

Bank System Client Server Project

Presented by Kerollos Samaan

Introduction

In this presentation, we will explore the Bank System Client-Server Project, which is designed to facilitate secure and efficient banking operations through a client-server architecture. The project leverages the power of Qt framework for building the user interface and managing network communications.

Key Features:

- User Management: The system allows users to create accounts, log in, and manage their profiles securely.
- Transaction Handling: Users can perform various transactions, including deposits, withdrawals, and transfers, with real-time updates.
- Data Security: The project implements encryption for sensitive data, ensuring secure communication between the client and server.
- Error Handling: Comprehensive error handling mechanisms are in place to provide feedback and maintain system integrity.

Architecture Overview:

- Server Side: The server handles incoming client requests, processes them using various request handlers, and communicates with the database to perform operations. The `ServerHandler` class manages individual client connections and request processing.
- Client Side: The client application is responsible for user interactions, sending requests to the server, and displaying responses. It utilizes the `User` class for managing connections and requests.

Command Design Pattern in Bank System Server

- Definition: The Command Design Pattern is a behavioral design pattern that encapsulates a request as an object, allowing for parameterization of clients with queues, requests, and operations.
- Purpose: It promotes loose coupling between the sender and receiver of a request, enhancing flexibility and scalability in software design.

Components of Command Design Pattern

- Command Interface
Defines the interface for executing a command.
- Concrete Command
Implements the command interface and binds a receiver to an action.
- Receiver
The object that performs the actual work when the command is executed.
- Invoker
Holds the command and invokes it.

Command Interface

```
./ RequestHandler.h
class RequestHandler {
public:
    virtual ~RequestHandler() = default;
    virtual void execute(const QStringList &RequestParts, QString
&statusMessage) = 0; // Command interface
};
```

Explanation: The RequestHandler interface defines the execute method, which all concrete commands must implement.

ConcreteCommand

```
// LoginHandler.h
class LoginHandler : public RequestHandler {
public:
    void execute(const QStringList &RequestParts, QString &statusMessage) override {
        // Logic for handling login
    }
};
```

```
// AddClientHandler.h
class AddClientHandler : public RequestHandler {
public:
    void execute(const QStringList &RequestParts, QString &statusMessage) override {
        // Logic for adding a client
    }
};
```

Explanation: Each handler (e.g., LoginHandler, AddClientHandler) implements the RequestHandler interface, encapsulating specific command logic.

Receiver

```
// BankDataBase.h
class BankDataBase : public QObject {
public:
    bool addClient(const QString &username, const QString &password, const QString
&fullName, const QString &age, const QString &email, const QString &balance);
    // Other database operations...
};
```

Explanation: The BankDataBase class acts as the receiver, performing operations like adding clients and managing account data.

Invoker

```
// ServerHandler.cpp
void ServerHandler::Operation(QString Request) {
    QStringList List = Request.split(":");
    if (requestHandlerMap.contains(List[0])) {
        requestHandlerMap[List[0]]->execute(List, statusMessage); // Invoking the command
        sendResponse(statusMessage);
    }
}
```

Explanation: The ServerHandler class invokes the appropriate command based on the request received from the client.

Advantages of Command Design Pattern

1-Decoupling

- **Benefit:** Reduces dependencies between sender and receiver.
- **Code Example:** The ServerHandler does not need to know the details of how each command works, only that it can call execute.

```
// ServerHandler.cpp  
requestHandlerMap[List[0]]->execute(List, statusMessage);  
// No knowledge of command internals
```

Advantages of Command Design Pattern

2- Parameterization

- **Benefit:** Commands can be parameterized and queued, enabling features like undo/redo.
- **Code Example:** Each command can accept different parameters through the execute method.

```
// LoginHandler.cpp
void LoginHandler::execute(const QStringList &RequestParts, QString
&statusMessage) {
    const QString userName = RequestParts[1]; // Parameterized input
}
```

Advantages of Command Design Pattern

3- Extensibility

- **Benefit:** New commands can be added without modifying existing code, adhering to the open/closed principle.
- **Code Example:** Adding a new command like UpdateAccountHandler requires minimal changes.

```
// ServerHandler.cpp
requestHandlerMap["UpdateAccount"] = newUpdateAccountHandler(dataBase);
// New command added
```

Advantages of Command Design Pattern

4- Encapsulation of All Information

- **Benefit:** Each command encapsulates all the information needed to perform an action, making it easier to manage and pass around.
- **Code Example:** The AddClientHandler encapsulates all necessary data for adding a client.

```
// AddClientHandler.cpp
void AddClientHandler::execute(const QStringList &RequestParts, QString &statusMessage) {
    const QString fullName = RequestParts[1];
    const QString userName = RequestParts[2];
    // Other parameters...
    DataBase.addClient(userName, password, fullName, ageStr, email, balanceStr); // Encapsulated data
}
```

SOLID Principles Compatibility

Single Responsibility Principle (SRP): Each command class has a single responsibility, handling one specific action.

```
// AddClientHandler.cpp
void AddClientHandler::execute(...) {
    // Only responsible for adding a client
}
```

Open/Closed Principle (OCP): The system is open for extension (new commands) but closed for modification (existing code remains unchanged).

```
// ServerHandler.cpp
requestHandlerMap["NewCommand"] = new NewCommandHandler(dataBase); //
Adding new command without modifying existing handlers
```

SOLID Principles Compatibility

Dependency Inversion Principle (DIP): High-level modules (e.g., ServerHandler) should not depend on low-level modules (e.g., AddClientHandler), but both should depend on abstractions (the RequestHandler interface).

```
// ServerHandler.h
class ServerHandler : public QThread {
    Q_OBJECT
public:
    explicit ServerHandler(qint32 ID, QObject *parent = nullptr);
    void Operation(QString Request); // High-level module

private:
    QMap<QString, RequestHandler*> requestHandlerMap; // Depends on abstraction
};

// AddClientHandler.h
class AddClientHandler : public RequestHandler {
public:
    void execute(const QStringList &RequestParts, QString &statusMessage) override {
        // Logic for adding a client
    }
};
```

Secure Request Sending in Banking System

User::SendRequest(QString Data)

Function Overview:

- Purpose: Sends a user request through a socket connection after validating the system environment and securing the data.

Key Steps:

1. Socket Validation:

- Check if the socket is open using `Socket.isOpen()`.

2. Environment Variable Check:

- Retrieve the `SECRET_KEY` from the system environment.
- Debug log the `secretKey` value.
- Exit the function if the `secretKey` is not set.

3. Generate Signature:

- Generate a digital signature using the `generateSignature` function with the request Data and `secretKey`.

4. Create JSON Request:

- Construct a JSON object (`requestObj`) containing:
 - `RequestSize`: Size of the request data.
 - `requestData`: The actual data being sent.
 - `Signature`: The generated signature.
- Convert the JSON object to a compact format using `QJsonDocument`.

5. Prepare Frame:

- Create a header containing the size of the JSON data followed by a colon.
- Combine the header and JSON data into a single frame.

6. Encrypt Data:

- Encrypt the frame using the `encryptRequest` function.

7. Send Data:

- Write the encrypted data to the socket using `Socket.write`.
- Emit a signal `UserSendRequest` with the hexadecimal representation of the encrypted data.

Secure Request Handling in Banking System

ServerHandler::onReadyRead()

Overview:

- Purpose: Handles incoming requests by reading, decrypting, validating, and processing the data received through a socket.

Key Steps:

1. Read and Log Data:

- Read all data from the socket (Socket->readAll()).
- Log the received encrypted request.

2. Decrypt Data:

- Decrypt the received data (decryptRequest(ByteArr)).
- Remove padding from decrypted data.

3. Extract and Validate Frame Size:

- Find the colon (:) separating the frame size.
- Validate the frame size.

4. Validate JSON Data:

- Extract and validate the JSON part of the frame.
- Parse the JSON data.

5. Validate Request Size:

- Verify the request size matches the actual data size.

6. Verify Signature:

- Retrieve SECRET_KEY from the environment.
- Verify the request signature.

7. Log and Process Valid Request:

- Log the valid request.
- Call Operation(requestData) to process the request.