

# Go + Flutter Course

## Database & Persistence

Timur Harin

Lecture 04: Database & Persistence

*Building robust data storage solutions*

# Block 4: Database & Persistence

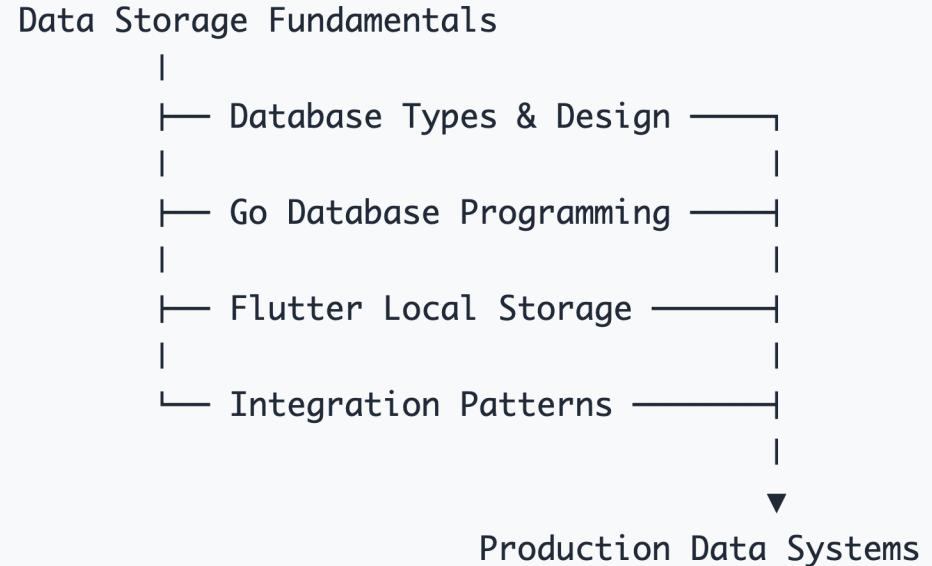
## Lecture 04 Overview

- **Database Fundamentals:** SQL, NoSQL, design principles
- **Go Database Programming:** SQL drivers, ORMs, transactions
- **Flutter Local Storage:** SQLite, SharedPreferences, secure storage
- **Integration:** Data sync, caching, offline-first patterns

## What we'll learn

- Why databases exist and how they evolved
- SQL vs NoSQL database types and use cases
- ACID properties and transaction management
- Go database programming with database/sql
- Flutter local storage solutions and patterns
- Data synchronization and offline strategies
- Performance optimization and best practices

# Learning path



- **Foundation:** Database theory and design principles
- **Backend:** Go database programming and SQL operations
- **Frontend:** Flutter local storage and persistence
- **Integration:** Real-world data synchronization patterns

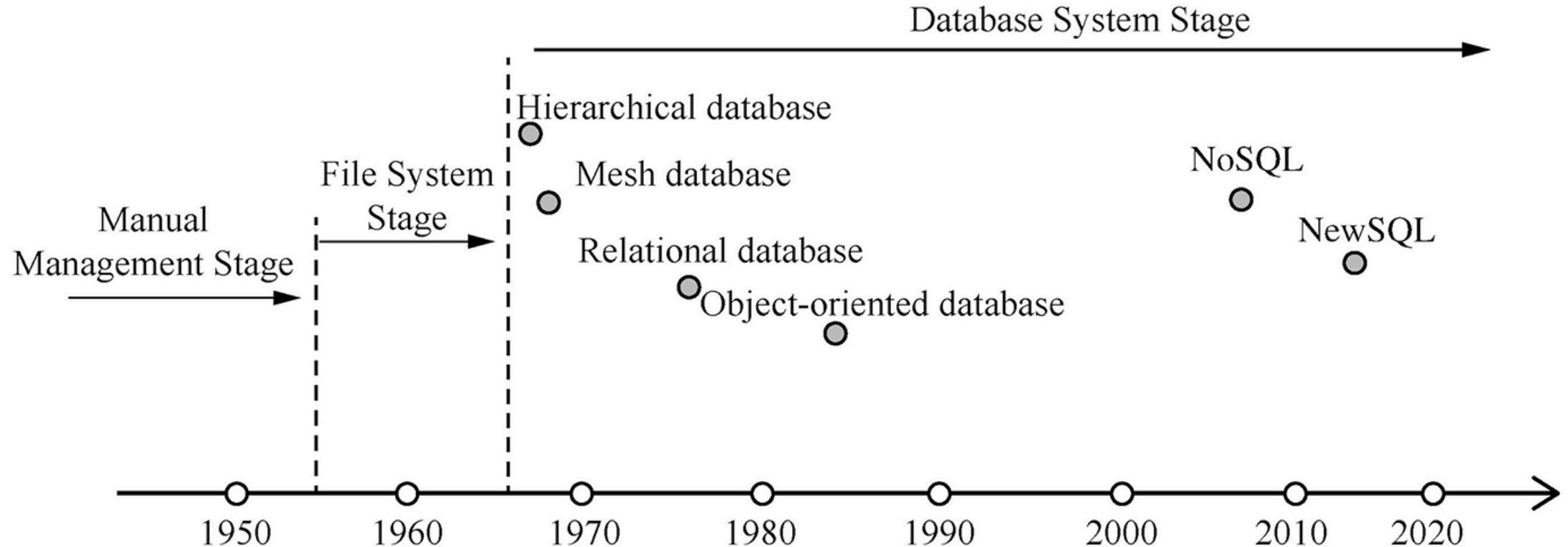
# Part I: Database fundamentals

**Database** is an organized collection of structured information, or data, typically stored electronically in a computer system.

## Why databases exist

- **Persistence:** Data survives application restarts
- **Concurrency:** Multiple users can access data simultaneously
- **Consistency:** Data integrity and validation rules
- **Performance:** Optimized storage and retrieval
- **Security:** Access control and audit trails
- **Backup & Recovery:** Data protection mechanisms

# Database history



# Relational (SQL) Databases

```
-- Structured data with relationships
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE,
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    title VARCHAR(200) NOT NULL,
    content TEXT,
    published_at TIMESTAMP
);

-- Complex queries with joins
SELECT u.name, p.title, p.published_at
FROM users u
JOIN posts p ON u.id = p.user_id
WHERE p.published_at > '2025-01-01'
ORDER BY p.published_at DESC;
```

# NoSQL Databases

```
{ // Document-based (MongoDB, CouchDB)
  "id": "user_123",
  "name": "John Doe",
  "email": "john@example.com",
  "posts": [
    {
      "title": "My First Post",
      "content": "Hello world!",
      "published_at": "2025-07-09T10:00:00Z"
    }
  ],
  "preferences": {
    "theme": "dark",
    "notifications": true
  }
}

// Key-Value (Redis, DynamoDB)
SET user:123:name "John Doe"
SET user:123:email "john@example.com"
HSET user:123:prefs theme dark notifications true
```

# SQL vs NoSQL comparison

Aspect	SQL (Relational)	NoSQL
<b>Schema</b>	Fixed, predefined	Flexible, dynamic
<b>Scaling</b>	Vertical (scale up)	Horizontal (scale out)
<b>ACID</b>	Full ACID compliance	Eventually consistent
<b>Queries</b>	SQL (standardized)	Database-specific APIs
<b>Relationships</b>	Foreign keys, joins	Denormalized, embedded

## When to choose SQL

- **ACID compliance** is critical (banking, e-commerce)
- **Complex relationships** between entities
- **Mature ecosystem** and tooling needed
- **Team expertise** with SQL

## When to choose NoSQL

- **Rapid development** with changing requirements
- **Massive scale** and high throughput needed
- **Unstructured data** (JSON, documents, graphs)
- **Real-time analytics** and big data

## Atomicity

```
-- All operations succeed or all fail
BEGIN;
UPDATE accounts
SET balance = balance - 100
WHERE id = 1;
UPDATE accounts
SET balance = balance + 100
WHERE id = 2;
COMMIT; -- Both updates or neither
```

## Isolation

```
-- Concurrent transactions don't interfere
-- Transaction 1
BEGIN;
SELECT balance FROM accounts WHERE id = 1;
-- Other transactions can't see changes yet
-- Transaction 2 (runs simultaneously)
BEGIN;
SELECT balance FROM accounts WHERE id = 1;
-- Sees original value, not Transaction 1's changes
```

## Consistency

```
-- Database rules are always enforced
ALTER TABLE accounts
ADD CONSTRAINT positive_balance
CHECK (balance >= 0);
-- This will fail if balance goes negative
UPDATE accounts
SET balance = balance - 1000
WHERE id = 1;
```

## Durability

```
-- Committed changes survive system failures
COMMIT; -- Data is written to disk
-- Even if power fails, this data persists
```

# Database design principles

## 1. Normalization (reducing redundancy)

```
-- Before normalization (redundant data)
CREATE TABLE orders_bad (
    id SERIAL PRIMARY KEY,
    customer_name VARCHAR(100),
    customer_email VARCHAR(255),
    customer_address TEXT,
    product_name VARCHAR(200),
    product_price DECIMAL(10,2),
    quantity INTEGER
);
```

```
-- After normalization (3NF)
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(255),
    address TEXT);
```

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(200),
    price DECIMAL(10,2)
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    customer_id INTEGER REFERENCES customers(id),
    product_id INTEGER REFERENCES products(id),
    quantity INTEGER,
    order_date TIMESTAMP DEFAULT NOW()
);
```

# Database design principles (continued)

## 2. Indexing for performance

```
-- Create indexes for frequently queried columns  
CREATE INDEX idx_users_email ON users(email);  
CREATE INDEX idx_posts_user_id ON posts(user_id);  
CREATE INDEX idx_posts_published_at ON posts(published_at);  
  
-- Composite index for multiple columns  
CREATE INDEX idx_user_posts_date  
ON posts(user_id, published_at);  
  
-- Unique index for constraints  
CREATE UNIQUE INDEX idx_users_email_unique  
ON users(email);
```

```
-- Query performance comparison  
-- Without index: Full table scan  
EXPLAIN SELECT * FROM posts  
WHERE user_id = 123;  
-- Cost: 1000 (table scan)  
  
-- With index: Index seek  
EXPLAIN SELECT * FROM posts  
WHERE user_id = 123;  
-- Cost: 5 (index lookup)  
  
-- Bad query (no index on title)  
SELECT * FROM posts  
WHERE title LIKE '%golang%';  
-- Should add: CREATE INDEX idx_posts_title_gin  
-- ON posts USING gin(to_tsvector('english', title));
```

# Common database patterns

## One-to-Many

```
-- User has many posts
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    title VARCHAR(200)
);

-- Query: Get user with all posts
SELECT u.name, p.title
FROM users u
LEFT JOIN posts p ON u.id = p.user_id
WHERE u.id = 1;
```

## Many-to-Many

```
-- Posts have many tags, tags have many posts
CREATE TABLE tags (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) UNIQUE
);

CREATE TABLE post_tags (
    post_id INTEGER REFERENCES posts(id),
    tag_id INTEGER REFERENCES tags(id),
    PRIMARY KEY (post_id, tag_id)
);

-- Query: Get posts with specific tag
SELECT p.title
FROM posts p
JOIN post_tags pt ON p.id = pt.post_id
JOIN tags t ON pt.tag_id = t.id
WHERE t.name = 'golang';
```

# Part II: Go database programming

“ Go's `database/sql` package provides a lightweight, efficient interface to SQL databases with excellent performance and safety features. ”

## Key advantages

- **Database agnostic:** Works with any SQL database driver
- **Connection pooling:** Automatic management of database connections
- **Prepared statements:** Protection against SQL injection
- **Null handling:** Safe handling of nullable database values
- **Transactions:** Full ACID transaction support
- **Context support:** Timeout and cancellation

## Popular drivers

- **PostgreSQL:** [github.com/lib/pq](https://github.com/lib/pq), [github.com/jackc/pgx](https://github.com/jackc/pgx)
- **MySQL:** [github.com/go-sql-driver/mysql](https://github.com/go-sql-driver/mysql)
- **SQLite:** [github.com/mattn/go-sqlite3](https://github.com/mattn/go-sqlite3), [modernc.org/sqlite](https://modernc.org/sqlite)

# Go SQL drivers overview

“ Go's `database/sql` package works with various database drivers, each optimized for specific.”

## Popular Go SQL drivers

Database	Driver Package	Features
PostgreSQL	<a href="https://github.com/lib/pq">github.com/lib/pq</a>	Pure Go, stable, widely used
PostgreSQL	<a href="https://github.com/jackc/pgx">github.com/jackc/pgx</a>	Native protocol, better performance
MySQL	<a href="https://github.com/go-sql-driver/mysql">github.com/go-sql-driver/mysql</a>	Pure Go, full MySQL support
SQLite	<a href="https://github.com/mattn/go-sqlite3">github.com/mattn/go-sqlite3</a>	CGO-based, requires GCC
SQLite	<a href="https://modernc.org/sqlite">modernc.org/sqlite</a>	Pure Go, no CGO required
SQL Server	<a href="https://github.com/denisenkom/go-mssql">github.com/denisenkom/go-mssql</a>	Pure Go implementation

- **Pure Go vs CGO:** Pure Go drivers are easier to deploy
- **Performance:** Native protocol drivers (like pgx) offer better performance
- **Cross-compilation:** Pure Go drivers work better for cross-platform builds

# PostgreSQL setup options

## 1. Local installation (macOS example) - package manager

```
# Install PostgreSQL via Homebrew
brew install postgresql
brew services start postgresql
# Create database and user
createdb myapp_development
psql myapp_development
```

## 2. Docker containerized

```
# Run PostgreSQL in Docker
docker run --name postgres-dev \
-e POSTGRES_DB=myapp_development \
-e POSTGRES_USER=myuser \
-e POSTGRES_PASSWORD=mypassword \
-p 5432:5432 -d postgres:15
```

## 3. Remote (Cloud providers)

- **AWS RDS**: Managed PostgreSQL service
- **Google Cloud SQL**: Fully managed database
- **Heroku Postgres**: Add-on for Heroku apps
- **DigitalOcean**: Managed databases

*We'll focus on local development setup*

# Database connection setup

## Basic PostgreSQL connection

```
import (
    "database/sql"
    _ "github.com/lib/pq"
)

func setupPostgreSQL() (*sql.DB, error) {
    dsn := "postgres://user:pass@localhost/db?sslmode=disable"

    db, err := sql.Open("postgres", dsn)
    if err != nil {
        return nil, err
    }

    if err := db.Ping(); err != nil {
        return nil, err
    }

    return db, nil
}
```

## Production-ready connection

```
func setupProductionDB() (*sql.DB, error) {
    db, err := sql.Open("postgres", dsn)
    if err != nil {
        return nil, err
    }

    // Connection pool configuration
    db.SetMaxOpenConns(25)
    db.SetMaxIdleConns(5)
    db.SetConnMaxLifetime(5 * time.Minute)
    db.SetConnMaxIdleTime(2 * time.Minute)

    // Verify connection
    if err := db.Ping(); err != nil {
        return nil, err
    }

    return db, nil
}
```

# Migration strategies

**“Database migrations** manage schema changes over time, ensuring consistent database structure across environments.

## Manual vs Automated migrations

### Manual migrations

```
// Simple migration runner
func runMigrations(db *sql.DB) error {
    migrations := []string{
        "CREATE TABLE users (...)",
        "ALTER TABLE users ADD COLUMN...",
        "CREATE INDEX idx_users_email...",
    }
    for _, migration := range migrations {
        if _, err := db.Exec(migration); err != nil {
            return err
        }
    }
    return nil
}
```

### Goose migration tool

```
# Install goose
go install github.com/pressly/goose/v3/cmd/goose@latest

# Create migration
goose create add_users_table sql

# Run migrations
goose postgres $DSN up

# Rollback
goose postgres $DSN down
```

# Migration enumeration strategies

## 1. Sequential numbers

```
001_create_users.sql  
002_add_posts_table.sql  
003_add_user_index.sql  
004_add_posts_index.sql
```

### Pros:

- Simple ordering
- Easy to understand sequence
- Deterministic order

### Cons:

- Merge conflicts in teams
- Hard to reorder
- Branch management issues

## 2. Timestamp-based

```
20250709120000_create_users.sql  
20250709120500_add_posts_table.sql  
20250709121000_add_user_index.sql  
20250710093000_add_posts_index.sql
```

### Pros:

- No merge conflicts
- Natural chronological order
- Easy branch merging

### Cons:

- Longer filenames
- Less obvious sequence
- Timezone considerations

## Recommended for teams

# Goose migration example

## Creating a migration

```
# Create new migration file
goose create add_users_table sql
# Creates: 20250709120000_add_users_table.sql
```

## Migration file structure

```
-- +goose Up
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT NOW()
);
CREATE INDEX idx_users_email ON users(email);

-- +goose Down
DROP TABLE users;
```

## Running migrations

```
# Run all pending migrations
goose postgres $DATABASE_URL up

# Run specific number of migrations
goose postgres $DATABASE_URL up-by-one

# Check migration status
goose postgres $DATABASE_URL status

# Rollback last migration
goose postgres $DATABASE_URL down

# Reset database (careful!)
goose postgres $DATABASE_URL reset
```

# SQL interaction approaches

“ Go offers multiple ways to interact with SQL databases, each with different trade-offs between performance, safety, and developer experience. ”

## Approaches overview

1. **Manual SQL** - Raw queries with `database/sql`
2. **Mapping libraries** - `scany` for result mapping
3. **Query builders** - `squirrel` for dynamic queries
4. **ORMs** - `gorm` for object-relational mapping
5. **Code generation** - `sqlc` for type-safe queries

## Key considerations

- **Performance**: How fast are queries executed?
- **Type safety**: Compile-time error detection
- **Boilerplate**: Amount of code required
- **Learning curve**: How easy to adopt?

# Manual SQL approach

## Raw SQL queries

```
func getUserByID(db *sql.DB, id int) (*User, error) {
    query := `SELECT id, name, email, created_at
               FROM users
              WHERE id = $1 `
    var user User
    err := db.QueryRow(query, id).Scan(
        &user.ID,
        &user.Name,
        &user.Email,
        &user.CreatedAt,
    )
    if err != nil {
        return nil, err
    }
    return &user, nil
}
```

## Complex queries

```
func getUserPosts(db *sql.DB, userID int) ([]Post, error) {
    query := `SELECT p.id, p.title, p.content,
                  u.name as author_name
             FROM posts p
            JOIN users u ON p.user_id = u.id
           WHERE p.user_id = $1
           ORDER BY p.created_at DESC`
    rows, err := db.Query(query, userID)
    if err != nil {
        return nil, err
    }
    defer rows.Close()
    var posts []Post
    for rows.Next() {
        var post Post
        err := rows.Scan(&post.ID, &post.Title,
                        &post.Content, &post.AuthorName)
        if err != nil {
            return nil, err
        }
        posts = append(posts, post)
    }
    return posts, nil
}
```

# Scany mapping library

## Installation and setup

```
go get github.com/georgysavva/scany/v2/pgxscan
```

```
import (
    "github.com/georgysavva/scany/v2/pgxscan"
    "github.com/jackc/pgx/v5/pgxpool"
)

type User struct {
    ID      int      `db:"id"`
    Name    string   `db:"name"`
    Email   string   `db:"email"`
    CreatedAt time.Time `db:"created_at"`
}
```

## Simple mapping

```
func getUserByID(db *pgxpool.Pool, id int) (*User, error) {
    query := `
        SELECT id, name, email, created_at
        FROM users
        WHERE id = $1
    `

    var user User
    err := pgxscan.Get(context.Background(), db, &user, query, id)
    if err != nil {
        return nil, err
    }
    return &user, nil
}

func getAllUsers(db *pgxpool.Pool) ([]User, error) {
    query := `SELECT id, name, email, created_at FROM users`
    var users []User
    err := pgxscan.Select(context.Background(), db, &users, query)
    return users, err
}
```

# Squirrel query builder

## Installation and basic usage

```
go get github.com/Masterminds/squirrel

import "github.com/Masterminds/squirrel"
// Use PostgreSQL placeholder format
var psql = squirrel.StatementBuilder.PlaceholderFormat(squirrel.Dollar)
func getUserByID(db *sql.DB, id int) (*User, error) {
    sql, args, err := psql.
        Select("id", "name", "email", "created_at").
        From("users").
        Where(squirrel.Eq{"id": id}).
        ToSql()
    if err != nil {
        return nil, err
    }
    var user User
    err = db.QueryRow(sql, args...).Scan(
        &user.ID, &user.Name, &user.Email, &user.CreatedAt,
    )
    return &user, err
}
```

## Dynamic query building

```
func searchUsers(db *sql.DB, filters UserFilters) ([]User, error) {
    query := psql.Select("id", "name", "email").From("users")
    if filters.Name != "" {
        query = query.Where(squirrel.Like{"name": "%" + filters.Name + "%"})
    }
    if filters.Email != "" {
        query = query.Where(squirrel.Eq{"email": filters.Email})
    }
    if !filters.CreatedAfter.IsZero() {
        query = query.Where(squirrel.Gt{"created_at": filters.CreatedAfter})
    }
    query = query.OrderBy("created_at DESC").
        Limit(uint64(filters.Limit)).
        Offset(uint64(filters.Offset))
    sql, args, err := query.ToSql()
    if err != nil {
        return nil, err
    }
    rows, err := db.Query(sql, args...)
    // ... scan results
    return users, err
}
```

# GORM ORM approach

## Installation and setup

```
go get gorm.io/gorm
go get gorm.io/driver/postgres

import (
    "gorm.io/gorm"
    "gorm.io/driver/postgres"
)

type User struct {
    ID      uint      `gorm:"primaryKey"`
    Name    string    `gorm:"size:100;not null"`
    Email   string    `gorm:"uniqueIndex;not null"`
    Posts   []Post    `gorm:"foreignKey:UserID"`
    CreatedAt time.Time
    UpdatedAt time.Time
    DeletedAt gorm.DeletedAt `gorm:"index"`
}

func setupGORM() (*gorm.DB, error) {
    dsn := "host=localhost user=postgres dbname=test"
    return gorm.Open(postgres.Open(dsn), &gorm.Config{})
}
```

## GORM operations

```
func createUser(db *gorm.DB, user *User) error {
    return db.Create(user).Error }

func getUserByID(db *gorm.DB, id uint) (*User, error) {
    var user User
    err := db.Preload("Posts").First(&user, id).Error
    return &user, err }

func searchUsers(db *gorm.DB, name string) ([]User, error) {
    var users []User
    err := db.Where("name ILIKE ?", "%" + name + "%").
        Order("created_at desc").
        Find(&users).Error
    return users, err }

func updateUser(db *gorm.DB, user *User) error {
    return db.Save(user).Error }

func deleteUser(db *gorm.DB, id uint) error {
    return db.Delete(&User{}, id).Error // Soft delete
}
```

# SQLC code generation

## Installation and configuration

```
go install github.com/sqlc-dev/sqlc/cmd/sqlc@latest
```

```
# sqlc.yaml
version: "2"
sql:
  - engine: "postgresql"
    queries: "./sql/queries/"
    schema: "./sql/schema/"
gen:
  go:
    package: "db"
    out: "./internal/db"
    emit_json_tags: true
    emit_prepared_queries: false
    emit_interface: true
```

## Schema definition

```
-- sql/schema/001_users.sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT NOW()
);
```

# SQLC code generation (continued)

## Query definitions

```
-- sql/queries/users.sql
-- name: GetUser :one
SELECT * FROM users
WHERE id = $1 LIMIT 1;
-- name: ListUsers :many
SELECT * FROM users
ORDER BY name;
-- name: CreateUser :one
INSERT INTO users (name, email)
VALUES ($1, $2)
RETURNING *;
-- name: UpdateUser :exec
UPDATE users
SET name = $2, email = $3
WHERE id = $1;
-- name: DeleteUser :exec
DELETE FROM users
WHERE id = $1;
```

## Generated usage

```
// Auto-generated type-safe functions
func (q *Queries) GetUser(ctx context.Context,
                           id int32) (User, error)
func (q *Queries) ListUsers(ctx context.Context) ([]User, error)
func (q *Queries) CreateUser(ctx context.Context,
                            arg CreateUserParams) (User, error)
```

# Approach comparison table

Approach	Performance	Type Safety	Boilerplate	Learning Curve	Complex Queries
Manual SQL	★★★★★	★★	★★	★★★	★★★★★
Scany	★★★★★	★★★	★★★★	★★★★	★★★★★
Squirrel	★★★★★	★★★	★★★★	★★★	★★★★★
GORM	★★★	★★★★★	★★★★★	★★	★★★★
SQLC	★★★★★	★★★★★	★★★★★	★★★	★★★★★

## Recommendations

- **High performance + simple queries:** Manual SQL or SQLC
- **Rapid development:** GORM
- **Dynamic queries:** Squirrel
- **Type safety priority:** SQLC
- **Team new to SQL:** GORM

# CRUD with raw SQL: Data models

## Basic user model

```
type User struct {
    ID      int      `json:"id" db:"id"`
    Name    string   `json:"name" db:"name"`
    Email   string   `json:"email" db:"email"`
    PasswordHash string   `json:"-" db:"password_hash"`
    CreatedAt time.Time `json:"created_at" db:"created_at"`
    UpdatedAt time.Time `json:"updated_at" db:"updated_at"`
}

type Post struct {
    ID      int      `json:"id" db:"id"`
    UserID  int      `json:"user_id" db:"user_id"`
    Title   string   `json:"title" db:"title"`
    Content  string   `json:"content" db:"content"`
    Published bool     `json:"published" db:"published"`
    CreatedAt time.Time `json:"created_at" db:"created_at"`
    UpdatedAt time.Time `json:"updated_at" db:"updated_at"`
}
```

## Handling nullable fields

```
import "database/sql"

type UserProfile struct {
    ID      int      `db:"id"`
    Name    string   `db:"name"`
    Email   string   `db:"email"`
    Bio     sql.NullString `db:"bio"`
    Age    sql.NullInt64 `db:"age"`
    LastLoginAt sql.NullTime `db:"last_login_at"`
}

// Helper methods for null types
func (up *UserProfile) GetBio() string {
    if up.Bio.Valid {
        return up.Bio.String
    } return ""
}
func (up *UserProfile) SetBio(bio string) {
    if bio == "" {
        up.Bio = sql.NullString{Valid: false}
    } else {
        up.Bio = sql.NullString{String: bio, Valid: true}}
}
```

# CRUD: Create operations

## Single user creation

```
func createUser(db *sql.DB, user *User) error {
    query := `
        INSERT INTO users (name, email, password_hash)
        VALUES ($1, $2, $3)
        RETURNING id, created_at, updated_at
    `

    err := db.QueryRow(
        query,
        user.Name,
        user.Email,
        user.PasswordHash,
    ).Scan(
        &user.ID,
        &user.CreatedAt,
        &userUpdatedAt,
    )
    return err
}
```

## Batch user creation

```
func createUsers(db *sql.DB, users []User) error {
    tx, err := db.Begin()
    if err != nil {
        return err
    }
    defer tx.Rollback()
    stmt, err := tx.Prepare(`
        INSERT INTO users (name, email, password_hash)
        VALUES ($1, $2, $3)
    `)
    if err != nil {
        return err
    }
    defer stmt.Close()
    for _, user := range users {
        _, err := stmt.Exec(user.Name, user.Email, user.PasswordHash)
        if err != nil {
            return err
        }
    }
    return tx.Commit()
}
```

# CRUD: Read operations

## Get single user

```
func getUserByID(db *sql.DB, id int) (*User, error) {
    user := &User{}
    query := `

        SELECT id, name, email, password_hash,
               created_at, updated_at
        FROM users
        WHERE id = $1
    `

    err := db.QueryRow(query, id).Scan(
        &user.ID, &user.Name, &user.Email,
        &user.PasswordHash, &user.CreatedAt, &user.UpdatedAt,
    )
    if err != nil {
        if err == sql.ErrNoRows {
            return nil, fmt.Errorf("user not found")
        }
        return nil, err
    }
    return user, nil
}
```

## Get multiple users with pagination

```
func getUsers(db *sql.DB, limit, offset int) ([]User, error) {
    query := `SELECT id, name, email, created_at, updated_at
              FROM users
              ORDER BY created_at DESC
              LIMIT $1 OFFSET $2`
    rows, err := db.Query(query, limit, offset)
    if err != nil {
        return nil, err
    }
    defer rows.Close()
    var users []User
    for rows.Next() {
        var user User
        err := rows.Scan(&user.ID, &user.Name, &user.Email,
                         &user.CreatedAt, &user.UpdatedAt)
        if err != nil {
            return nil, err
        }
        users = append(users, user)
    }
    return users, rows.Err()
}
```

# CRUD: Update operations

## Full user update

```
func updateUser(db *sql.DB, user *User) error {
    query := `UPDATE users
        SET name = $1, email = $2, updated_at = NOW()
        WHERE id = $3
        RETURNING updated_at
    `

    err := db.QueryRow(
        query,
        user.Name,
        user.Email,
        user.ID,
    ).Scan(&user.UpdatedAt)

    return err
}
```

## Partial update with dynamic fields

```
func updateUserFields(db *sql.DB, id int,
                      fields map[string]interface{}) error {
    if len(fields) == 0 {
        return fmt.Errorf("no fields to update")
    }
    setParts := []string{"updated_at = NOW()"}
    args := []interface{}{}
    argIndex := 1
    for field, value := range fields {
        setParts = append(setParts, fmt.Sprintf("%s = $%d", field, argIndex))
        args = append(args, value)
        argIndex++
    }
    query := fmt.Sprintf("UPDATE users SET %s WHERE id = $%d",
                         strings.Join(setParts, ", "), argIndex)
    args = append(args, id)
    result, err := db.Exec(query, args...)
    if err != nil {
        return err
    }
    if rowsAffected, _ := result.RowsAffected(); rowsAffected == 0 {
        return fmt.Errorf("user not found")
    }
    return nil
}
```

# CRUD: Delete operations

## Hard delete

```
func deleteUser(db *sql.DB, id int) error {
    query := `DELETE FROM users WHERE id = $1`

    result, err := db.Exec(query, id)
    if err != nil {
        return err
    }
    rowsAffected, err := result.RowsAffected()
    if err != nil {
        return err
    }
    if rowsAffected == 0 {
        return fmt.Errorf("user not found")
    }
    return nil
}
```

## Soft delete (recommended)

```
func softDeleteUser(db *sql.DB, id int) error {
    query := `UPDATE users
              SET deleted_at = NOW(), updated_at = NOW()
              WHERE id = $1 AND deleted_at IS NULL`
    result, err := db.Exec(query, id)
    if err != nil {
        return err
    }
    if rowsAffected, _ := result.RowsAffected(); rowsAffected == 0 {
        return fmt.Errorf("user not found or already deleted")
    } return nil }

func restoreUser(db *sql.DB, id int) error {
    query := `UPDATE users
              SET deleted_at = NULL, updated_at = NOW()
              WHERE id = $1 AND deleted_at IS NOT NULL`
    result, err := db.Exec(query, id)
    if err != nil {
        return err
    }
    if rowsAffected, _ := result.RowsAffected(); rowsAffected == 0 {
        return fmt.Errorf("user not found or not deleted")
    } return nil }
```

# Transactions in Go

```
func transferMoney(db *sql.DB, fromID, toID int, amount float64) error {
    tx, err := db.Begin()
    if err != nil {
        return err
    }
    // Always rollback on function exit if tx is not committed
    defer func() {
        if p := recover(); p != nil {
            tx.Rollback()
            panic(p) // Re-throw panic after rollback
        } else if err != nil {
            tx.Rollback()
        }
    }()
    // Check source account balance
    var fromBalance float64
    err = tx.QueryRow(
        "SELECT balance FROM accounts WHERE id = $1 FOR UPDATE",
        fromID,
    ).Scan(&fromBalance)
    if err != nil {
        return fmt.Errorf("failed to get source balance: %v", err)
    }
}
```

```
    if fromBalance < amount {
        return fmt.Errorf("insufficient funds")
    } // Debit source account
    _, err = tx.Exec(
        "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
        amount, fromID,
    )
    if err != nil {
        return fmt.Errorf("failed to debit source: %v", err)
    } // Credit destination account
    _, err = tx.Exec(
        "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
        amount, toID,
    )
    if err != nil {
        return fmt.Errorf("failed to credit destination: %v", err)
    } // Log transaction
    _, err = tx.Exec(
        "INSERT INTO transactions (from_id, to_id, amount) VALUES ($1, $2, $3)",
        fromID, toID, amount,
    )
    if err != nil {
        return fmt.Errorf("failed to log transaction: %v", err)
    }
    return tx.Commit()
}
```

# Part III: Flutter local storage

*Flutter provides multiple options for local data persistence, from simple key-value storage to full-featured SQL databases.*

## Storage options comparison

Solution	Use Case	Complexity	Performance
<b>SharedPreferences</b>	Settings, simple data	Low	Fast
<b>sqflite</b>	Relational data, complex queries	Medium	Fast
<b>Hive</b>	Objects, NoSQL-style	Low	Very Fast
<b>Isar</b>	Advanced NoSQL with indexing	Medium	Very Fast
<b>Secure Storage</b>	Sensitive data	Low	Fast

# SharedPreferences: key-value storage

```
import 'package:shared_preferences/shared_preferences.dart';

class PreferencesService {
    static SharedPreferences? _prefs;

    static Future<void> init() async {
        _prefs = await SharedPreferences.getInstance();
    }

    // String operations
    static Future<void> setString(String key, String value) async {
        await _prefs?.setString(key, value);
    }

    static String? getString(String key) {
        return _prefs?.getString(key);
    }

    // Int operations
    static Future<void> setInt(String key, int value) async {
        await _prefs?.setInt(key, value);
    }

    static int? getInt(String key) {
        return _prefs?.getInt(key);
    }
}
```

```
// Bool operations
static Future<void> setBool(String key, bool value) async {
    await _prefs?.setBool(key, value);
}

static bool? getBool(String key) {
    return _prefs?.getBool(key);
}

// List operations
static Future<void> setStringList(String key, List<String> value) async {
    await _prefs?.setStringList(key, value);
}

static List<String>? getStringList(String key) {
    return _prefs?.getStringList(key);
}

// Remove and clear
static Future<void> remove(String key) async {
    await _prefs?.remove(key);
}

static Future<void> clear() async {
    await _prefs?.clear();
}
```

# SharedPreferences for app settings

```
class AppSettings {
    static const String _themeKey = 'theme_mode';
    static const String _languageKey = 'language';
    static const String _notificationsKey = 'notifications_enabled';
    static const String _userDataKey = 'user_data';

    // Theme management
    static Future<void> setThemeMode(ThemeMode mode) async {
        await PreferencesService.setString(_themeKey, mode.name);
    }

    static ThemeMode getThemeMode() {
        final themeName = PreferencesService.getString(_themeKey);
        return ThemeMode.values.firstWhere(
            (mode) => mode.name == themeName,
            orElse: () => ThemeMode.system,
        );
    }

    // Language setting
    static Future<void> setLanguage(String languageCode) async {
        await PreferencesService.setString(_languageKey, languageCode);
    }

    static String getLanguage() {
        return PreferencesService.getString(_languageKey) ?? 'en';
    }
}
```

```
// Notification settings
static Future<void> setNotificationsEnabled(bool enabled) async {
    await PreferencesService.setBool(_notificationsKey, enabled);
}

static bool getNotificationsEnabled() {
    return PreferencesService.getBool(_notificationsKey) ?? true;
}

// Complex object storage (JSON)
static Future<void> saveUserData(Map<String, dynamic> userData) async {
    final jsonString = jsonEncode(userData);
    await PreferencesService.setString(_userDataKey, jsonString);
}

static Map<String, dynamic>? getUserData() {
    final jsonString = PreferencesService.getString(_userDataKey);
    if (jsonString != null) {
        return jsonDecode(jsonString) as Map<String, dynamic>;
    }
    return null;
}

// Clear all settings
static Future<void> resetSettings() async {
    await PreferencesService.remove(_themeKey);
    await PreferencesService.remove(_languageKey);
    await PreferencesService.remove(_notificationsKey);
}
```

# SQLite with sqflite

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

class DatabaseService {
    static Database? _database;
    static const String _dbName = 'app_database.db';
    static const int _version = 1;
    static Future<Database> get database async {
        _database ??= await _initDatabase();
        return _database!;
    }
    static Future<Database> _initDatabase() async {
        final dbPath = await getDatabasesPath();
        final path = join(dbPath, _dbName);
        return await openDatabase(
            path,
            version: _version,
            onCreate: _onCreate,
            onUpgrade: _onUpgrade,
        );
    }
}
```

```
static Future<void> _onCreate(Database db, int version) async {
    await db.execute('''
        CREATE TABLE users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            email TEXT UNIQUE NOT NULL,
            created_at INTEGER NOT NULL
        )
    ''');
    await db.execute('''
        CREATE TABLE posts (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            title TEXT NOT NULL,
            content TEXT,
            created_at INTEGER NOT NULL,
            FOREIGN KEY (user_id) REFERENCES users (id)
        )
    ''');
    await db.execute('CREATE INDEX idx_posts_user_id ON posts(user_id)');
}
static Future<void> _onUpgrade(Database db, int oldVersion, int newVersion) async {
    // Handle database schema upgrades
    if (oldVersion < 2) {
        await db.execute('ALTER TABLE users ADD COLUMN avatar_url TEXT');
    }
}
```

# SQLite CRUD operations

```

class User {
    final int? id;
    final String name;
    final String email;
    final DateTime createdAt;
    User({this.id, required this.name, required this.email, required this.createdAt});
    Map<String, dynamic> toMap() {
        return {
            'id': id,
            'name': name,
            'email': email,
            'created_at': createdAt.millisecondsSinceEpoch,
        };
    }
    factory User.fromMap(Map<String, dynamic> map) {
        return User(
            id: map['id'],
            name: map['name'],
            email: map['email'],
            createdAt: DateTime.fromMillisecondsSinceEpoch(map['created_at']),
        );
    }
}
class UserRepository {
    // Create
    static Future<int> createUser(User user) async {
        final db = await DatabaseService.database;
        return await db.insert('users', user.toMap());
    }
}

```

```

// Read
static Future<List<User>> getAllUsers() async {
    final db = await DatabaseService.database;
    final List<Map<String, dynamic>> maps = await db.query('users');
    return List.generate(maps.length, (i) => User.fromMap(maps[i]));
}
static Future<User?> getUserById(int id) async {
    final db = await DatabaseService.database;
    final List<Map<String, dynamic>> maps = await db.query(
        'users',
        where: 'id = ?',
        whereArgs: [id],
    );
    return maps.isNotEmpty ? User.fromMap(maps.first) : null;
}
// Update
static Future<int> updateUser(User user) async {
    final db = await DatabaseService.database;
    return await db.update(
        'users',
        user.toMap(),
        where: 'id = ?',
        whereArgs: [user.id],
    );
}
// Delete
static Future<int> deleteUser(int id) async {
    final db = await DatabaseService.database;
    return await db.delete('users', where: 'id = ?', whereArgs: [id]);
}

```

# Hive: Fast NoSQL database

```

import 'package:hive/hive.dart';
// Simple data model for Hive
@HiveType(typeId: 0)
class Person {
  @HiveField(0)
  final String name;
  @HiveField(1)
  final int age;
  @HiveField(2)
  final List<String> hobbies;
  Person(this.name, this.age, this.hobbies);
} // Generated adapter (run: flutter packages pub run build_runner build)
class PersonAdapter extends TypeAdapter<Person> {
  @override
  final typeId = 0;
  @override
  Person read(BinaryReader reader) {
    return Person(
      reader.readString(),
      reader.readInt(),
      reader.readStringList(),);
  }
  @override
  void write(BinaryWriter writer, Person obj) {
    writer.writeString(obj.name);
    writer.writeInt(obj.age);
    writer.writeStringList(obj.hobbies);
  }
}

```

```

class HiveService {
  static Future<void> init() async {
    // Initialize Hive
    await Hive.initFlutter();
    // Register adapters
    Hive.registerAdapter(PersonAdapter());
    // Open boxes
    await Hive.openBox<Person>('persons');
    await Hive.openBox('settings');
  }
  // Person operations
  static Box<Person> get personBox => Hive.box<Person>('persons');
  static Box get settingsBox => Hive.box('settings');
  // CRUD operations
  static Future<void> addPerson(Person person) async {
    await personBox.add(person);
  }
  static List<Person> getAllPersons() {
    return personBox.values.toList();
  }
  static Future<void> updatePerson(int index, Person person) async {
    await personBox.putAt(index, person);
  }
  static Future<void> deletePerson(int index) async {
    await personBox.deleteAt(index);
  }
}

```

# Secure storage for sensitive data

```
import 'package:flutter_secure_storage/flutter_secure_storage.dart';
class SecureStorageService {
  static const _storage = FlutterSecureStorage(
    aOptions: AndroidOptions(
      encryptedSharedPreferences: true,
    ),
    iOptions: IOSOptions(
      accessibility: IOSEncryptor.accessibilityFirstUnlockThisDevice,
    ),
  );
  // Token management
  static Future<void> saveAuthToken(String token) async {
    await _storage.write(key: 'auth_token', value: token);
  }
  static Future<String?> getAuthToken() async {
    return await _storage.read(key: 'auth_token');
  }
  static Future<void> deleteAuthToken() async {
    await _storage.delete(key: 'auth_token');
  }
  // User credentials
  static Future<void> saveCredentials(String username, String password) async {
    await _storage.write(key: 'username', value: username);
    await _storage.write(key: 'password', value: password);
  }
}
```

```
static Future<Map<String, String?>> getCredentials() async {
  final username = await _storage.read(key: 'username');
  final password = await _storage.read(key: 'password');
  return {'username': username, 'password': password};
}
// Biometric settings
static Future<void> setBiometricEnabled(bool enabled) async {
  await _storage.write(key: 'biometric_enabled', value: enabled.toString());
}
static Future<bool> isBiometricEnabled() async {
  final value = await _storage.read(key: 'biometric_enabled');
  return value == 'true';
}
// Clear all secure data
static Future<void> clearAll() async {
  await _storage.deleteAll();
}
// Check if key exists
static Future<bool> containsKey(String key) async {
  return await _storage.containsKey(key: key);
}
// Get all keys
static Future<List<String>> getAllKeys() async {
  final all = await _storage.readAll();
  return all.keys.toList();
}}
```

# File system operations

```

import 'dart:io';
import 'package:path_provider/path_provider.dart';

class FileService {
    // Get application directories
    static Future<Directory> getAppDocumentsDirectory() async {
        return await getApplicationDocumentsDirectory();
    }
    static Future<Directory> getAppSupportDirectory() async {
        return await getApplicationSupportDirectory();
    }
    static Future<Directory> getTempDirectory() async {
        return await getTemporaryDirectory();
    }
    // File operations
    static Future<File> writeFile(String filename, String content) async {
        final directory = await getAppDocumentsDirectory();
        final file = File('${directory.path}/$filename');
        return await file.writeAsString(content);
    }
    static Future<String> readFile(String filename) async {
        final directory = await getAppDocumentsDirectory();
        final file = File('${directory.path}/$filename');
        return await file.readAsString();
    }
}

```

```

// JSON file operations
static Future<void> saveJsonFile(String filename, Map<String, dynamic> data) async {
    final content = jsonEncode(data);
    await writeFile(filename, content);
}
static Future<Map<String, dynamic>?> loadJsonFile(String filename) async {
    try {
        final content = await readFile(filename);
        return jsonDecode(content) as Map<String, dynamic>;
    } catch (e) {
        return null;
    }
}
// Image operations
static Future<File> saveImage(String filename, Uint8List imageBytes) async {
    final directory = await getAppDocumentsDirectory();
    final file = File('${directory.path}/$filename');
    return await file.writeAsBytes(imageBytes);
}
// Directory operations
static Future<List<FileSystemEntity>> listFiles() async {
    final directory = await getAppDocumentsDirectory();
    return directory.listSync();
}
static Future<void> deleteFile(String filename) async {
    final directory = await getAppDocumentsDirectory();
    final file = File('${directory.path}/$filename');
    if (await file.exists()) {
        await file.delete();
    }
}

```

# Part IV: Integration Patterns

*Effective data management requires seamless integration between server databases, APIs, and local storage.*

## Common integration patterns

- **Cache-first:** Check local storage before API calls
- **API-first:** Always fetch from server, cache locally
- **Offline-first:** Work offline by default, sync when online
- **Hybrid:** Intelligent switching based on connectivity

## Key considerations

- **Data consistency:** Handling conflicts between local and remote data
- **Performance:** Minimizing network requests and storage operations
- **Reliability:** Graceful degradation when offline
- **Security:** Protecting sensitive data in transit and at rest

# Cache-first pattern

```
class UserCacheService {
    static const String _usersCacheKey = 'users_cache';
    static const Duration _cacheExpiry = Duration(hours: 1);

    static Future<List<User>> getUsers({bool forceRefresh = false}) async {
        // Check cache first (unless force refresh)
        if (!forceRefresh) {
            final cachedUsers = await _getCachedUsers();
            if (cachedUsers != null) {
                return cachedUsers;
            }
        }
        try {
            // Fetch from API
            final users = await ApiService.fetchUsers();
            // Cache the results
            await _cacheUsers(users);
            return users;
        } catch (e) {
            // If API fails, try to return stale cache
            final cachedUsers = await _getCachedUsers(ignoreExpiry: true);
            if (cachedUsers != null) {
                return cachedUsers;
            }
            rethrow;
        }
    }
}
```

```
static Future<List<User>> _getCachedUsers({bool ignoreExpiry = false}) async {
    try {
        final cacheData = await FileService.loadJsonFile(_usersCacheKey);
        if (cacheData == null) return null;
        final timestamp = DateTime.parse(cacheData['timestamp']);
        final age = DateTime.now().difference(timestamp);
        if (!ignoreExpiry && age > _cacheExpiry) {
            return null; // Cache expired
        }
        final usersJson = cacheData['users'] as List;
        return usersJson.map((json) => User.fromJson(json)).toList();
    } catch (e) {
        return null;
    }
}
static Future<void> _cacheUsers(List<User> users) async {
    final cacheData = {
        'timestamp': DateTime.now().toIso8601String(),
        'users': users.map((user) => user.toJson()).toList(),
    };
    await FileService.saveJsonFile(_usersCacheKey, cacheData);
}
static Future<void> clearCache() async {
    await FileService.deleteFile(_usersCacheKey);
}}
```

# Offline-first pattern

```
class OfflineFirstService {
    static Future<void> createPost(Post post) async {
        await PostRepository.createPost(post); // Always save locally first
        try { // Try to sync with server
            final serverPost = await ApiService.createPost(post);
            // Update local post with server ID
            await PostRepository.updatePost(post.copyWith(
                serverId: serverPost.id,
                synced: true,
            ));
        } catch (e) {
            // Mark as pending sync
            await PostRepository.updatePost(post.copyWith(
                synced: false,
                pendingSync: true,
            ));}}
    static Future<List<Post>> getPosts() async {
        // Always return local data immediately
        final localPosts = await PostRepository.getAllPosts();
        // Try to sync in background
        _syncInBackground();
        return localPosts;
    }
}
```

```
static Future<void> _syncInBackground() async {
    try {
        // Get pending posts
        final pendingPosts = await PostRepository.getPendingPosts();
        for (final post in pendingPosts) {
            try {
                final serverPost = await ApiService.createPost(post);
                await PostRepository.updatePost(post.copyWith(
                    serverId: serverPost.id,
                    synced: true,
                    pendingSync: false,
                ));} catch (e) { // Keep as pending, try again later
                    continue; }
        }
        // Fetch updates from server
        final serverPosts = await ApiService.fetchPosts();
        await _mergeServerUpdates(serverPosts);
    } catch (e) {
        // Sync failed, will try again next time
    }
}
static Future<void> _mergeServerUpdates(List<Post> serverPosts) async {
    for (final serverPost in serverPosts) {
        final localPost = await PostRepository.getPostById(serverPost.id);
        if (localPost == null) {
            // New post from server
            await PostRepository.createPost(serverPost);
        } else if (serverPost.updatedAt.isAfter(localPost.updatedAt)) {
            // Server has newer version
            await PostRepository.updatePost(serverPost);
        }}}}
```

# Data synchronization strategies

```

class SyncService {
    static Future<SyncResult> syncData() async {
        final result = SyncResult();
        try {
            await _uploadPendingChanges(result); // 1. Upload pending local changes
            await _downloadUpdates(result); // 2. Download server updates
            await _resolveConflicts(result); // 3. Resolve conflicts
            result.success = true;
        } catch (e) {
            result.error = e.toString();
        }
        return result;
    }

    static Future<void> _uploadPendingChanges(SyncResult result) async {
        final pendingUsers = await UserRepository.getPendingUsers();
        for (final user in pendingUsers) {
            try {
                if (user.isNew) {
                    final serverUser = await ApiService.createUser(user);
                    await UserRepository.updateUser(user.copyWith(
                        serverId: serverUser.id,
                        synced: true,
                    )); result.uploaded++;
                } else if (user.isUpdated) {
                    await ApiService.updateUser(user);
                    await UserRepository.markSynced(user.id);
                    result.uploaded++;
                }
            } catch (e) {
                result.uploadErrors.add('User ${user.name}: $e');
            }
        }
    }
}

```

```

static Future<void> _downloadUpdates(SyncResult result) async {
    final lastSync = await PreferencesService.getString('last_sync');
    final serverUsers = await ApiService.fetchUsersModifiedSince(lastSync);
    for (final serverUser in serverUsers) {
        final localUser = await UserRepository.getUserById(serverUser.id);
        if (localUser == null) {
            await UserRepository.createUser(serverUser);
            result.downloaded++;
        } else if (serverUser.updatedAt.isAfter(localUser.updatedAt)) {
            if (localUser.hasLocalChanges) {
                // Conflict detected, queue for resolution
                await ConflictRepository.addConflict(localUser, serverUser);
                result.conflicts++;
            } else {
                await UserRepository.updateUser(serverUser);
                result.downloaded++;
            }
        }
    }
    await PreferencesService.setString('last_sync', DateTime.now().toIso8601String());
}

static Future<void> _resolveConflicts(SyncResult result) async {
    final conflicts = await ConflictRepository.getPendingConflicts();
    for (final conflict in conflicts) {
        // Simple last-write-wins strategy
        if (conflict.serverVersion.updatedAt.isAfter(conflict.localVersion.updatedAt)) {
            await UserRepository.updateUser(conflict.serverVersion);
            result.conflictsResolved++;
        } else {
            // Keep local version, will be uploaded next sync
        }
    }
    await ConflictRepository.resolveConflict(conflict.id);
}

```

# Performance optimization

## Go database optimization

```
func optimizeDatabase(db *sql.DB) { // Connection pooling
    db.SetMaxOpenConns(25)
    db.SetMaxIdleConns(5)
    db.SetConnMaxLifetime(5 * time.Minute)
    db.SetConnMaxIdleTime(2 * time.Minute)
}
// Prepared statements for repeated queries
type UserQueries struct {
    getUserId      *sql.Stmt
    getUserByEmail *sql.Stmt
    createUser     *sql.Stmt}
func prepareStatements(db *sql.DB) (*UserQueries, error) {
    getUserId, err := db.Prepare("SELECT id, name, email FROM users WHERE id = $1")
    if err != nil { return nil, err }
    getUserByEmail, err := db.Prepare("SELECT id, name, email FROM users WHERE email = $1")
    if err != nil { return nil, err }
    createUser, err := db.Prepare("INSERT INTO users (name, email) VALUES ($1, $2) RETURNING id")
    if err != nil { return nil, err }
    return &UserQueries{
        getUserId:      getUserId,
        getUserByEmail: getUserByEmail,
        createUser:     createUser,
    }, nil
}
```

## Flutter storage optimization

```
class OptimizedStorage {
    // Batch operations for better performance
    static Future<void> saveBatchUsers(List<User> users) async {
        final db = await DatabaseService.database;
        await db.transaction((txn) async {
            final batch = txn.batch();
            for (final user in users) {
                batch.insert('users', user.toMap());
            } await batch.commit(noResult: true);})
    }
    // Lazy loading for large datasets
    static Stream<List<User>> getUsersStream({int batchSize = 50}) async* {
        int offset = 0;
        while (true) {
            final users = await UserRepository.getUsersPaginated(
                offset: offset,
                limit: batchSize,);
            if (users.isEmpty) break;
            yield users;
            offset += batchSize; }
    }
    // Memory-efficient image handling
    static Future<String> saveImageOptimized(Uint8List imageBytes) async {
        final compressedBytes = await _compressImage(imageBytes); // Compress image before saving
        final filename = 'img_${DateTime.now().millisecondsSinceEpoch}.jpg';
        await FileService.saveImage(filename, compressedBytes);
        return filename; }}
```

# What we've learned

## Database Fundamentals

- **Database types:** SQL vs NoSQL trade-offs and use cases
- **ACID properties:** Ensuring data consistency and reliability
- **Design principles:** Normalization, indexing, and relationships
- **Evolution:** From hierarchical to modern distributed systems

## Go Database Programming

- **database/sql package:** Connection pooling, prepared statements
- **Migrations:** Version control for database schema changes
- **CRUD operations:** Safe and efficient data manipulation
- **Transactions:** Ensuring data integrity across operations

# What we've learned (continued)

## Flutter Local Storage

- **SharedPreferences**: Simple key-value storage for settings
- **SQLLite with sqflite**: Relational database for complex data
- **Hive**: High-performance NoSQL database for objects
- **Secure storage**: Encrypted storage for sensitive data
- **File system**: Direct file operations and image handling

## Integration Patterns

- **Cache strategies**: Cache-first, API-first, and hybrid approaches
- **Offline-first**: Building resilient apps that work without connectivity
- **Data synchronization**: Conflict resolution and real-time updates
- **Performance optimization**: Batch operations, lazy loading, compression

# Thank You!

## What's Next:

- Lab 04: Try yourself with databases connection and data manipulation

## Resources:

- Go database/sql: <https://pkg.go.dev/database/sql>
- SQLite guide: <https://www.sqlite.org/docs.html>
- Flutter sqflite: <https://pub.dev/packages/sqflite>
- Hive documentation: <https://docs.hivedb.dev/>
- Course Repository: <https://github.com/timur-harin/sum25-go-flutter-course>

## Contact:

- Email: [timur.harin@mail.com](mailto:timur.harin@mail.com)
- Telegram: @timur\_harin

**Next Lecture:** Authentication & Security

# Questions?