scan(&author.Id, &author.Name, &author.Bio, &author.BirthYear, &author.Country)

    if err == sql.ErrNoRows {
        return nil, status.Errorf(codes.NotFound, "author not found: id=%d", req.AuthorId)
    }
    if err != nil {
        return nil, status.Errorf(codes.Internal, "database error: %v", err)
    }

    // Call Book service to get books by this author
    // This demonstrates service-to-service communication!
    bookResp, err := s.bookClient.GetBooksByAuthor(ctx, &bookpb.GetBooksByAuthorRequest{
        AuthorId: req.AuthorId,
    })
    if err != nil {
        log.Printf("Failed to get books: %v", err)
        // Continue even if book service fails
        return &authorpb.GetAuthorBooksResponse{
            Author:    &author,
            Books:     nil,
            BookCount: 0,
        }, nil
    }

    // Convert books to BookSummary
    var bookSummaries []*authorpb.BookSummary
    for _, book := range bookResp.Books {
        bookSummaries = append(bookSummaries, &authorpb.BookSummary{
            Id:            book.Id,
            Title:         book.Title,
            Price:         book.Price,
            PublishedYear: book.PublishedYear,
        })
    }

    return &authorpb.GetAuthorBooksResponse{
        Author:    &author,
        Books:     bookSummaries,
        BookCount: int32(len(bookSummaries)),
    }, nil
}
```

GetAuthorBooks – Cross-Service Communication

- Check local DB to confirm the author exists.
- Return NotFound if missing.
- Call Book Service via gRPC (GetBooksByAuthor) with the same context.
- If Book Service fails: degrade gracefully → return author with empty book list.
- If successful: convert returned books into lightweight BookSummary objects.
- Combine author info + book summaries into one unified response.

```go
func connectToBookService() (bookpb.BookCatalogClient, error) {
    conn, err := grpc.Dial("localhost:50051",
        grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        return nil, err
    }

    return bookpb.NewBookCatalogClient(conn), nil
}
```

- connectToBookService establishes gRPC connection to Book Service at port 50051.
- Uses insecure credentials for internal communication.
- Connection stays open for the Author Service lifetime.

```go
func initDB() (*sql.DB, error) {
    db, err := sql.Open("sqlite3", "./authors.db")
    if err != nil {
        return nil, err
    }

    // Create authors table
    _, err = db.Exec(`
        CREATE TABLE IF NOT EXISTS authors (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            bio TEXT,
            birth_year INTEGER,
            country TEXT
        );
    `)
    if err != nil {
        return nil, err
    }

    // Seed sample authors (only if table is empty)
    var count int
    db.QueryRow("SELECT COUNT(*) FROM authors").Scan(&count)
    if count == 0 {
        log.Println("Seeding sample authors...")
        _, err = db.Exec(`
            INSERT INTO authors (name, bio, birth_year, country) VALUES
            ('J.K. Rowling', 'British author, best known for Harry Potter series', 1965, 'UK'),
            ('George R.R. Martin', 'American novelist, author of A Song of Ice and Fire', 1948, 'USA'),
            ('Stephen King', 'American author of horror and suspense novels', 1947, 'USA'),
            ('Agatha Christie', 'English writer known for detective novels', 1890, 'UK'),
            ('Isaac Asimov', 'American writer and professor of biochemistry', 1920, 'USA');
        `)
        if err != nil {
            return nil, err
        }
    }

    return db, nil
}
```

- initDB opens or creates authors.db.
- Ensures authors table exists.

- Seeds five famous authors if table is empty.
- Follows database-per-service pattern: each service manages its own schema/data.

```go
func main() {
    // Initialize database
    db, err := initDB()
    if err != nil {
        log.Fatalf("Failed to init DB: %v", err)
    }
    defer db.Close()

    // Connect to Book service
    bookClient, err := connectToBookService()
    if err != nil {
        log.Fatalf("Failed to connect to Book service: %v", err)
    }

    // Create listener on port 50052
    lis, err := net.Listen("tcp", ":50052")
    if err != nil {
        log.Fatalf("Failed to listen: %v", err)
    }

    // Create gRPC server
    grpcServer := grpc.NewServer()

    // Register service
    authorpb.RegisterAuthorCatalogServer(grpcServer, newServer(db, bookClient))

    log.Println("🚀 Author Catalog gRPC server listening on :50052")
    log.Println("📚 Connected to Book Catalog service on :50051")

    // Start serving
    if err := grpcServer.Serve(lis); err != nil {
        log.Fatalf("Failed to serve: %v", err)
    }
}
```

- Initialize database via initDB.
- Connect to Book Service.
- Listen on port 50052.
- Create and run gRPC server.
- Register Author Catalog service with DB handle + Book Service client.
- Log key startup steps for clarity and debugging.

Output:

Book Service:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab6\Task5\bookservice> go run main.go
Seeding sample books...
📚 Book Catalog gRPC server running on :50051
2025/12/03 11:00:17 CreateBook: title=Refactoring, author_id=6
2025/12/03 11:03:01 CreateBook: title=Refactoring, author_id=7
2025/12/03 11:03:01 CreateBook: title=Patterns of Enterprise Application Architecture, author_id=7
2025/12/03 11:03:01 GetBooksByAuthor: author_id=7
2025/12/03 11:05:30 CreateBook: title=Refactoring, author_id=8
2025/12/03 11:05:30 CreateBook: title=Patterns of Enterprise Application Architecture, author_id=8
2025/12/03 11:05:30 GetBooksByAuthor: author_id=8
```

Author Service:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab6\Task5\authorservice> go run .\main.go
2025/12/03 11:00:05 Seeding sample authors...
2025/12/03 11:00:05 🚀 Author Catalog gRPC server listening on :50052
2025/12/03 11:00:05 📚 Connected to Book Catalog service on :50051
2025/12/03 11:00:17 CreateAuthor: name=Martin Fowler
2025/12/03 11:03:01 CreateAuthor: name=Martin Fowler
2025/12/03 11:03:01 GetAuthorBooks: author_id=7
2025/12/03 11:05:30 CreateAuthor: name=Martin Fowler
2025/12/03 11:05:30 GetAuthorBooks: author_id=8
```

Client:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab6\Task5\client> go run .\main.go
=== Microservice Demo ===

1. Creating author...
✓ Created author: Martin Fowler (ID: 8)

2. Creating books for author...
✓ Created book: Refactoring
✓ Created book: Patterns of Enterprise Application Architecture

3. Fetching author's books (cross-service call)...
✓ Author: Martin Fowler
✓ Books written: 2
  1. Refactoring (2018)
  2. Patterns of Enterprise Application Architecture (2002)

✓ Microservice demo completed!
```