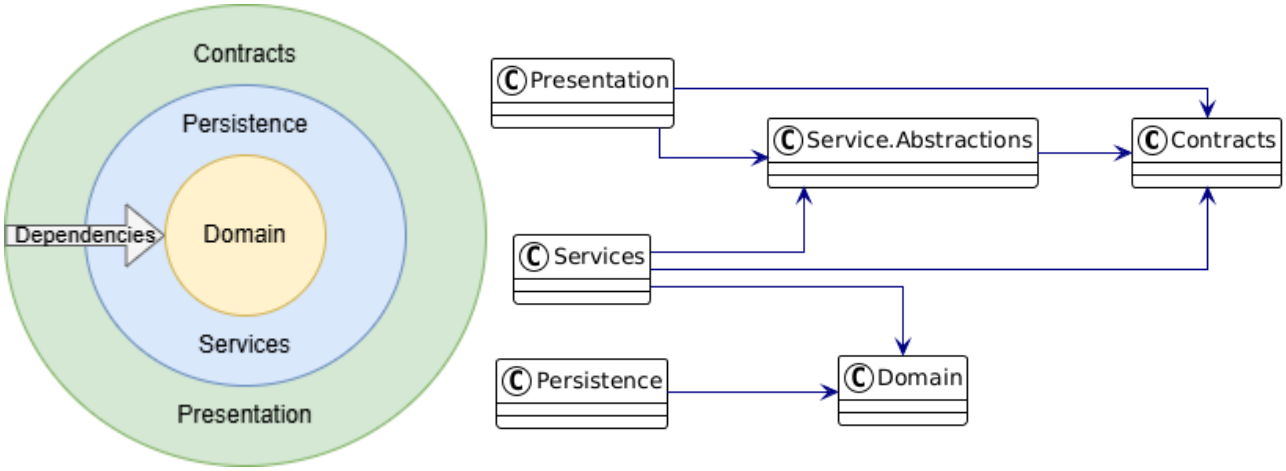
Simple Banking App

Inhaltsverzeichnis

Encapsulation and Abstraction

Onion Architecture

Die **Onion Architecture** ist ein Ansatz zur Softwareentwicklung, der darauf abzielt, eine flexible und wartbare Struktur für Anwendungen zu schaffen. Die Idee ist, dass die Anwendung aus mehreren Schichten besteht, die wie die Schalen einer Zwiebel angeordnet sind. Jede Schicht hat eine klare Aufgabe, und die Abhängigkeiten gehen immer nur von außen nach innen, nicht umgekehrt.



Domain

Die Domänenschicht befindet sich im Zentrum und enthält die Geschäftslogik, Entitäten und Regeln der Anwendung. Diese Schicht ist unabhängig von äußeren Details wie Benutzeroberfläche, Datenbank oder Frameworks. Dadurch hat sie maximale Isolation und wenige Abhängigkeiten. Die zentralen Aspekte der Domäne sind:

- Entities
- Repository-Interfaces
- Exceptions
- Entitylogic

Services

Die Serviceschicht ist in zwei Module unterteilt: Services.Abstractions und Services.

Im Module **Services.Abstractions** finden sich die Definitionen der Service-Interfaces, die die Hauptgeschäftslogik kapseln. Außerdem verwenden wir das Contracts-Module, um die Data Transfer Objects (DTOs) zu definieren, die wir mit den Service-Interfaces nutzen werden.

Damit definieren wir, was höheren Schichten der Onion-Architektur tun können und was nicht. Es ist jedoch wichtig, dass das **Service.Abstractions**-Module keine Referenz auf das **Domain**-Module hat.

Persistence

Hier werden alle Details zum Datenzugriff (z. B. Repository-Implementierungen) gekapselt. Diese Schicht kommuniziert mit der **Domäne**, bleibt aber von ihr abstrahiert.

Contracts

Kapselt Daten, die zwischen den Schichten übertragen werden, wie in Abschnitt Services beschrieben wurde.

Presentation

Der Zweck der Präsentationsschicht besteht darin, den Einstiegspunkt in unser System bereitzustellen, damit die Nutzer mit den Daten interagieren können.

Encapsulation

Sensible Daten wie Kontostände oder Benutzerinformationen werden in der Onion-Architektur durch klare Schichten geschützt. Die sensiblen Daten und die Logik zu deren Verarbeitung befinden sich im Kern der Architektur (Domain). Dieser Layer ist vollständig unabhängig von äußeren Schichten wie Datenbank, UI oder anderen Technologien. Damit ist er isoliert und nicht direkt von außen zugänglich. Der Zugriff auf sensible Daten erfolgt daher ausschließlich über klar definierte Schnittstellen, die in der Domain- oder Service-Schicht implementiert sind. Direkter Zugriff auf Datenbank-Entities wird vermieden.

Abstraction

Statt sich direkt auf konkrete Implementierungen zu verlassen, arbeitet die Onion-Architektur mit Interfaces und Abstraktionen. Änderungen in äußeren Schichten (z. B. ein Wechsel der Datenbank) haben keinen Einfluss auf die inneren Schichten.

Logging

Das Logging im Projekt wird durch ein eigenes Modul gekapselt und kann so in allen Schichten verwendet werden. Als Bibliothek kommt **Log4j** zum Einsatz. Diese Struktur ermöglicht eine klare Trennung der Zuständigkeiten und vereinfacht zukünftige Wartung oder den Wechsel auf eine andere Logging-Bibliothek.

Konfiguration

Die zentrale Konfiguration ist in einer XML-Datei hinterlegt. In diesem Beispiel wird Log4j so eingestellt, dass **nur noch** in die Datei application.log geschrieben wird und **nicht mehr** in die Konsole.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <File name="File" fileName="logs/application.log" append="true">
            <PatternLayout pattern="%d{ISO8601} [%t] %-5level %c - %msg%n"
/>
        </File>
    </Appenders>

    <Loggers>
        <Logger name="org.mandl" level="debug" additivity="false">
            <AppenderRef ref="File" />
        </Logger>
        <Logger name="org.hibernate" level="off" additivity="false" />
        <Logger name="org.jboss.weld.Bootstrap" level="off"
additivity="false" />

        <Root level="info">
            <AppenderRef ref="File" />
        </Root>
    </Loggers>
</Configuration>
```

- Appender: File schreibt in logs/application.log
- PatternLayout: Bestimmt das Format der Log-Ausgaben (Zeitstempel, Thread, Level, Klasse, Nachricht)
- Root-Logger-Level: Ist auf info gesetzt. Dadurch werden Meldungen unterhalb der info-Ebene im root-Logger nicht geschrieben.
- Spezielle Logger:
    - org.mandl auf debug: Detaillierte Informationen für die interne Projektstruktur.
    - org.hibernate, org.jboss.weld.Bootstrap auf off: Deaktiviert Logging für diese Frameworks, um die Log-Datei nicht unnötig zu füllen.

Anwendung

Ein beispielhafter Einsatz des Loggings erfolgt in der abstrakten Klasse BaseDomainService, von der alle Services erben:

```
public abstract class BaseDomainService {

    @Inject
    protected UserContext userContext;

    protected LoggingHandler logger;
```

```
    protected BaseDomainService() {
        logger = LoggingHandler.getLogger(this.getClass());
    }
}
```

```
logger.info("Registered user: " + username);
```

```
logger.error(message, serviceException);
```

Dadurch hat jede erbende Klasse Zugriff auf den Logger, ohne ihn extra implementieren zu müssen.

Error Handling

Im Projekt wird besonderes Augenmerk auf eine saubere Trennung zwischen Fachlogik (Service-Ebene) und Präsentationslogik (UI-Ebene) gelegt. Fehler, die im Service-Schicht auftreten, werden nicht direkt an die Präsentation weitergereicht, sondern zuerst im **ExceptionHandler** verarbeitet. Auf diese Weise können sowohl **passende Fehlermeldungen** für den Endanwender bereitgestellt als auch **kritische Informationen** für Entwickler im Log festgehalten werden.

Implementierung

Trennung von Service und Präsentation

- Die **Service-Ebene** wirft Exceptions, die auf fachlicher Ebene entstehen (z. B. ServiceException).
- Die **Präsentationsebene** (etwa eine Android-App oder eine JavaFX-GUI) erhält von der Service-Schicht im Regelfall nur **Rückgabecodes** bzw. **bekannte Exceptions**, keine internen Fehlermeldungen.

Exception-Handling-Konzept

- **Bekannte Exceptions** (z. B. InvalidInputException oder andere valide Fehlerzustände) führen zu einer Fehlermeldung im UI, ohne diese in den Logs zu überfrachten.
- **Unerwartete Exceptions** (z. B. NullPointerException, Datenbankfehler) werden detailliert geloggt und dem Benutzer wird eine allgemeine Fehlermeldung angezeigt, ohne technische Details zu leaken.

Zentrale Steuerung über ExceptionHandler

- **Entscheidung über Logging:** Je nach Exception-Typ (bekannt vs. unbekannt) wird die Log-Ausgabe gesteuert.

- **Exception an Benutzer anzeigen:** Freundliche, nicht-technische Meldung für erwartete Fehler, zum Beispiel “Invalid Input” oder “Benutzername/Passwort falsch”.
- **Loggen von kritischen Fehlern:** Falls eine Ursache (Cause) vorhanden ist oder der Fehler unerwartet ist, wird auf **error**- oder **fatal**-Level geloggt.

```
case ServiceException serviceException -> {
    var message = serviceException.getMessage();
    if (serviceException.getCause() != null) {
        logger.error(message, serviceException);
    }

    MessageHandler.printExceptionMessage(message);
}
default -> {
    var message = e.getMessage();
    logger.fatal(message, e);
    MessageHandler.printExceptionMessage("Unexpected error occurred.");
}
```

Durch diese Implementierung erhoffe ich mir **klare Trennung**, **zentrale Steuerung** und **gezieltes Logging.**

Anwendung

Der kritischer code innerhalb eines Try Catch Block befinden. Wenn die Authentifizierung fehlschlägt, fängt der ExceptionHandler die Exception.

```
private void startLogin() {
        MessageHandler.printHeader(LOGIN);
        printPrompt("Enter Username: ");
        var username = lastInput;
        printPrompt("Enter Password: ");
        var password = lastInput;
    try {
        var user = serviceManager.getAuthenticationService().loginUser(username, password);
        var navigationController = ControllerFactory.getNavigationController(user, serviceManager);
        navigationController.start();
    } catch (Exception ex) {
        ExceptionHandler.handleException(ex);
    }
}
```

Im nachfolgenden Beispiel wird die Authentifizierung in einem Try-Catch-Block eingebettet. Scheitert der Login-Vorgang, fängt der **ExceptionHandler** die Exception zentral ab.

Validation

Die Validierung in diesem Projekt kontextabhängig – je nach Verantwortungsbereich. Einige Entitäten validieren sich selbst, während bestimmte Eingaben (z. B. Passwörter) in dedizierten Services geprüft werden.

- Entitäten sind selbst für die Validierung zuständig.
- Spezifische Logik wird in dafür zuständigen Services gekapselt.
- Doppelte Validierung ist besonders bei verteilter Architektur (z. B. Server und Client) sinnvoll, in diesem Projekt jedoch nicht zwingend erforderlich.

Implementierung

Jedes Entity übernimmt die Verantwortung für ihre eigenen Daten. So kann ein Objekt niemals in einen ungültigen Zustand gelangen. Beispiel das **IdentityUser** Objekt prüft schon bei der Zuweisung des Usernames, ob dieser gültig ist.

```
public void setUsername(String username) {
    validateUsername(username);
    this.username = username;
}

public void validateUsername(String username) {
    if (username == null) {
        throw new IllegalArgumentException("Username cannot be null.");
    }

    var usernameRegex = "^[a-zA-Z0-9]{3,20}$";

    if (!username.matches(usernameRegex)) {
        throw new IllegalArgumentException(
                "Invalid username! Username must be 3-20 characters long and alphanumeric only."
        );
    }
}
```

Nicht jede Validierung ist sinnvoll im Entity selbst. Manche Daten können nicht mehr im Entity validiert werdenm weil die Daten bei der Übergabe bereits transformiert/verschlüsselt sind oder die Möglichkeit unerwünscht ist den Zugriff in der Präsentation Ebene zu ermöglichen (z.B. UserDto sollte keinen Zugriff auf Passwort haben). **Passwort-Validierung** erfolgt vor der Verschlüsselung im **PasswordService**.

```
public static void validatePassword(String password) {
    if (password.isEmpty()) {
        throw new IllegalArgumentException("Password cannot be empty");
    }

    String passwordRegex = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z\\d@$!%*?&]{8,}$";

    if (!password.matches(passwordRegex)) {
        throw new IllegalArgumentException(
```

```
            "Password must have at least 8 characters, including one uppercase letter, one lowercase letter, one
number, and one special character."
        );
    }
}
```

Validierung sollte prinzipiell bei einer Server-Client Aufbau immer zweimal validiert werden. Eine Clientseitige für Schnelle Rückmeldung an den Nutzer und eine endgültige Validierung, um Manipulation auszuschließen. In diesem Projekt ist ein extern laufender Server nicht vorgesehen, daher passiert doppelte Validierung nur, wo es wirklich sinnvoll ist (z.B. kritische oder häufige Eingaben w.z.B. Transaktionsbetrag).

```
private BigDecimal getValidatedAmount(String promptMessage, boolean isPositive) {
    while (true) {
        printPrompt(promptMessage);
        try {
            String input = lastInput.replace(",", ".");
            BigDecimal amount = new BigDecimal(input);

            if ((isPositive && amount.compareTo(BigDecimal.ZERO) <= 0) ||
                    (!isPositive && amount.compareTo(BigDecimal.ZERO) >= 0)) {
                throw new IllegalArgumentException("Amount must be " + (isPositive ? "positive." : "negative."));
            }

            return amount;
        } catch (NumberFormatException | ArithmeticException e) {
            MessageHandler.printExceptionMessage("Invalid input. Please enter a valid number.");
        } catch (IllegalArgumentException e) {
            MessageHandler.printExceptionMessage(e.getMessage());
        }
    }
}
```

Die Datenbank-Absicherung wird von **Prepared Statements** geschützt.

Cryptography

Die im Projekt eingesetzte **Verschlüsselungslösung** nutzt einen **JPA-Converter** (Jakarta) zum **Encrypten** und **Decrypten** sensibler Daten direkt beim Speichern und Laden von **Entity-Properties**. Dadurch können Daten in der Datenbank **verschlüsselt** abgelegt werden, ohne dass separate Schritte für die Verschlüsselung nötig sind.

Implementierung

JPA-Converter wurd durch die **Annotaiton @Convert** deklariert und somit auf das Entity-Feld angewendet, die verschlüsselt in der Datenbank liegen soll.

```
@Column(nullable = false, unique = true)
@Convert(converter = StringEncryptionConverter.class)
private String accountNumber;
```

Das Framework übernimmt automatisch das Aufrufen von **convertToDatabaseColumn()** vor dem Speichern und **converToEntityAttribute()** beim Laden.

Encrypt und Decrypt

Konfiguration

- Es wird ein Cipher-Objekt mit dem angegebenen Verschlüsselungsalgorithmus **(ENCRYPTION_ALGORITHM)** angelegt.
- Ein Initialisierungsvektor (IV) mit fester Länge **IV_LENGTH** wird durch ein Byte-Array erzeugt.
- Mithilfe dieses IV und der definierten Tag-Länge (**GCM_TAG_LENGTH**) wird ein **GCMParameterSpec** erzeugt, um den **GCM-Modus** zu konfigurieren.

Verschlüsselung

- Das Cipher-Objekt wird auf **ENCRYPT_MODE** gesetzt und mit dem geheimen Schlüssel (**SECRET_KEY_SPEC**) sowie den **GCM-Parametern** initialisiert.
- Anschließend werden die Byte-Daten des Strings (**attribute.getBytes()**) per **cipher.doFinal()** verschlüsselt.
- Abschließend werden die Byte-Daten in **Base64**-kodiert,

```
@Override
public String convertToDatabaseColumn(String attribute) {
    if (attribute == null) return null;

    try {
        var cipher = Cipher.getInstance(ENCRYPTION_ALGORITHM);
        var iv = new byte[IV_LENGTH];
        var spec = new GCMParameterSpec(GCM_TAG_LENGTH, iv);
        cipher.init(Cipher.ENCRYPT_MODE, SECRET_KEY_SPEC, spec);

        var encryptedBytes = cipher.doFinal(attribute.getBytes());
        return Base64.getEncoder().encodeToString(encryptedBytes);
    } catch (Exception e) {
        throw new RuntimeException("Error during encryption", e);
    }
}
```

Entschlüsselung

- Das Cipher-Objekt wird auf **DECRYPT_MODE** gesetzt und mit dem geheimen Schlüssel (**SECRET_KEY_SPEC**) sowie den **GCM-Parametern** initialisiert.
- Der zuvor **Base64**-kodierte String (dbData) wird nun zunächst dekodiert (**Base64.getDecoder().decode(...)**), sodass wieder die eigentlichen verschlüsselten Byte-Daten vorliegen.
- Diese Byte-Daten werden mit **cipher.doFinal(...)** entschlüsselt, was die ursprünglichen Klartext-Daten ergibt.

```java
@Override
public String convertToEntityAttribute(String dbData) {
    if (dbData == null) return null;

    try {
        var cipher = Cipher.getInstance(ENCRYPTION_ALGORITHM);
        var iv = new byte[IV_LENGTH];
        var spec = new GCMParameterSpec(GCM_TAG_LENGTH, iv);
        cipher.init(Cipher.DECRYPT_MODE, SECRET_KEY_SPEC, spec);

        var decodedBytes = Base64.getDecoder().decode(dbData);
        return new String(cipher.doFinal(decodedBytes));
    } catch (Exception e) {
        throw new RuntimeException("Error during decryption", e);
    }
}
```