<seed:think>
Got it, let's tackle creating a Qt6 tool for managing Invoices, Customers, and Articles using SQLite. First, I need to outline the core components:

1. **Database Schema**: Design tables for Customers, Articles, and Invoices (with a many-to-many relationship since an invoice can have multiple articles).
2. **C++ Backend**: Use QSql classes to interact with SQLite, handle CRUD operations for each entity.
3. **QML UI**: Create a responsive interface with forms for adding/editing entities and views for listing them (e.g., invoices list, customers table).
4. **Example Data**: Generate a sample SQLite database with test data for Customers, Articles, and Invoices.

1. Database Schema Design
We need three main tables: `Customers`, `Articles`, and `Invoices`. Since an invoice can have multiple articles (line items), we'll use a junction table `Invoice_Articles` to represent the many-to-many relationship between Invoices and Articles.

Tables:
- **Customers**: Stores customer information.
  - `id` (INTEGER PRIMARY KEY AUTOINCREMENT)
  - `name` (TEXT NOT NULL)
  - `email` (TEXT)
  - `phone` (TEXT)
  - `address` (TEXT)

- **Articles**: Stores product/service details.
  - `id` (INTEGER PRIMARY KEY AUTOINCREMENT)
  - `name` (TEXT NOT NULL)
  - `description` (TEXT)
  - `price` (REAL NOT NULL CHECK(price >= 0))
  - `stock_quantity` (INTEGER NOT NULL DEFAULT 0)

- **Invoices**: Stores invoice headers.
  - `id` (INTEGER PRIMARY KEY AUTOINCREMENT)
  - `customer_id` (INTEGER NOT NULL, FOREIGN KEY REFERENCES Customers(id))
  - `invoice_date` (DATE NOT NULL DEFAULT CURRENT_DATE)
  - `total_amount` (REAL NOT NULL CHECK(total_amount >= 0))
  - `status` (TEXT NOT NULL CHECK(status IN ('pending', 'paid', 'canceled'))) DEFAULT 'pending'

- **Invoice_Articles**: Junction table for invoice line items.
  - `invoice_id` (INTEGER NOT NULL, FOREIGN KEY REFERENCES Invoices(id))
  - `article_id` (INTEGER NOT NULL, FOREIGN KEY REFERENCES Articles(id))
  - `quantity` (INTEGER NOT NULL CHECK(quantity > 0))
  - `unit_price` (REAL NOT NULL CHECK(unit_price >= 0))
  - PRIMARY KEY (`invoice_id`, `article_id`)

2. Example SQLite Database

We'll create a sample database `company.db` with test data. Use SQLite Studio or run this SQL script to initialize it:

```sql
-- Create tables
CREATE TABLE IF NOT EXISTS Customers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT UNIQUE,
    phone TEXT,
    address TEXT
);

CREATE TABLE IF NOT EXISTS Articles (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    description TEXT,
    price REAL NOT NULL CHECK(price >= 0),
    stock_quantity INTEGER NOT NULL DEFAULT 0
);

CREATE TABLE IF NOT EXISTS Invoices (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    customer_id INTEGER NOT NULL,
    invoice_date DATE NOT NULL DEFAULT CURRENT_DATE,
    total_amount REAL NOT NULL CHECK(total_amount >= 0),
    status TEXT NOT NULL CHECK(status IN ('pending', 'paid', 'canceled')) DEFAULT
'pending',
    FOREIGN KEY (customer_id) REFERENCES Customers(id)
);

CREATE TABLE IF NOT EXISTS Invoice_Articles (
    invoice_id INTEGER NOT NULL,
    article_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL CHECK(quantity > 0),
    unit_price REAL NOT NULL CHECK(unit_price >= 0),
    PRIMARY KEY (invoice_id, article_id),
    FOREIGN KEY (invoice_id) REFERENCES Invoices(id),
    FOREIGN KEY (article_id) REFERENCES Articles(id)
);

-- Insert sample data
INSERT INTO Customers (name, email, phone, address) VALUES
('Alice Smith', 'alice@example.com', '+1-555-1234', '123 Main St, New York'),
('Bob Johnson', 'bob@example.com', '+1-555-5678', '456 Oak Ave, Boston');

INSERT INTO Articles (name, description, price, stock_quantity) VALUES
('Laptop', 'Dell XPS 13', 999.99, 10),
('Mouse', 'Logitech Wireless', 29.99, 50),
('Keyboard', 'Mechanical RGB', 79.99, 30);
```

```
INSERT INTO Invoices (customer_id, invoice_date, total_amount, status) VALUES
(1, '2024-05-01', 1059.98, 'paid'), -- Alice bought 1 Laptop + 1 Mouse
(2, '2024-05-02', 79.99, 'pending'); -- Bob bought 1 Keyboard

INSERT INTO Invoice_Articles (invoice_id, article_id, quantity, unit_price) VALUES
(1, 1, 1, 999.99),
(1, 2, 1, 29.99),
(2, 3, 1, 79.99);
```

3. C++ Backend (Model Classes)
Create model classes for each entity to encapsulate database operations. We'll use `QSqlTableModel` for simplicity (since it provides read/write access to a single table) or custom models for more control. For this example, `QSqlTableModel` is sufficient for listing/editing, but we'll need custom methods for invoice line items (many-to-many).

Step 1: Database Initialization
Create a `DatabaseManager` class to handle connecting to SQLite and initializing the schema if it doesn't exist.

```cpp
// database_manager.h
#ifndef DATABASEMANAGER_H
#define DATABASEMANAGER_H

#include <QObject>
#include <QSqlDatabase>
#include <QSqlError>
#include <QSqlQuery>

class DatabaseManager : public QObject {
    Q_OBJECT
public:
    explicit DatabaseManager(QObject *parent = nullptr);
    ~DatabaseManager();

    bool openConnection(const QString &dbPath = "company.db");
    void closeConnection();
    bool isConnected() const;

private:
    QSqlDatabase m_db;
};

#endif // DATABASEMANAGER_H
```

```cpp
// database_manager.cpp
```

```
#include "database_manager.h"
#include <QDebug>

DatabaseManager::DatabaseManager(QObject *parent) : QObject(parent) {}

DatabaseManager::~DatabaseManager() {
    closeConnection();
}

bool DatabaseManager::openConnection(const QString &dbPath) {
    if (m_db.isOpen()) return true;

    m_db = QSqlDatabase::addDatabase("QSQLITE");
    m_db.setDatabaseName(dbPath);

    if (!m_db.open()) {
        qCritical() << "Failed to connect to database:" << m_db.lastError().text();
        return false;
    }

    // Initialize schema (create tables if not exists)
    QSqlQuery query;
    QString createCustomers = R"(
        CREATE TABLE IF NOT EXISTS Customers (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            email TEXT UNIQUE,
            phone TEXT,
            address TEXT
        )
    )";
    if (!query.exec(createCustomers)) {
        qCritical() << "Failed to create Customers table:" << query.lastError().text();
        return false;
    }

    QString createArticles = R"(
        CREATE TABLE IF NOT EXISTS Articles (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            description TEXT,
            price REAL NOT NULL CHECK(price >= 0),
            stock_quantity INTEGER NOT NULL DEFAULT 0
        )
    )";
    if (!query.exec(createArticles)) {
        qCritical() << "Failed to create Articles table:" << query.lastError().text();
        return false;
    }

    QString createInvoices = R"(
```

```
        CREATE TABLE IF NOT EXISTS Invoices (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            customer_id INTEGER NOT NULL,
            invoice_date DATE NOT NULL DEFAULT CURRENT_DATE,
            total_amount REAL NOT NULL CHECK(total_amount >= 0),
            status TEXT NOT NULL CHECK(status IN ('pending', 'paid', 'canceled')) DEFAULT
'pending',
            FOREIGN KEY (customer_id) REFERENCES Customers(id)
        )
    )";
    if (!query.exec(createInvoices)) {
        qCritical() << "Failed to create Invoices table:" << query.lastError().text();
        return false;
    }

    QString createInvoiceArticles = R"(
        CREATE TABLE IF NOT EXISTS Invoice_Articles (
            invoice_id INTEGER NOT NULL,
            article_id INTEGER NOT NULL,
            quantity INTEGER NOT NULL CHECK(quantity > 0),
            unit_price REAL NOT NULL CHECK(unit_price >= 0),
            PRIMARY KEY (invoice_id, article_id),
            FOREIGN KEY (invoice_id) REFERENCES Invoices(id),
            FOREIGN KEY (article_id) REFERENCES Articles(id)
        )
    )";
    if (!query.exec(createInvoiceArticles)) {
        qCritical() << "Failed to create Invoice_Articles table:" << query.lastError().text();
        return false;
    }

    qInfo() << "Connected to database successfully.";
    return true;
}

void DatabaseManager::closeConnection() {
    if (m_db.isOpen()) {
        m_db.close();
        qInfo() << "Disconnected from database.";
    }
}

bool DatabaseManager::isConnected() const {
    return m_db.isOpen();
}
```

Step 2: Model Classes

Create `CustomerModel`, `ArticleModel`, and `InvoiceModel` using `QSqlTableModel` for basic CRUD operations. For invoices, we'll need to handle line items separately since it's a many-to-many relationship.

```cpp
// customer_model.h
#ifndef CUSTOMERMODEL_H
#define CUSTOMERMODEL_H

#include <QSqlTableModel>

class CustomerModel : public QSqlTableModel {
    Q_OBJECT
public:
    explicit CustomerModel(QObject *parent = nullptr, QSqlDatabase db = QSqlDatabase());

    // Optional: Custom roles for QML (if needed)
    enum Roles {
        IdRole = Qt::UserRole + 1,
        NameRole,
        EmailRole,
        PhoneRole,
        AddressRole
    };

    QHash<int, QByteArray> roleNames() const override;
    QVariant data(const QModelIndex &idx, int role) const override;
};

#endif // CUSTOMERMODEL_H
```

```cpp
// customer_model.cpp
#include "customer_model.h"

CustomerModel::CustomerModel(QObject *parent, QSqlDatabase db)
    : QSqlTableModel(parent, db) {
    setTable("Customers");
    setEditStrategy(QSqlTableModel::OnManualSubmit); // Manual submit for validation
    select(); // Load initial data
}

QHash<int, QByteArray> CustomerModel::roleNames() const {
    QHash<int, QByteArray> roles = QSqlTableModel::roleNames();
    roles[IdRole] = "id";
    roles[NameRole] = "name";
    roles[EmailRole] = "email";
    roles[PhoneRole] = "phone";
    roles[AddressRole] = "address";
    return roles;
```

```
}

QVariant CustomerModel::data(const QModelIndex &idx, int role) const {
    if (!idx.isValid()) return QVariant();

    int column = -1;
    switch (role) {
        case IdRole: column = 0; break;
        case NameRole: column = 1; break;
        case EmailRole: column = 2; break;
        case PhoneRole: column = 3; break;
        case AddressRole: column = 4; break;
        default: return QSqlTableModel::data(idx, role);
    }

    return QSqlTableModel::data(this->index(idx.row(), column), Qt::DisplayRole);
}
```

Repeat similar classes for `ArticleModel` (adjust fields: id, name, description, price, stock_quantity) and `InvoiceModel` (id, customer_id, invoice_date, total_amount, status). For `InvoiceModel`, you'll need to join with Customers to display the customer name instead of just the ID—use `setQuery()` with a JOIN:

```cpp
// invoice_model.cpp (partial)
InvoiceModel::InvoiceModel(QObject *parent, QSqlDatabase db)
    : QSqlTableModel(parent, db) {
    // Join Invoices with Customers to get customer name
    setQuery(R"(
        SELECT i.id, i.customer_id, c.name AS customer_name,
               i.invoice_date, i.total_amount, i.status
        FROM Invoices i
        JOIN Customers c ON i.customer_id = c.id
    )", db);
    setEditStrategy(QSqlTableModel::OnManualSubmit);
    select();
}

// RoleNames for InvoiceModel would include CustomerNameRole instead of customer_id
```

Step 3: Invoice Line Items Handling
Since invoices have multiple articles, create a `InvoiceLineItemModel` class to manage the `Invoice_Articles` junction table. This model will take an invoice ID and load only the line items for that invoice.

```cpp
// invoice_line_item_model.h
#ifndef INVOICELINEITEMMODEL_H
```

```
#define INVOICELINEITEMMODEL_H

#include <QSqlQueryModel>

class InvoiceLineItemModel : public QSqlQueryModel {
    Q_OBJECT
public:
    explicit InvoiceLineItemModel(QObject *parent = nullptr, QSqlDatabase db =
QSqlDatabase());

    void setInvoiceId(int invoiceId);
    bool addLineItem(int articleId, int quantity, double unitPrice);
    bool removeLineItem(int row);

    // Custom roles for line items (article name, quantity, unit price, total)
    enum Roles {
        ArticleIdRole = Qt::UserRole + 1,
        ArticleNameRole,
        QuantityRole,
        UnitPriceRole,
        TotalRole
    };

    QHash<int, QByteArray> roleNames() const override;
    QVariant data(const QModelIndex &idx, int role) const override;

private:
    QSqlDatabase m_db;
};

#endif // INVOICELINEITEMMODEL_H
```

```cpp
// invoice_line_item_model.cpp
#include "invoice_line_item_model.h"
#include <QDebug>

InvoiceLineItemModel::InvoiceLineItemModel(QObject *parent, QSqlDatabase db)
    : QSqlQueryModel(parent), m_db(db) {}

void InvoiceLineItemModel::setInvoiceId(int invoiceId) {
    setQuery(R"(
        SELECT ia.article_id, a.name AS article_name, ia.quantity, ia.unit_price,
               (ia.quantity * ia.unit_price) AS total
        FROM Invoice_Articles ia
        JOIN Articles a ON ia.article_id = a.id
        WHERE ia.invoice_id = ?
    )", m_db);
    setQueryParam(0, invoiceId); // Use QSqlQuery::bindValue if needed
    select();
```

```
}

bool InvoiceLineItemModel::addLineItem(int articleId, int quantity, double unitPrice) {
    QSqlQuery query(m_db);
    query.prepare(R"(
        INSERT INTO Invoice_Articles (invoice_id, article_id, quantity, unit_price)
        VALUES (:invoice_id, :article_id, :quantity, :unit_price)
    )");
    query.bindValue(":invoice_id", m_currentInvoiceId); // Need to track current invoice ID!
    query.bindValue(":article_id", articleId);
    query.bindValue(":quantity", quantity);
    query.bindValue(":unit_price", unitPrice);

    if (!query.exec()) {
        qCritical() << "Failed to add line item:" << query.lastError().text();
        return false;
    }
    setInvoiceId(m_currentInvoiceId); // Refresh model
    return true;
}

// Implement other methods (removeLineItem, roleNames, data) similarly
```


4. QML UI
Create a responsive UI with tabs for Customers, Articles, and Invoices. Use `ListView` or `TableView` to display models, and forms for adding/editing entities.

Main QML (main.qml)
```qml
import QtQuick 2.15
import QtQuick.Controls 2.15
import QtQuick.Layouts 1.15
import QtSql 6.0

ApplicationWindow {
    id: window
    visible: true
    width: 1200
    height: 800
    title: "Invoice & Customer Manager"

    // Initialize DatabaseManager (C++ backend)
    DatabaseManager {
        id: dbManager
        Component.onCompleted: openConnection()
    }

    // Models
    CustomerModel {
```

```
    id: customerModel
    database: dbManager.database // Assume DatabaseManager has a database property
  }

  ArticleModel {
    id: articleModel
    database: dbManager.database
  }

  InvoiceModel {
    id: invoiceModel
    database: dbManager.database
  }

  TabView {
    anchors.fill: parent
    tabBar: TabBar { implicitHeight: 40 }

    // Customers Tab
    Tab {
      title: "Customers"
      CustomerListView { model: customerModel }
      CustomerForm { model: customerModel }
    }

    // Articles Tab
    Tab {
      title: "Articles"
      ArticleListView { model: articleModel }
      ArticleForm { model: articleModel }
    }

    // Invoices Tab
    Tab {
      title: "Invoices"
      InvoiceListView { model: invoiceModel }
      InvoiceForm { model: invoiceModel; lineItemModel: InvoiceLineItemModel {} }
    }
  }
}
```

Customer List View (CustomerListView.qml)
```qml
import QtQuick 2.15
import QtQuick.Controls 2.15
import QtQuick.Layouts 1.15

Item {
  id: listViewContainer
```

```
    width: parent.width
    height: parent.height

    TableView {
        id: customerTable
        anchors.fill: parent
        model: model // Injected from parent (CustomerModel)
        selectionMode: TableView.SingleSelection

        TableViewColumn { role: "name"; title: "Name"; width: 200 }
        TableViewColumn { role: "email"; title: "Email"; width: 300 }
        TableViewColumn { role: "phone"; title: "Phone"; width: 200 }
        TableViewColumn { role: "address"; title: "Address"; width: 400 }
    }

    Button {
        text: "Delete Selected"
        onClicked: {
            var index = customerTable.currentIndex;
            if (index >= 0) model.removeRow(index);
        }
    }
}
```

Customer Form (CustomerForm.qml)
```qml
import QtQuick 2.15
import QtQuick.Controls 2.15
import QtQuick.Layouts 1.15

Item {
    id: formContainer
    width: parent.width
    height: 300

    ColumnLayout {
        anchors.centerIn: parent
        spacing: 10

        TextField {
            placeholderText: "Customer Name"
            focus: true
            Layout.fillWidth: true
        }

        TextField {
            placeholderText: "Email"
            Layout.fillWidth: true
        }
```

```
        TextField {
            placeholderText: "Phone"
            Layout.fillWidth: true
        }

        TextField {
            placeholderText: "Address"
            Layout.fillWidth: true
        }

        RowLayout {
            spacing: 10
            Button { text: "Save"; onClicked: model.submitAll() }
            Button { text: "Cancel"; onClicked: model.revertAll() }
        }
    }
}
```

5. Key Features
- **CRUD Operations**: Add, edit, delete Customers/Articles/Invoices via forms and tables.
- **Relationship Management**: Invoices can have multiple line items (articles) with quantity and price.
- **Data Validation**: Use SQLite constraints (e.g., `CHECK(price >= 0)`) and QML form validation.
- **Responsive UI**: Tabs for different modules, scrollable lists, and editable forms.

6. Testing the Tool
1. **Compile**: Build the project with Qt6 (ensure SQLite driver is included—Qt6 includes `QSQLITE` by default).
2. **Run**: The app will create `company.db` in the working directory if it doesn't exist, or use an existing one with sample data.
3. **Test Workflows**:
   - Add a new customer: Fill the form and click "Save".
   - Create an invoice: Select a customer, add articles from the list, set quantities/prices, and save.
   - View invoices: See the list with customer names, dates, and totals.

7. Extensions
- **PDF Invoicing**: Generate PDF invoices using `QtPdf` or a library like `wkhtmltopdf`.
- **Reporting**: Use `QChart` to display sales trends (e.g., monthly revenue).
- **User Authentication**: Add login functionality with encrypted passwords.
- **Cloud Sync**: Integrate with Firebase or AWS S3 to sync data across devices.

This example provides a solid foundation for an invoice/customer management tool using Qt6 and SQLite, following best practices for database integration and UI design in Qt.