Qt Database Exam: Complete Solutions

Detailed Answers with Key Concepts

Introduction

This document provides comprehensive solutions to the Qt Database exam, with detailed explanations of key concepts. Each solution is presented with:

- Key terms in blue
- Important warnings in red
- Best practices in green
- Code examples with syntax highlighting

1 Solutions to Exercise 1: Database Fundamentals

1. **What is a database?**
   - A database is an organized collection of data
   - Examples: Banking systems, e-commerce sites, hospital records
   - Key concept: Databases provide persistent storage
2. **Qt module for database support**

```
QT += sql  # Add this to your .pro file

```

   SQL module provides all Qt database classes
3. **SQLite characteristics**

- Serverless - No separate server process
- Zero-configuration - No setup needed
- Single-file - Entire database in one file
- Warning: Not suitable for high-concurrency apps

4. **Database connection code**

```
// Create connection
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName("my_first_db.db");

// Open connection
if (!db.open()) {
    qDebug() << "Error:" << db.lastError().text();
    return;
}

// Close connection (when done)
db.close();

```

QSqlDatabase manages connections

5. **Connection lifecycle**

- Best practice: Open in application startup
- Close when no longer needed (app shutdown)
- Warning: Don't open/close repeatedly

2 Solutions to Exercise 2: Creating Tables

1. **Creating the Tasks table**

```
QSqlQuery query;
query.exec("CREATE TABLE Tasks ("
           "id INTEGER PRIMARY KEY, "
           "description TEXT, "
           "due_date TEXT, "
           "completed INTEGER)");  // SQLite uses 0/1 for
    boolean

```

SQLite types: INTEGER, TEXT, REAL, BLOB

2. **Query preparation methods**
   - Direct execution: Simple but vulnerable to SQL injection
   - Prepared statements: Safer with parameter binding
   - Parameter binding prevents SQL injection

3. **Adding tasks safely**

```
QSqlQuery query;
query.prepare("INSERT INTO Tasks (description, due_date,
    completed) "
                "VALUES (?, ?, ?)");
query.addBindValue("Learn Qt");
query.addBindValue("2023-12-01");
query.addBindValue(0);
query.exec();

```

4. **Counting incomplete tasks**

```
query.exec("SELECT COUNT(*) FROM Tasks WHERE completed =
    0");
if (query.next()) {
    int count = query.value(0).toInt();
}

```

   WHERE clause filters records

5. **QSqlQuery::lastError()**
   - Returns QSqlError object
   - Check after every database operation
   - Best practice: Always check for errors

3 Solutions to Exercise 3: Model-View Programming

1. **Basic application setup**

```
// In constructor
QTableView *view = new QTableView(this);
QPushButton *refreshBtn = new QPushButton("Refresh", this
    );

```

2. **Connecting model to view**

```
QSqlTableModel *model = new QSqlTableModel(this);
model->setTable("Tasks");
model->select();  // Load data
view->setModel(model);

```

QSqlTableModel bridges database and view

3. **Refresh implementation**

```
connect(refreshBtn, &QPushButton::clicked, [model](){
    model->select();  // Reloads data
});

```

4. **Filtering incomplete tasks**

```
connect(filterCheckbox, &QCheckBox::toggled, [model](bool
    checked){
    model->setFilter(checked ? "completed = 0" : "");
    model->select();
});

```

5. **Model vs direct queries**

- Model advantages:
  - Automatic view updates
  - Built-in editing capabilities
  - Easier data navigation
- Direct queries better for:
  - Complex operations
  - Bulk data processing

4 Solutions to Exercise 4: Task Manager

1. **UI elements**

   - QLineEdit for description
   - QDateEdit for due date
   - QPushButton for add/delete

2. **Adding tasks**

```
// Input validation
if (description.isEmpty()) {
    QMessageBox::warning(this, "Error", "Description cannot be empty");
    return;
}

// Insert new task
QSqlRecord record = model->record();
record.setValue("description", description);
record.setValue("due_date", dueDate.toString("yyyy-MM-dd"));
record.setValue("completed", 0);
model->insertRecord(-1, record);
model->submitAll();  // Save to database

```

3. **Marking tasks complete**

```
QModelIndex index = view->currentIndex();
model->setData(index.sibling(index.row(), 3), 1);  // Set completed
model->submitAll();

```

4. **Status bar updates**

```
QSqlQuery query("SELECT COUNT(*) FROM Tasks WHERE completed = 0");
if (query.next()) {
    statusBar()->showMessage(QString("%1 pending tasks").arg(query.value(0).toInt()));
}

```

5. **Editing existing tasks**
    - Use QDataWidgetMapper for form-based editing
    - Or enable editing directly in table view

5 Solutions to Exercise 5: Best Practices

1. **Connection cleanup**
    - Prevents resource leaks
    - Best practice: Use RAII (constructors/destructors)
2. **Transactions**

```
db.transaction();
try {
    // Multiple operations
    db.commit();
} catch (...) {
    db.rollback();
}

```

    ACID properties: Atomicity, Consistency, Isolation, Durability
3. **SQL injection prevention**
    - Always use prepared statements
    - Never concatenate user input into queries
4. **MVC architecture**
    - Database code belongs in model layer
    - Keep UI separate from data access
5. **Project structure**

Key Takeaways

- Database concepts: Tables, queries, models
- Qt SQL classes: QSqlDatabase, QSqlQuery, QSqlTableModel
- Security: Always use parameterized queries
- Architecture: Separate database logic from UI