

# Entwurfsdokument für das Projekt SecureDataOutsourcing

Maik Wild  
Michael Fechner  
Fabian Scheytt  
Julian Hinrichs

Betreuer :  
Alexander Degitz  
Till Neudecker

10. Dezember 2016

# Inhaltsverzeichnis

<b>1</b>	<b>Grobentwurf</b>	<b>4</b>
1.1	Allgemeiner Überblick . . . . .	4
1.2	Architektur . . . . .	4
1.3	MVC- MVVM - Vergleich . . . . .	4
1.4	Aktivitätsdiagramm : SQL - Anfrage . . . . .	6
<b>2</b>	<b>Feinentwurf</b>	<b>8</b>
2.1	Klassenbeschreibung . . . . .	8
2.1.1	Shared Code . . . . .	8
2.1.1.1	Message . . . . .	9
2.1.1.2	<<Enumeration>> MessageType . . . . .	10
2.1.1.3	CallMessage extends Message . . . . .	11
2.1.1.4	<<Enumeration>> Call . . . . .	12
2.1.1.5	ErrorMessage extends Message . . . . .	13
2.1.1.6	ResultMessage extends Message . . . . .	14
2.1.1.7	SQLMessage extends Message . . . . .	15
2.1.1.8	SQLQuery . . . . .	15
2.1.1.9	CommandType . . . . .	16
2.1.1.10	Configuration . . . . .	17
2.1.1.11	ConfigurationFactory . . . . .	17
2.1.1.12	Logger . . . . .	18
2.1.1.13	LoggerFactory . . . . .	19
2.1.1.14	ICloudController : Interface . . . . .	20
2.1.1.15	IMediator : Interface . . . . .	21
2.1.1.16	ErrorHandler . . . . .	21
2.1.1.17	FileTransferException extends Exception . .	22
2.1.1.18	InternalServerException extends Exception .	23
2.1.1.19	UnexpectedConnectionInterruptException ex- tends Exception . . . . .	23
2.1.1.20	ServerUneachableException extends Excepti- on . . . . .	24
2.1.2	Connect Service . . . . .	24
2.1.2.1	ConnectService . . . . .	26
2.1.2.2	ServerMediator : IMediator . . . . .	27
2.1.2.3	ServerCloudController : IMediator . . . . .	28
2.1.2.4	Controller . . . . .	30
2.1.2.5	SocketListener . . . . .	30
2.1.2.6	StoreAlgorithm : Interface . . . . .	32

2.1.3	Client	33
2.1.3.1	MainView	33
2.1.3.2	SQLView	34
2.1.3.3	CloudTabView	34
2.1.3.4	AdminAuthView	35
2.1.3.5	AdminView	35
2.1.3.6	LogView	36
2.1.3.7	TestView	36
2.1.3.8	OptionsView	37
2.1.3.9	MainViewModel : INotifyPropertyChanged	37
2.1.3.10	SQLViewModel	38
2.1.3.11	CloudTabViewModel	39
2.1.3.12	AdminAuthViewModel	41
2.1.3.13	AdminViewModel	41
2.1.3.14	LogViewModel	42
2.1.3.15	TestViewModel	43
2.1.3.16	OptionsViewModel	43
2.1.3.17	ClientMediator : IMediator	45
2.1.3.18	ServerConnection	47
<b>3</b>	<b>Sequenzdiagramme</b>	<b>48</b>
3.1	Anwendungsfall: Administrator ändert Default Zugangsdaten	48
3.2	Anwendungsfall: Ausführen einer SQL-Anfrage	49
3.3	Anwendungsfall: Benutzer speichert Datei im Cloud Tab	50

# **1 Grobentwurf**

## **1.1 Allgemeiner Überblick**

Dieses Dokument erläutert den Entwurf des Projekts SecureDataOutsourcing genauer.

Das zu entwickelnde Produkt besteht aus zwei Anwendungen, der Clientanwendung und dem serverseitigen Mediator Dienst. Dieser teilt sich wiederum in einen Connect Service, der die Verbindung zwischen Clientanwendung und Server herstellt und verwaltet, und Encrypt Service, der die Datenbankoperationen ausführt und diese sowie die Datenbanken selbst verschlüsselt. Außerdem kommuniziert der Encrypt Service mit einem externen Cloudspeicher, auf dem die genutzten Datenbanks gelagert werden.

Wir werden in diesem Dokument zuerst auf die grobe Architektur der Software eingehen, danach in größerem Detail, aufgeteilt in die drei Bereiche gemeinsamer Code, Code der Serveranwendung und Code der Clientanwendung, die verwendeten Klassen und Entwurfsmuster beschreiben und schließlich anhand von drei Sequenzdiagrammen beispielhaft die wichtigsten Abläufe erklären.

## **1.2 Architektur**

In diesem Abschnitt gehen wir auf die Softwarearchitektur ein. Die Clientseite benutzt das Model-View-ViewModel (MVVM) Architekturnuster. Dieses wird im nächsten Abschnitt genauer erläutert und dem für vergleichbare Projekte zumeist verwendeten Model-View-Controller(MVC) Muster gegenübergestellt.

## **1.3 MVC- MVVM - Vergleich**

Nun werden wir die Gemeinsamkeiten und Unterschiede zwischen den beiden Entwurfsmustern MVC und MVVM genauer beleuchten.

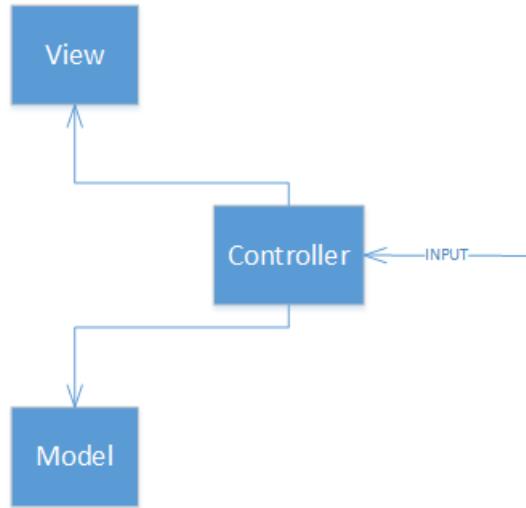


Abbildung 1: Typischer MVC Aufbau

Der Unterschied offenbart sich, sobald man die Assoziationen betrachtet. Bei MVC steuert der Controller sowohl View als auch Model. Das Besondere an MVVM ist, dass das ViewModel nicht unsere View steuert, sondern lediglich Daten für diese bereitstellt. Das ganze funktioniert über eine Technik, die sich Data Binding nennt. Dadurch ist es möglich, eine sehr lose Kopplung zwischen den Schichten zu bekommen, d.h. hier sind nun sowohl Model und View als auch ViewModel leicht austauschbar.

Darüber hinaus ist es durch MVVM möglich, per Unit Test die GUI zu testen, da die Darstellungslogik unabhängig von der Anzeige instantiierbar und ausführbar ist. Dadurch wird zusätzlich die gleichzeitige Arbeit an mehreren Komponenten durch verschiedene Entwickler, vor allem die Trennung von Design- und (Business-)Logikentwicklung, unterstützt. So können Designer durch die Nutzung von Expression Blend Beispieldaten erzeugen, während die Entwickler ungestört an ViewModel und Model arbeiten.

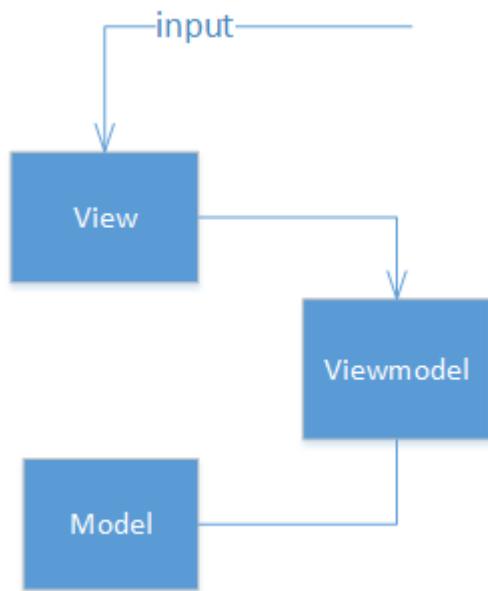


Abbildung 2: Typischer MVVM Aufbau

## 1.4 Aktivitätsdiagramm : SQL - Anfrage

In diesem Abschnitt erläutern wir anhand eines Aktivitätsdiagramms typische Abläufe des Projekts. Zuerst verbindet sich der Benutzer mit dem Mediator. Falls dieser Vorgang erfolgreich ist, sendet der Mediator seine Capabilities. Danach beginnt der normale Programmfluss. Der Benutzer kann nun SQL-Anfrage senden. Diese wird auf Gültigkeit überprüft, verschlüsselt und danach an den Provider gesendet. Dessen Antwort wird entschlüsselt, verarbeitet und angezeigt.

Das Aktivitätsdiagramm beschreibt den typischen Kontrollfluss des Programms. Zudem enthält es auch noch den Kontrollfluss falls eine Ausnahme auftritt.

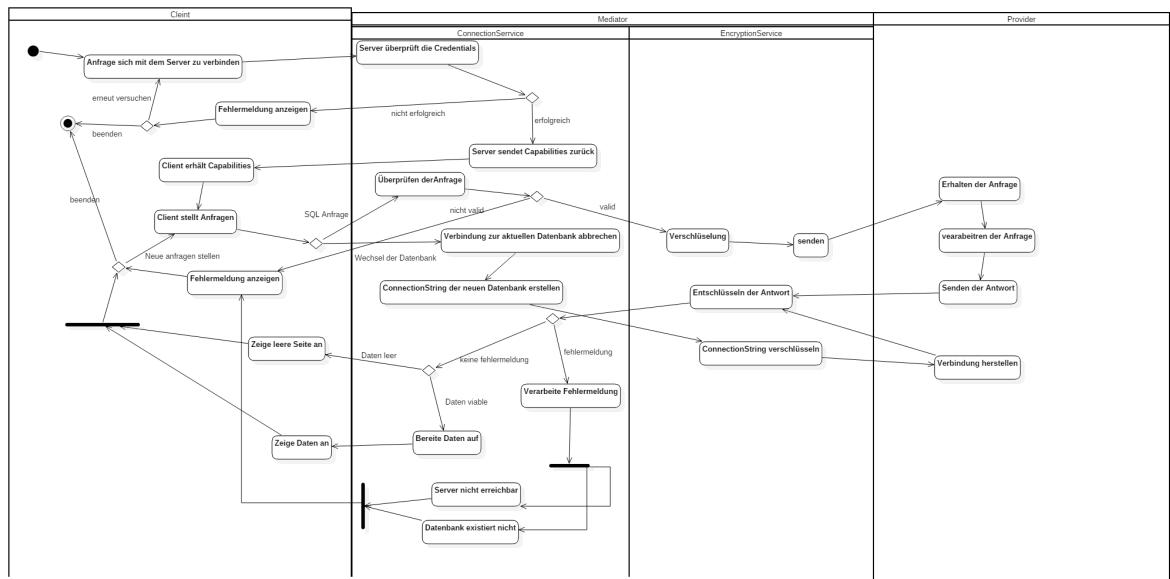


Abbildung 3: Aktivitätsdiagramm für typische Abläufe

## 2 Feinentwurf

Der Feinentwurf dient der detaillierten Definition des Projekts und dazu das Projekt auf Klassenebene zu beschreiben. Dadurch vermittelt der Feinentwurf ein spezifisches Bild des internen Aufbaus bezüglich der Kommunikation zwischen den Klassen. Zudem werden in diesem Abschnitt die verwendeten Entwurfsmuster erläutert und deren Einsatz im Projekt verdeutlicht.

Serverseitig und in dem Code der von beiden Seiten genutzt wird kommt das Strategie Entwurfsmuster zum Einsatz, zum Einen bei den schon vorhandenen StorageAlgorithm Klassen und zum Anderen bei den von uns zu implementierenden Message Klassen. Dies gewährleistet einerseits die einfache Austauschbarkeit der Algorithmen und andererseits den vereinfachten Umgang mit verschiedenen Arten von Nachrichten.

### 2.1 Klassenbeschreibung

In diesem Abschnitt gehen wir detailliert auf die verwendeten Klassen und deren Funktion im Gesamtprodukt ein. Das zu entwickelnde Projekt teilt sich grob in drei Komponenten auf.

1. Shared Code : Klassen, die sowohl von der Clientanwendung als auch vom ConnectService genutzt werden.
2. Client : Die Anwendung beim Benutzer.
3. Connect Service : Modul das die Verbindung zwischen Provider und Client herstellt.

Im Folgenden stellen wir jede Komponente vor und beschreiben ihre Aufgaben, Klassen und Methoden kurz. Aus Gründen der Übersichtlichkeit wurde auf Utility-Methoden (Getter, Setter, etc) verzichtet. Zu jeder Klasse gibt es eine kurze Erläuterung, eventuell noch kurz ihre Rolle in den verwendeten Entwurfsmustern.

#### 2.1.1 Shared Code

In diesem Abschnitt werden die Klassen beschrieben, die sowohl in der Clientanwendung als auch im serverseitigen Dienst verwendet werden.

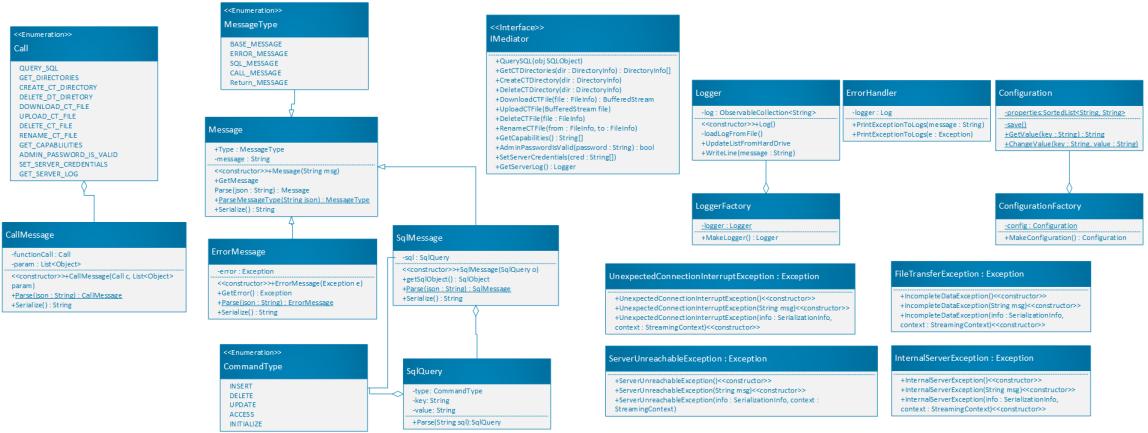


Abbildung 4: Klassendiagramm des gemeinsam genutzten Codes

### 2.1.1.1 Message



Die Basis Klasse für alle Nachrichten, die zwischen der Client und Server übertragen werden. Sie dient nur als Container der 4 Subklassen und sollte nicht als eigenständiges Objekt instanziiert werden. Nachrichten (Messages) sind im allgemeinen Träger von Informationen. In diesem konkreten Fall beinhalten die Messages, je nach Typ bestimmte Objekte oder Funktionsaufrufe. Dazu werden Messages mit der entsprechenden Information (Siehe Properties der Subklassen) instanziiert, serialisiert, übertragen, beim Empfänger geparst und interpretiert.

#### Attributes

1. public Type : MessageType  
Die Art der Nachricht. Siehe MessageType.

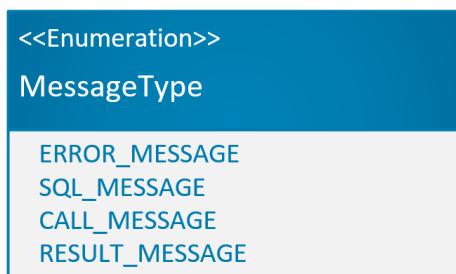
#### Methods

1. <<constructor>> protected Message()  
Konstrutor der Message. Die Sichtbarkeit ist protected, damit

Message nicht alleinstehend instanziert werden kann, die Sub-Klassen welche Message beerben, welche mit Message zusammen einen alleinigen Namespace bilden, allerdings schon.

2. public static ParseMessageType(json : String) : MessageType  
Parst die Art der Nachricht aus dem ersten Block eines serialisierten Message Objektes. Dies ist notwenig, um zu entscheiden welcher Nachrichten Typ instanziert werden soll.

#### 2.1.1.2 <<Enumeration>> MessageType

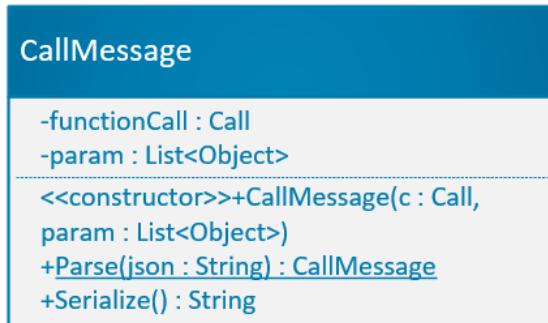


Eine Enumeration über alle Nachrichten Typen. Diese werden benötigt, um zu entscheiden, welche Nachricht zu parsen ist.

#### Attributes

1. CALL\_MESSAGE  
Simuliert Methodenaufruf
2. ERROR\_MESSAGE  
Enthält einen Error, der auf dem Server aufgetreten ist
3. SQL\_MESSAGE  
Enthält ein auszuführendes SQLQuery Objekt
4. RESULT\_MESSAGE  
Enthält die Serverantwort auf einen Call

### 2.1.1.3 CallMessage extends Message



Diese Klasse wird genutzt, um Remote-Function-Calls zu emulieren. Dazu wird ein CallMessage Objekt mit entsprechendem Typ und Parameterliste instanziert, serialisiert, zum Server geschickt, dort geparsst und entsprechend dem Typ, die entsprechende Methode aufgerufen.

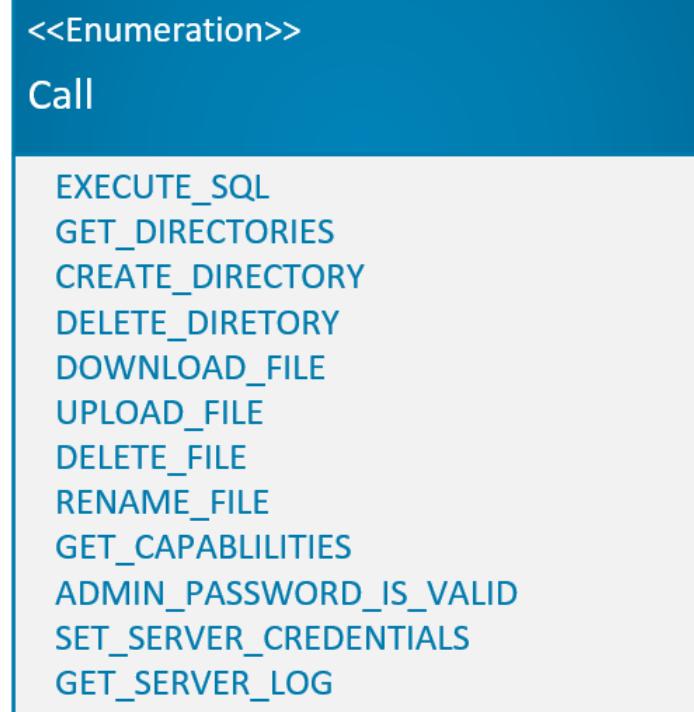
#### Attributes

1. private functionCall : Call  
Der in der Nachricht enthaltene Methodenaufruf.
2. private param : List<Object>  
Die in der Nachricht enthaltenen Methodenargumente.

#### Methods

1. <<constructor>> public CallMessage(c : Call, param : List <Object>)  
: CallMessage  
Instanziiert die CallMessage
2. public static Parse(json : String) : CallMessage  
Parst einen serialisierten CallMessage-String zu einem CallMessage-Objekt.
3. public Serialize() : String  
Speichert den Zustand der Nachricht in einem String und gibt diesen zurück. Aus dem String kann durch das aufrufen von CallMessage.Parse wieder ein CallMessage-Objekt rekonstruiert werden.

#### 2.1.1.4 <<Enumeration>> Call



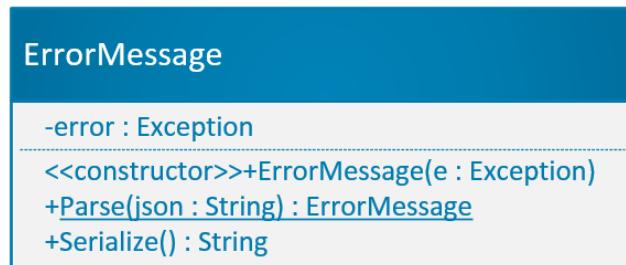
Eine Enumeration über die verschiedenen unterstützten Methodenfernaufrufe des ICloudController und IMediator. Sie bildet den Typ der CallMessage. Für nähere Information zu den Calls siehe IMediator bzw. ICloudController.

#### Attributes

1. EXECUTE\_SQL  
Führe SQL-Anfrage aus
2. GET\_DIRECTORIES  
Hole eine Liste aller Unterverzeichnisse und Dateien des aktuellen CloudTab Verzeichnisses
3. CREATE\_DIRECTORY  
Erstelle CloudTab Verzeichnis
4. DELETE\_DIRECTORY  
Lösche CloudTab Ordner
5. DOWNLOAD\_FILE  
Lade CloudTab Datei herunter

6. UPLOAD\_FILE  
Lade Datei in Cloud hoch
7. DELETE\_FILE  
Lösche eine CloudTab Datei
8. RENAME\_FILE  
Benenne eine CloudTab Datei um
9. GET\_CAPABILITIES  
Hole eine Liste unterstützter SQL-Befehle des Servers
10. ADMIN\_PASSWORD\_IS\_VALID  
Überprüfe die Richtigkeit eines eingegebenen Administratorpassworts
11. SET\_SERVER\_CREDENTIALS  
Ändere die Verbindungsdetails (Hostname, Benutzername, Passwort) im Mediator des verwendeten Providers
12. GET\_SERVER\_LOG  
Hole den Server Log
13. INITIALIZE  
Setzt den Server in einen vordefinierten Zustand um gleiche Ausgangsbedingungen für Test sicherzustellen. Der Zustand ist hinter der Connect Service Schnittstelle versteckt.

#### 2.1.1.5 ErrorMessage extends Message



Diese Klasse dient als Container zur übertragen von Exceptions

#### Attributes

1. private Exception exception

#### Methods

1. <<constructor>> public ErrorMessage(c : Call, param : List <Object>)  
Instanziert die ErrorMessage
2. public static Parse(json : String) : ErrorMessage  
Parst einen serialisierten ErrorMessage-String zu einem ErrorMessage-Objekt.
3. public Serialize() : String  
Parst serialisierte Nachricht zu ErrorMessage Objekt. Speichert den Zustand der Nachricht in einem String und gibt diesen zurück. Aus dem String kann durch das aufrufen von ErrorMessage.Parse wieder ein ErrorMessage-Objekt rekonstruiert werden.

#### 2.1.1.6 ResultMessage extends Message



Diese Klasse dient als Container aller return-Objekte der IMediator und ICloudController

#### Attributes

1. private result Object

#### Methods

1. <<constructor>> public ResultMessage(o : Object)  
Instanziert die ResultMessage
2. public static Parse(json : String) : ResultMessage  
Parst einen serialisierten ResultMessage-String zu einem ResultMessage-Objekt.
3. public Serialize() : String  
Parst serialisierte Nachricht zu ResultMessage Objekt. Speichert den Zustand der Nachricht in einem String und gibt diesen zurück. Aus dem String kann durch das aufrufen von ResultMessage.Parse wieder ein ResultMessage-Objekt rekonstruiert werden.

### 2.1.1.7 SQLMessage extends Message



Diese Klasse dient als Container zur Übertragung von SQLQuery-Objekten

#### Attributes

1. private SQLQuery sql

#### Methods

1. `<<constructor>> public SQLMessage(sql : SqlQuery)`  
Instanziert die SQLMessage
2. `public static Parse(json : String) : SQLMessage`  
Parst einen serialisierten SQLMessage-String zu einem ErrorMessage-Objekt.
3. `public Serialize() : String`  
Parst serialisierte Nachricht zu SQLMessage Objekt. Speichert den Zustand der Nachricht in einem String und gibt diesen zurück.  
Aus dem String kann durch das aufrufen von SQLMessage.Parse wieder ein SQLMessage-Objekt rekonstruiert werden.

### 2.1.1.8 SQLQuery



Diese Klasse ist für SQL-Befehle zuständig. Sie parst eingehende Befehle und erzeugt ein SQLQuery Objekt, in dem die geparsten Eigenschaften gespeichert werden.

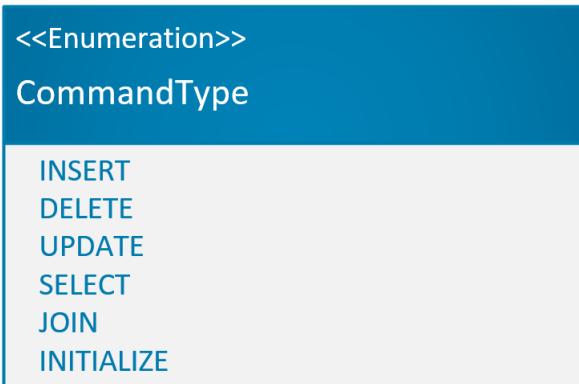
### Attributes

1. private key : String  
Ziel des SQL-Befehls.
2. private value : String  
Inhalt des SQL-Befehls. Kann null sein.
3. private type : CommandType  
Die Art des SQL-Befehls.

### Methods

1. public static Parse(sqlCommand : String) : SQLQuery  
Parst einen String in ein SQLQuery Objekt, falls dieser die syntaktischen Voraussetzungen erfüllt, falls nicht wird eine ParseException ausgelöst.

#### 2.1.1.9 CommandType

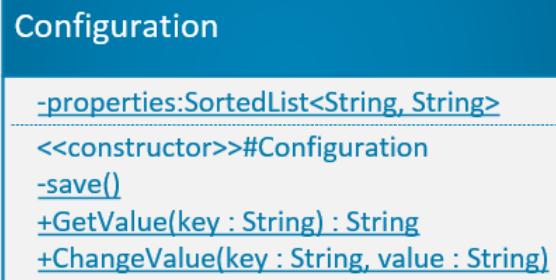


Der Typ eines auszuführenden SQLQuery Objekts

### Attributes

1. INSERT
2. DELETE
3. UPDATE
4. SELECT
5. JOIN

### 2.1.1.10 Configuration



Diese Klasse ist für die Verwaltung der Einstellungen zuständig.

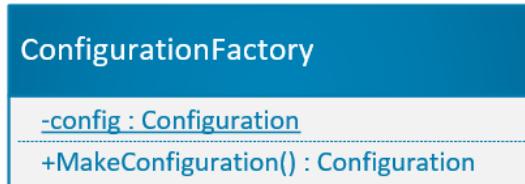
#### Attributes

1. private static properties : SortedList<String, String>  
Eine Liste von Einstellungen

#### Methods

1. <<constructor>> protected Configuration()  
Instanziert ein Configuration-Objekt. Der Konstruktor ist protected, damit nur die ConfigurationFactory, welche ein alleinigen Namespace mit Configuration bildet ihn instanziieren kann.
2. private static save()  
Speichert die gesetzten Einstellungen
3. public static GetValue(key : String) : String  
Stellt einen Einstellungswert bereit
4. public static ChangeValue(key : String, value : String)  
Ändert eine Einstellung

### 2.1.1.11 ConfigurationFactory



Die ConfigurationFactory dient als Werkzeug zur Erstellung von Configurationinstanzen. Zweck der Factory ist es, dass nur ein Configuration Objekt instanziert werden kann. Dazu wird beim ersten instanziieren einer Configurationsinstanz, diese in einer privaten statischen property gespeicher, damit sie bei späteren make-aufrufen, ebenfalls zurückgegeben werden kann, statt eine neue Instanz zu erstellen

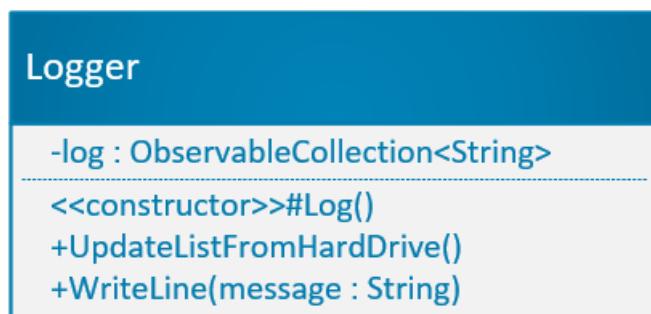
### Attributes

1. private static config : Configuration  
Die benutzte Konfigurationsinstanz

### Methods

1. public MakeConfiguration() : Configuration  
Falls noch keine Configurationsinstanz existiert wird eine neue Konfigurationsinstanz erstellt

### 2.1.1.12 Logger



Diese Klasse ist für Erstellung und Verwaltung der Logdateien zuständig.

### Attributes

1. private log : ObservableCollection<String>  
Die Logeinträge.

### Methods

1. <<constructor>> protected Logger()  
Instanziert ein Logger-Objekt. Der Konstrutor ist protected, damit nur die LoggerFactory, welche ein alleinigen Namespace mit Logger bildet ihn instanziieren kann.

2. private loadLogFile()  
Lädt alte Logeinträge von einem festgelegten Speicherplatz. Der Speicherort kann in der Konfiguration geändert werden.
3. public UpdateListFromHardDrive()  
Aktualisiert die Logs.
4. public WriteLine(String message)  
Fügt dem Log einen neuen Eintrag hinzu. Dieser wird sowohl in die log property geschrieben als auch auf die Festplatte.

### 2.1.1.13 LoggerFactory



Die LoggerFactory dient als Werkzeug zur Erstellung von Loggerinstanzen. Zweck der Factory ist es, dass nur ein Logger Objekt instanziert werden kann. Dazu wird beim ersten instanziieren einer Loggersinstanz, diese in einer privaten statischen property gespeicher, damit sie bei späteren make aufrufen, ebenfalls zurückgegeben werden kann, statt eine neue Instanz zu erstellen

#### Attributes

1. private static logger : Logger  
Die benutzte Logger Instanz

#### Methods

1. public MakeLogger() : Logger  
Falls noch keine existiert wird eine neue Loggerinstanz erstellt

### 2.1.1.14 ICloudController : Interface

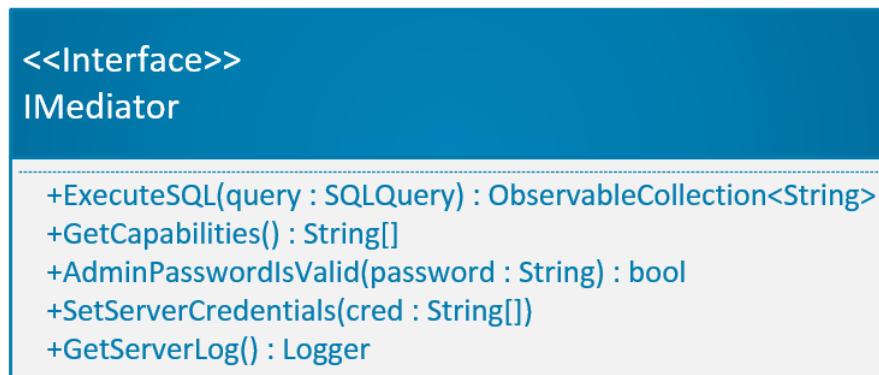


Gemeinsames Interface, das auf Client und Server in unterschiedlichen Implementierungen zum Einsatz kommt. Es dient als Repräsentant des Clouddatei Speichers des Servers. Ziel ist es, dass die ViewModel der Netzwerkkommunikation ignorant gegenüber sind. Sie agieren mit dem ICloudController, als wäre er Lokal verfügbar.

#### Methods

1. CreateDirectory(dir : DirectoryInfo)  
Erstellt ein Verzeichnis beim Pfad dir
2. DeleteDirectory(dir : DirectoryInfo)  
Löscht Verzeichnis dir
3. DeleteFile(file : FileInfo)  
Löscht Datei file
4. DownloadFile(file : FileInfo) : BufferedStream  
Gibt eine BufferedStream zurück, welcher auf file verweist
5. GetDirectories(dir : DirectoryInfo) : DirectoryInfo[]  
Gibt eine Liste mit allen Dateien und Verzeichnissen im Verzeichnis dir zurück
6. RenameFile(file : FileInfo)  
Nennt from um in to
7. UploadFile(path : FileInfo, bufferedStream : BufferedStream)  
Speichert die Datei, auf die bufferedStream zeigt an die Stelle path im Cloudverzeichnis

### 2.1.1.15 IMediator : Interface



Gemeinsames Interface, das auf Client und Server in unterschiedlichen Implementierungen zum Einsatz kommt. Es dient als allgemeiner Repräsentant des Servers. Ziel ist es, dass die ViewModel der Netzwerkkommunikation ignorant gegenüber sind. Sie agieren mit dem IMediator, als wäre er Lokal verfügbar.

#### Methods

1. public ExecuteSQLQuery(command: String)  
Führt eine SQL Anfrage aus.
2. public GetCapabilities() : String[]  
Fragt die vom Server unterstützten SQL-Befehle an.
3. public AdminPasswordIsValid(credentials : String) : boolean  
Sendet vom User eingegebene Zugangsdaten zur Authentifizierung als Administrator an den Server.
4. public SetServerCredentials(cred : String[])  
Sendet neue Default Zugangsdaten an den Server.
5. public GetServerLog() : Logger  
Ruft die Logs vom Server ab.

### 2.1.1.16 ErrorHandler



Diese Klasse kümmert sich um alle Exceptions, die während der Laufzeit auftreten.

## Attributes

1. private logger : Logger  
Der Log in dem die Fehlermeldungen festgehalten werden.

## Methods

1. public PrintExceptionToLogs(e : Exception)  
Hält Exceptions als Logeinträge fest.

### 2.1.1.17 FileTransferException extends Exception



Diese Exception wird geworfen, falls beim Übertragen der Daten über das CloudTab Teile fehlen oder fehlerhaft ankommen.

## Methods

1. <<constructor>> public FileTransferException()
2. <<constructor>> public FileTransferException(String msg)
3. <<constructor>> public FileTransferException(info : SerializationInfo, context : StreamingContext)<<constructor>>

### **2.1.1.18 InternalServerErrorException extends Exception**

#### **InternalServerErrorException**

```
<<constructor>>+InternalServerErrorException()
<<constructor>>+InternalServerErrorException(msg : String)
<<constructor>>+InternalServerErrorException(info :
SerializationInfo, context : StreamingContext)
```

Diese Exception wird geworfen, falls ein Fehler innerhalb des Servers auftritt.

#### **Methods**

1. <<constructor>> public InternalServerErrorException()
2. <<constructor>> public InternalServerErrorException(String msg)
3. <<constructor>>public InternalServerErrorException(info : SerializationInfo, context : StreamingContext)

### **2.1.1.19 UnexpectedConnectionInterruptException extends Exception**

#### **UnexpectedConnectionInterruptException**

```
<<constructor>>+UnexpectedConnectionInterruptException()
<<constructor>>+UnexpectedConnectionInterruptException(msg : String)
<<constructor>>+UnexpectedConnectionInterruptException(info : SerializationInfo,
context : StreamingContext)
```

Diese Exception wird geworfen, falls die Verbindung zum ConnectService unerwartet unterbrochen wird.

#### **Methods**

1. <<constructor>> public UnexpectedConnectionInterruptException()

2. <<constructor>> public UnexpectedConnectionInterruptException(String msg)
3. <<constructor>> public UnexpectedConnectionInterruptException(info : SerializationInfo, context : StreamingContext)

### 2.1.1.20 ServerUnreachableException extends Exception

#### ServerUnreachableException

```
<<constructor>>+ServerUnreachableException()
<<constructor>>+ServerUnreachableException(msg : String)
<<constructor>>+ServerUnreachableException(info :
SerializationInfo, context : StreamingContext)
```

Diese Exception wird geworfen, falls die Verbindung zum ConnectService nicht aufgebaut werden konnte.

#### Methods

1. <<constructor>> public ServerUnreachableException()
2. <<constructor>> public ServerUnreachableException(String msg)
3. <<constructor>> public ServerUnreachableException(info : SerializationInfo, context : StreamingContext)

### 2.1.2 Connect Service

Der Connect Service beschreibt die Serverseite des Projekts. Er führt die Anweisungen des Clients mithilfe des Encrypt Service auf der externen Datenbank aus und ist die einzige Schnittstelle mit dem der Client kommuniziert.

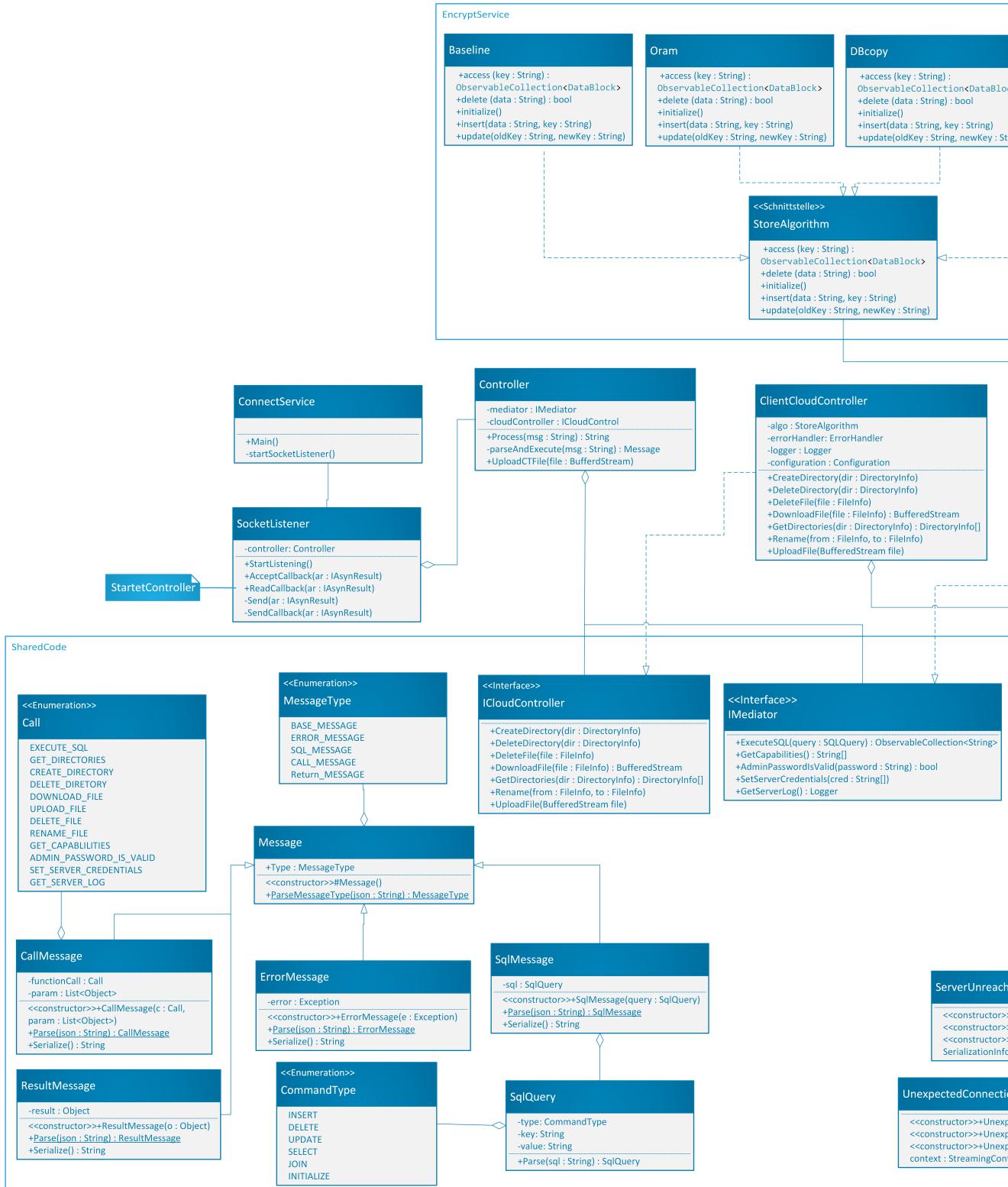
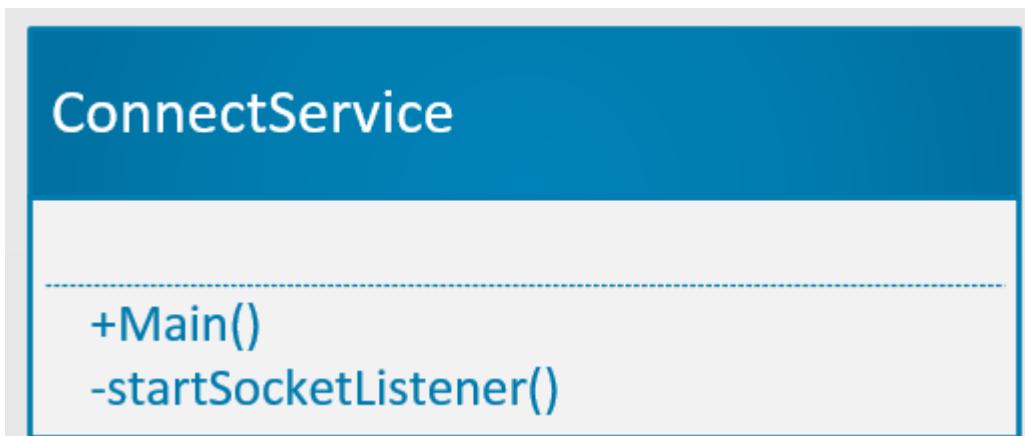


Abbildung 5: Klassendiagramm der serverseitigen Anwendung

### 2.1.2.1 ConnectService



Die ConnectService Klasse ist der Einsprungpunkt für die serverseitige Anwendung. Sie ist verantwortlich für den Start der gesamte Server- und Verbindungslogik.

#### Attributes

#### Methods

1. public void Main()  
Main Methode, die den serverseitigen Dienst startet.
2. private void startSocketListener()  
Wird von der Main Methode aufgerufen und startet den Socket Listener, um Clients den Verbindungsaufbau zu ermöglichen.

### 2.1.2.2 ServerMediator : IMediator

## ServerMediator

-algo : StoreAlgorithm

-logger : Logger

-errorHandler: ErrorHandler

-configuration : Configuration

+ExecuteSQL(query : SQLQuery) : ObservableCollection<S

+GetCapabilities() : String[]

+AdminPasswordIsValid(password : String) : bool

+SetServerCredentials(cred : String[])

+GetServerLog() : Logger

Implementiert das IMediator Interface.

### Attributes

1. private algo : StoreAlgorithm

Der Algorithmus, welcher auf dem derzeit verbundenen Datenbank Server eingesetzt wird.

2. private logger : Logger

Eine Logger Instanz mit dem alle Ereignisse vermerkt werden.

3. private errorHandler : ErrorHandler

Eine Error Logger Instanz mit dem alle Fehlermeldungen vermerkt werden.

4. private configuration : Configuration

Eine Configuration Instanz, welche die Einstellungen des Servers beinhaltet.

## Methods

1. public QuerySQL(obj : SQLObject)  
Erhält eine SQL-Anfrage als SQLObject und erstellt damit eine Datenbankanfrage, die anschließend über den Encrypt Service ausgeführt wird. Zurückgegeben wird das Ergebnis des Encrypt Service in einer Tabellen ähnlichen Form mit einem ObservableCollection Objekt.
2. public GetCapabilities() : String[]  
Gibt alle vom Connect Service unterstützten Befehle als String Array zurück.
3. public AdminPasswordIsValid(password : String) : boolean  
Überprüft die übergebenen Zugangsdaten und gibt je nach Ergebnis „true“ für korrekte und „false“ für abgelehnte Zugangsdaten zurück.
4. public SetServerCredentials(cred : String[])  
Übernimmt die übergebenen Zugangsdaten als neue Standard Zugangsdaten.
5. public GetServerLog() : Logger  
Gibt den Inhalt der Server Log Dateien zurück.

### 2.1.2.3 ServerCloudController : IMediator



Implementiert das ICloudController Interface.

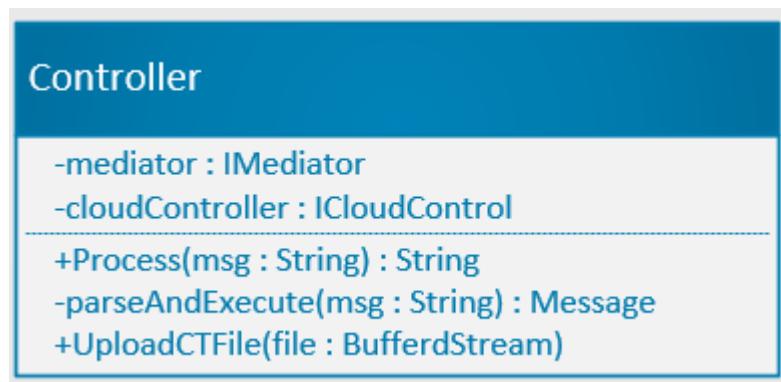
## Attributes

1. private algo : StoreAlgorithm  
Der Algorithmus, welcher auf dem derzeit verbundenen Datenbank Server eingesetzt wird.
2. private logger : Logger  
Eine Logger Instanz mit dem alle Ereignisse vermerkt werden.
3. private errorHandler : ErrorHandler  
Eine Error Logger Instanz mit dem alle Fehlermeldungen vermerkt werden.
4. private configuration : Configuration  
Eine Configuration Instanz, welche die Einstellungen des Servers beinhaltet.

## Methods

1. public CreateDirectory (dir : DirectoryInfo)  
Erstellt ein neues Verzeichnis im Cloud Tab auf dem Datenbankserver.
2. public DeleteDirectory (dir : DirectoryInfo)  
Löscht ein gewünschtes Verzeichnis aus dem Cloud Tab auf dem Datenbankserver.
3. public DeleteFile (file : FileInfo)  
Löscht die bestimmte Datei aus dem Cloud Tab.
4. public DownloadFile (file : FileInfo) : BufferedStream  
Lädt die gewünschte Datei über den Encrypt Service von der Datenbank und gibt einen BufferedStream zurück, mit dem die Datei gelesen werden kann.
5. public GetDirectories(dir : DirectoryInfo) : DirectoryInfo[]  
Liest den Inhalt des gewünschten Verzeichnisses über den Encrypt Service vom Datenbankserver und gibt dieses als Array zurück.
6. public Rename (from : FileInfo, to : FileInfo)  
Benennt eine bestimmte Datei aus dem Cloud Tab um.
7. public UploadFile (file : BufferedStream)  
Lädt eine gewünschte Datei aus einem BufferedStream in die Cloud Tab Datenbank.

#### 2.1.2.4 Controller



Diese Klasse sorgt für den richtigen Programmfluss auf dem Connect Service.

##### Attributes

1. private mediator : IMediator  
Die Implementierung des IMediator Interfaces, die für das verarbeiten jeder Nachricht vom Client verwendet wird.
2. private cloudController : ICloudController  
Die Implementierung des ICloudController Interfaces, die für das verarbeiten aller Cloud Tab Befehle verwendet wird.

##### Methods

1. public Process(msg : String) : String  
Verarbeitet eine vom Client empfangene serialisierte Nachricht und gibt eine ebenfalls serialisierte Antwort zurück, die an den Client gesendet werden kann.
2. private parseAndExecute(msg : String) : Message  
Nimmt eine serialisierte Nachricht als String an, deserialisiert diese und führt den Befehl aus. Zurückgegeben wird eine Nachricht, die das Ergebniss für den Client enthält.
3. public UploadCTFile(file : BufferedStream)  
Lädt eine Datei aus einem BufferedStream in die Cloud Datenbank.

#### 2.1.2.5 SocketListener

Die SocketListener Klasse wird verwendet um die Verbindungen zu den Clients zu verwalten.

## SocketListener

-controller: Controller

+StartListening()

+AcceptCallback(ar : IAsyncResult)

+ReadCallback(ar : IAsyncResult)

-Send(ar : IAsyncResult)

-SendCallback(ar : IAsyncResult)

### Attributes

1. private controller : Controller

Controller, der den Programmfluss auf dem Server reguliert.

### Methods

1. public StartListening()

Diese Methode startet einen Netzwerkdienst, der ankommende Messages empfängt und diese an den Controller weiter gibt.

2. public AcceptCallback(AsyncResult asynResult)

Diese Methode akzeptiert einen validen callback der Clientseite.

3. public ReadCallback (AsyncResult asynResult)

Diese Methode liest den asynchronen Callback , der vom Client geschickt wird.

4. private Send (ar : AsynResult)

Diese Methode schickt einen asynchronen result an den Client.

5. private SendCallback(AsyncResult asynResult)

Diese Methode sendet eine Callback an den Client.

### 2.1.2.6 StoreAlgorithm : Interface



Dieses Interface ist Teil des Encrypt Service und hilft bei einer dynamischen Auswahl des zu benutzenden Verschlüsselungsalgorithmus. Dieses Vorgehen entspricht dem Entwurfsmuster Strategie.

#### Attributes

#### Methods

1. public ObservableCollection<DataBlock> Access(key : String)  
Es wird der SQL Befehl „select“ verschlüsselt und an den Provider geschickt. Als Rückgabe wird ein DataBlock erwartet, der das Resultat einer SQL-Anfrage enthält.
2. public boolean Delete(key : String)  
Es wird der SQL Befehl „delete“ verschlüsselt und an den Provider geschickt
3. public void Initialize()  
Es wird der SQL Befehl „initialize“ verschlüsselt und an den Provider geschickt
4. public void insert(data : String, key : String)  
Es wird der SQL Befehl „insert“ verschlüsselt und an den Provider geschickt
5. public void update(oldKey : String, newKey : String)  
Es wird der SQL Befehl „update“ verschlüsselt und an den Provider geschickt

## 2.1.3 Client

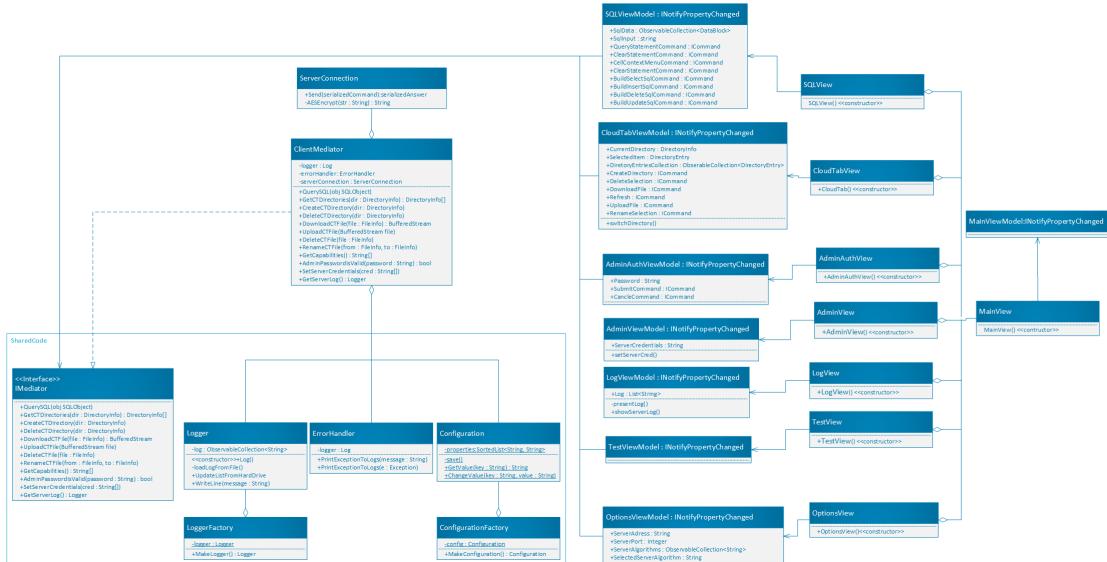


Abbildung 6: Klassendiagramm der Clientanwendung

### 2.1.3.1 MainView



Der MainView ist eine WPF View Klasse, welche die Benutzerschnittstelle mit XAML Code erstellt. Der MainView beschreibt das Client Anwendungs-fenster, in dem im Laufe der Anwendung verschiedene Views bedient werden können. Aufgrund der MVVM Struktur des Projektes, enthält der MainView keine Programm Daten und so wenig CodeBehind wie möglich.

### Methods

#### 1. public MainView() <<constructor>>

Der Konstruktor des MainView, in dem die Oberfläche erstellt wird.

### 2.1.3.2 SQLView

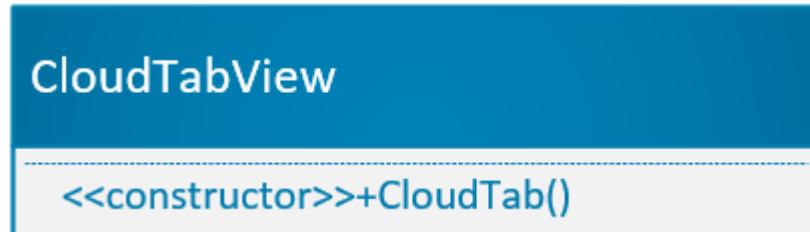


Der SQLView ist eine WPF View Klasse. Der View erzeugt die Benutzeroberfläche, die zum Erstellen und Anzeigen von SQL Abfragen benötigt wird. Aufgrund der MVVM Struktur des Projektes, enthält der SQLView keine Programm Daten und so wenig CodeBehind wie möglich.

#### Methods

1. public SQLView() <<constructor>>  
Der Konstruktor des SQLView, in dem die Oberfläche erstellt wird.

### 2.1.3.3 CloudTabView



Der SQLView ist eine WPF View Klasse, welche die Benutzeroberfläche für den Cloud Tab erstellt. Aufgrund der MVVM Struktur des Projektes, enthält der CloudTabView keine Programm Daten und so wenig CodeBehind wie möglich.

#### Methods

1. public CloudTabView() <<constructor>>  
Der Konstruktor des CloudTabView, in dem die Oberfläche erstellt wird.

#### 2.1.3.4 AdminAuthView

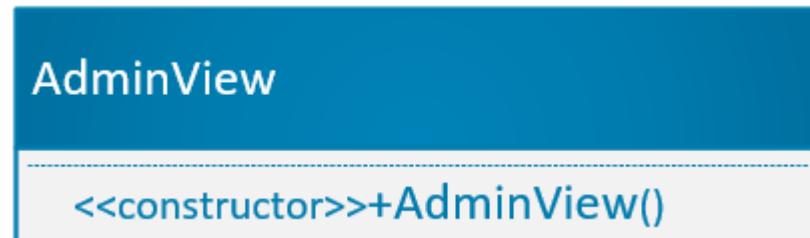


Der AdminAuthView ist eine WPF View Klasse, welche die Benutzeroberfläche für den Admin Authenticate View erstellt. Dieser View wird benutzt um einen Administrator am Server zu authentifizieren. Aufgrund der MVVM Struktur des Projektes, enthält der AdminAuthView keine Programm Daten und so wenig CodeBehind wie möglich.

#### Methods

1. public AdminAuthView() <<constructor>>  
Der Konstruktor des AdminAuthView, in dem die Oberfläche erstellt wird.

#### 2.1.3.5 AdminView



Der AdminView ist eine WPF View Klasse, welche die Benutzeroberfläche für den Admin View erstellt. Mit dem Admin View kann ein Administrator die Einstellungen auf dem Mediator Server anpassen. Aufgrund der MVVM Struktur des Projektes, enthält der AdminView keine Programm Daten und so wenig CodeBehind wie möglich.

#### Methods

1. public AdminView() <<constructor>>  
Der Konstruktor des AdminView, in dem die Oberfläche erstellt wird.

### 2.1.3.6 LogView

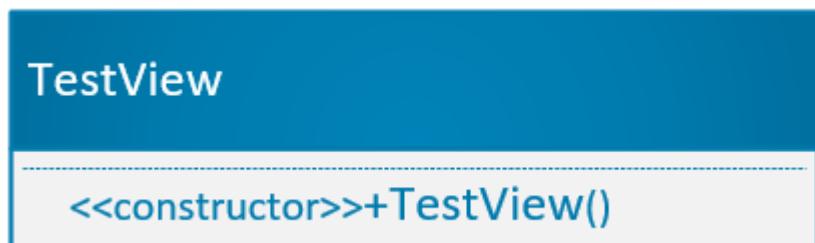


Der LogView ist eine WPF View Klasse, welche die Benutzeroberfläche für den Log View erstellt. Auf diesem kann ein Anwender sich alle vermerkten Anfragen, sowie aufgetretene Fehlermeldungen anzeigen und grafisch Aufbereiten lassen. Ein Administrator hat zusätzlich noch die Möglichkeit Server Log Dateien abzurufen und anzuzeigen. Aufgrund der MVVM Struktur des Projektes, enthält der LogView keine Programm Daten und so wenig CodeBehind wie möglich.

#### Methods

1. public LogView() <<constructor>>  
Der Konstruktor des LogView, in dem die Oberfläche erstellt wird.

### 2.1.3.7 TestView



Der TestView ist eine WPF View Klasse, welche die Benutzeroberfläche für den Test View erstellt. Mit diesem View können Performance Tests von Client aus auf dem Server ausgeführt werden. Aufgrund der MVVM Struktur des Projektes, enthält der TestView keine Programm Daten und so wenig CodeBehind wie möglich.

#### Methods

1. public TestView() <<constructor>>  
Der Konstruktor des TestView, in dem die Oberfläche erstellt wird.

### 2.1.3.8 OptionsView



Der OptionsView ist eine WPF View Klasse, welche die Benutzeroberfläche für den Options View erstellt. Mit diesem View kann der Anwender die lokale Einstellungen der Client Anwendung einsehen und anpassen. Aufgrund der MVVM Struktur des Projektes, enthält der OptionsView keine Programm Daten und so wenig CodeBehind wie möglich.

#### Methods

1. public TestView() <<constructor>>

Der Konstruktor des TestView, in dem die Oberfläche erstellt wird.

### 2.1.3.9 MainViewModel : INotifyPropertyChanged

## MainViewModel:INotifyPropertyChanged

Das ViewModel dient in der MVVM Struktur im allgemeinen als Bindeglied zwischen dem MainView und dem Model. Da der MainView nur als Controller für die verschiedenen Views dient und keine Anwendungsdaten visualisiert, stellt das MainViewModel hier keine öffentlichen Eigenschaften für den MainView bereit.

#### Attributes

-

#### Methods

-

### 2.1.3.10 SQLViewModel

#### SQLViewModel : INotifyPropertyChanged

```
+SqlData : ObservableCollection<DataBlock>
+SqlInput : String
+QueryStatementCommand : ICommand
+ClearStatementCommand : ICommand
+CellContextMenuCommand : ICommand
+ClearStatementCommand : ICommand
+BuildSelectSqlCommand : ICommand
+BuildInsertSqlCommand : ICommand
+BuildDeleteSqlCommand : ICommand
+BuildUpdateSqlCommand : ICommand
```

Das SQLViewModel stellt wie von der MVVM Struktur des Projektes gefordert, eine Bindung zwischen dem SQLView und dem Model her. Dazu werden Befehle und Daten des Models öffentlich vom SQLViewModel angeboten, die dann per DataBinding vom SQLView genutzt werden. Das ViewModel hat hierbei keinerlei Kenntnis über den View.

#### Attributes

1. public SqlData : ObservableCollection <DataBlock>  
Enthält Daten, die von der letzten erfolgreichen SQL Abfrage zurück gegeben wurden.
2. public SqlInput : string  
Enthält die Eingabe des Benutzers aus dem SQL-Befehl Eingabefeld.
3. public QueryStatementCommand() : ICommand  
Befehl der beim betätigen des Query Button ausgeführt wird.
4. public ClearStatementCommand() : ICommand  
Befehl der beim betätigen des Clear Button ausgeführt wird.

5. public CellContextMenuCommand() : ICommand  
Befehl der ausgelöst wird wenn ein Eintrag aus dem Kontext Menü gewählt wurde.
6. public BuildSelectSqlCommand() : ICommand  
Befehl der beim betätigen des “Select generieren” Button ausgeführt wird.
7. public BuildInsertSqlCommand() : ICommand  
Befehl der beim betätigen des “Insert generieren” Button ausgeführt wird.
8. public BuildDeleteSqlCommand() : ICommand  
Befehl der beim betätigen des “Delete generieren” Button ausgeführt wird.
9. public BuildUpdateSqlCommand() : ICommand  
Befehl der beim betätigen des “Update generieren” Button ausgeführt wird.

## Methods

### 2.1.3.11 CloudTabViewModel

#### CloudTabViewModel : INotifyPropertyChanged

```
+CurrentDirectory : DirectoryInfo
+SelectedItem : DirectoryEntry
+DirectoryEntriesCollection : ObservableCollection<DirectoryEntry>
+.CreateDirectory : ICommand
+DeleteSelection : ICommand
+DownloadFile : ICommand
+Refresh : ICommand
+UploadFile : ICommand
+RenameSelection : ICommand
+SwitchDirectory()
```

Das CloudTabViewModel stellt wie von der MVVM Struktur des Projektes gefordert, eine Bindung zwischen dem CloudTabView und dem Model her. Dazu werden Befehle und Daten des Models öffentlich vom CloudTabViewModel angeboten, die dann per DataBinding vom CloudTabView genutzt werden. Das ViewModel hat hierbei keinerlei Kenntnis über den View.

## Attributes

1. public CurrentDirectory : DirectoryInfo  
Enthält Daten und Eigenschaften des aktuell angezeigten Verzeichnisses.
2. public SelectedItem : DirectoryEntry  
Enthält ein Element des aktuellen Verzeichnisses, das vom User markiert wurde. Mit diesem Element können dann Aktionen durchgeführt werden.
3. public DirectoryEntriesCollection : ObservableCollection<DirectoryEntry>  
Enthält alle Elemente, die im aktuellen Verzeichnis liegen.
4. public CreateDirectory : ICommand  
Befehl, der im aktuellen Verzeichnis ein neues Verzeichnis erstellt.
5. public DeleteSelection : ICommand  
Befehl, der das aktuell selektierte Element (siehe SelectedItem) aus der Cloud Datenbank löscht.
6. public DownloadFile : ICommand  
Befehl, der das aktuell selektierte Element (siehe SelectedItem) aus der Cloud Datenbank abruft, sofern es sich um eine Datei handelt.
7. public Refresh : ICommand  
Befehl, der den Verzeichnisinhalt aktualisiert und erneut anzeigt.
8. public UploadFile : ICommand  
Befehl, der eine neue Datei vom Client auf den Cloud Datenbank Server überträgt.
9. public RenameSelection : ICommand  
Befehl, der das aktuell selektierte Element (siehe SelectedItem) umbenennt.

## Methods

1. public switchDirectory()  
Diese Funktion erlaubt es das aktuell angezeigte Verzeichnis wechseln.

### 2.1.3.12 AdminAuthViewModel

**AdminAuthViewModel : INotifyPropertyChanged**

+Password : String  
+SubmitCommand : ICommand  
+CancleCommand : ICommand

Das AdminAuthViewModel stellt wie von der MVVM Struktur des Projektes gefordert, eine Bindung zwischen dem AdminAuthView und dem Model her. Dazu werden Befehle und Daten des Models öffentlich vom AdminAuthViewModel angeboten, die dann per DataBinding vom AdminAuthView genutzt werden. Das ViewModel hat hierbei keinerlei Kenntnis über den View.

#### Attributes

1. public Password : String  
Enthält den vom Benutzer eingegebenen Authentifikations String.
2. public SubmitCommand : ICommand  
Befehl um das Passwort zum Server zu senden und prüfen zu lassen.
3. public CancleCommand : ICommand  
Befehl um die Authentifikation abzubrechen.

#### Methods

### 2.1.3.13 AdminViewModel

**AdminViewModel : INotifyPropertyChanged**

+ServerCredentials : String  
+setServerCred()

Das AdminViewModel stellt wie von der MVVM Struktur des Projektes gefordert, eine Bindung zwischen dem AdminView und dem Model her. Dazu werden Befehle und Daten des Models öffentlich vom AdminViewModel angeboten, die dann per DataBinding vom AdminView genutzt werden. Das ViewModel hat hierbei keinerlei Kenntnis über den View.

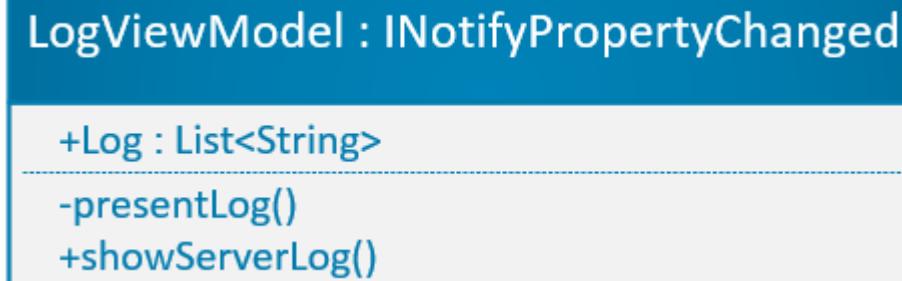
### Attributes

1. public ServerCredentials : String  
Enthält die vom Administrator eingegebenen neuen Cloud Server Zugangsdaten.

### Methods

1. public void setCredentials()  
Funktion um die vom Administrator gewählten Server Credentials auf dem Server zu speichern.

#### 2.1.3.14 LogViewModel



Das LogViewModel stellt wie von der MVVM Struktur des Projektes gefordert, eine Bindung zwischen dem LogView und dem Model her. Dazu werden Befehle und Daten des Models öffentlich vom LogViewModel angeboten, die dann per DataBinding vom LogView genutzt werden. Das ViewModel hat hierbei keinerlei Kenntnis über den View.

### Attributes

1. public Log : List<String>  
Enthält das aktuell angezeigte Log.

### Methods

1. public void showServerLog()  
Funktion um einem Adminstrator auch das herunterladen und ansehen der Server Log Dateien zu erlauben.
2. private void presentLog()  
Diese Funktion bereitet geladene Log Daten für das Anzeigen auf dem View vor.

#### 2.1.3.15 TestViewModel

### TestViewModel : INotifyPropertyChanged

Das TestViewModel stellt wie von der MVVM Struktur des Projektes gefordert, eine Bindung zwischen dem TestView und dem Model her. Dazu werden Befehle und Daten des Models öffentlich vom TestViewModel angeboten, die dann per DataBinding vom LogView genutzt werden. Das TestView hat hierbei keinerlei Kenntnis über den View.

#### Attributes

#### Methods

#### 2.1.3.16 OptionsViewModel

### OptionsViewModel : INotifyPropertyChanged

```
+ServerAdress : String  
+ServerPort : Integer  
+ServerAlgorithms : ObservableCollection<String>  
+SelectedServerAlgorithm : String
```

Das OptionsViewModel stellt wie von der MVVM Struktur des Projektes gefordert, eine Bindung zwischen dem OptionsView und dem Model her. Dazu werden Befehle und Daten des Models öffentlich vom OptionsViewModel angeboten, die dann per DataBinding vom OptionsView genutzt werden. Das TestView hat hierbei keinerlei Kenntnis über den View.

## Attributes

1. public ServerAdress : String  
Enthält die vom Benutzer konfigurierte IP Adresse an dem der Mediator Dienst angesprochen werden kann.
2. public ServerPort : Integer  
Enthält den vom Benutzer konfigurierten Port an dem der Mediator Dienst angesprochen werden kann.
3. public ServerAlgorithms : ObservableCollection<String>  
Enthält den Algorithmus den der Benutzer für die Datenbank ausgewählt hat.

## Methods

---

### 2.1.3.17 ClientMediator : IMediator

## ClientMediator

```
-configuration : Configuration  
-errorHandler : ErrorHandler  
-logger : Log  
-serverConnection : ServerConnection  
  
+ExecuteSQL(query : SQLQuery)  
+GetCapabilities() : String[]  
+AdminPasswordIsValid(password : String) : bool  
+SetServerCredentials(cred : String[])  
+GetServerLog() : Logger
```

Implementiert das IMediator Interface.

### Attributes

1. private logger : Logger  
Eine Logger Instanz mit dem alle Ereignisse vermerkt werden.
2. private errorHandler : ErrorHandler  
Eine Error Logger Instanz mit dem alle Fehlermeldungen vermerkt werden.
3. private serverConnection : ServerConnection  
Die ServerConnection Klasse, welche die Verbindung zum Mediator Dienst verwaltet.

### Methods

1. public QuerySQL(obj : SQLObject)  
Erhält eine SQL-Anfrage als String und gibt diese an die SQLQuery Klasse zum parsen weiter. Danach wird mit dem so erzeugten SQLQuery Objekt eine SQLMessage erstellt, die anschließend an den Server übermittelt wird.
2. public GetCTDirectories(dir : DirectoryInfo) : DirectoryInfo[]  
Holt die Verzeichnisliste des Cloud Tab vom Server.
3. public CreateCTDirectory (dir : DirectoryInfo)  
Erstellt ein neues Verzeichnis im Cloud Tab.
4. public DeleteCTDirectory (dir : DirectoryInfo)  
Löscht ein Verzeichnis aus dem Cloud Tab.
5. public DownloadCTFile (file : FileInfo) : BufferedStream  
Holt eine bestimmte Datei vom Server und stellt sie dem Benutzer zur Verfügung.
6. public UploadCTFile (file : BufferedStream)  
Erstellt eine neue Datei im Cloud Tab.
7. public DeleteCTFile (file : FileInfo)  
Löscht eine Datei aus dem Cloud Tab.
8. public RenameCTFile (from : FileInfo, to : FileInfo)  
Benennt eine Datei aus dem Cloud Tab um.
9. public GetCapabilities() : String[]  
Fragt die vom Server unterstützten SQL-Befehle an.
10. public AdminPasswordIsValid(password : String) : boolean  
Sendet vom Benutzer eingegebene Zugangsdaten zur Authentifizierung als Administrator an den Server.
11. public SetServerCredentials(cred : String[])  
Sendet neue Cloud Datenbank Zugangsdaten an den Server.
12. public GetServerLog() : Logger  
Ruft die Logs vom Server ab.

### 2.1.3.18 ServerConnection

## ServerConnection

+Send(serializedCommand : String) : String  
+Send(file : BufferedStream)  
-AESEncrypt(str : String) : String

Verwaltet den Verbindungsauflauf, die Kommunikation und die Verbindung zum Mediator Server.

### Attributes

-

### Methods

1. public Send(serializedCommand : String) : String  
Baut eine Verbindung zum Server auf, sendet einen serialisierten Befehl und gibt die Antwort des Servers zurück.
2. private AESEncrypter(str : String) : String

### 3 Sequenzdiagramme

#### 3.1 Anwendungsfall: Administrator ändert Default Zugangsdaten

Ein Administrator klickt auf die Schaltfläche zum Öffnen der Ansicht für Administratoren (AdminView). Daraufhin wird ihm ein Pop-Up-Fenster mit Textfeld angezeigt. In dieses tippt der Administrator sein Passwort und sendet es ab. Als nächstes wird das Passwort zur Überprüfung an den Server geschickt. Falls diese negativ ausfällt zeigt das Pop-Up-Fenster eine dementsprechende Nachricht an, falls das Ergebnis positiv ist, öffnet sich die AdminView und der Administrator kann die gewünschten Einstellungen ändern. Diese werden an den Server gesendet und dort vorgenommen.

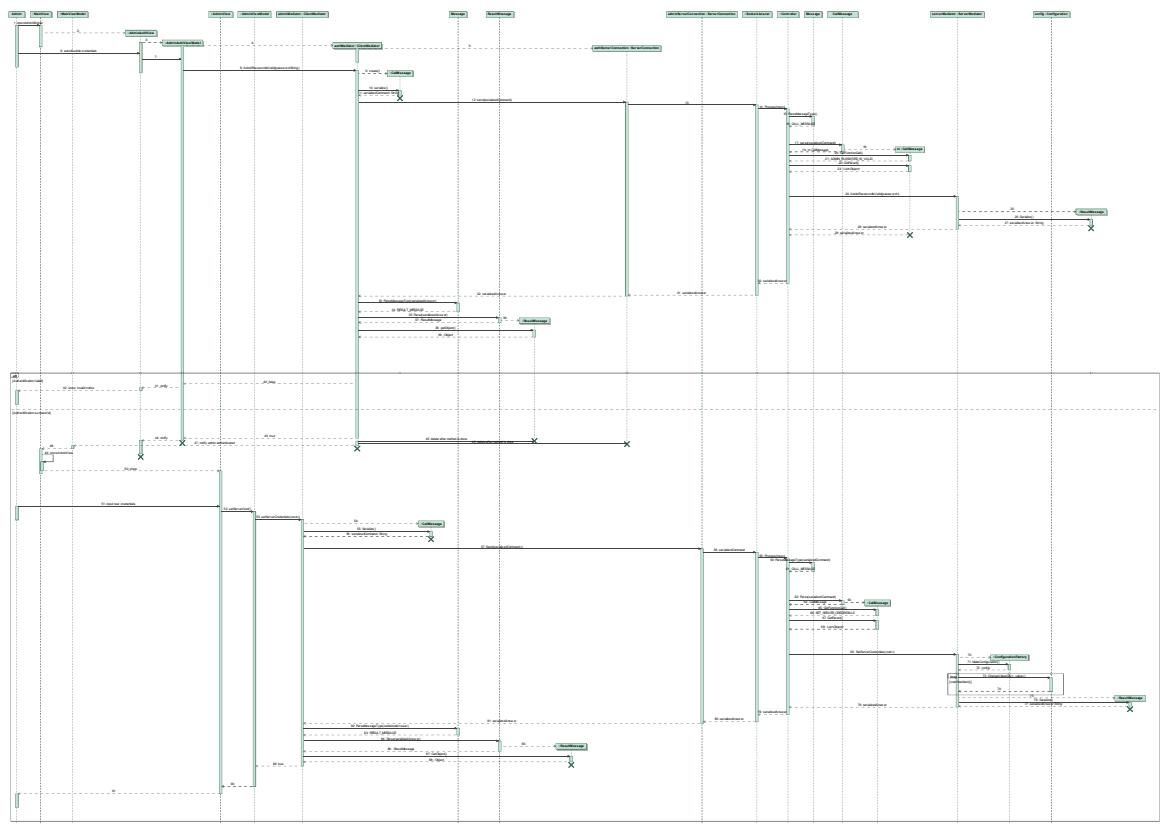


Abbildung 7: Sequenzdiagramm des Anwendungsfalls: Administrator ändert Default Zugangsdaten

## 3.2 Anwendungsfall: Ausführen einer SQL-Anfrage

Ein Benutzer schickt eine SQL-Anfrage per Knopfdruck ab, diese wird analysiert und, falls sie syntaktisch inkorrekt ist, erscheint eine Fehlermeldung. Bei einer korrekten Anfrage wird diese dem Server übergeben und dort vom Encrypt Service ausgeführt. Anschließend werden die Ergebnisse an den Client übertragen und dort dargestellt.

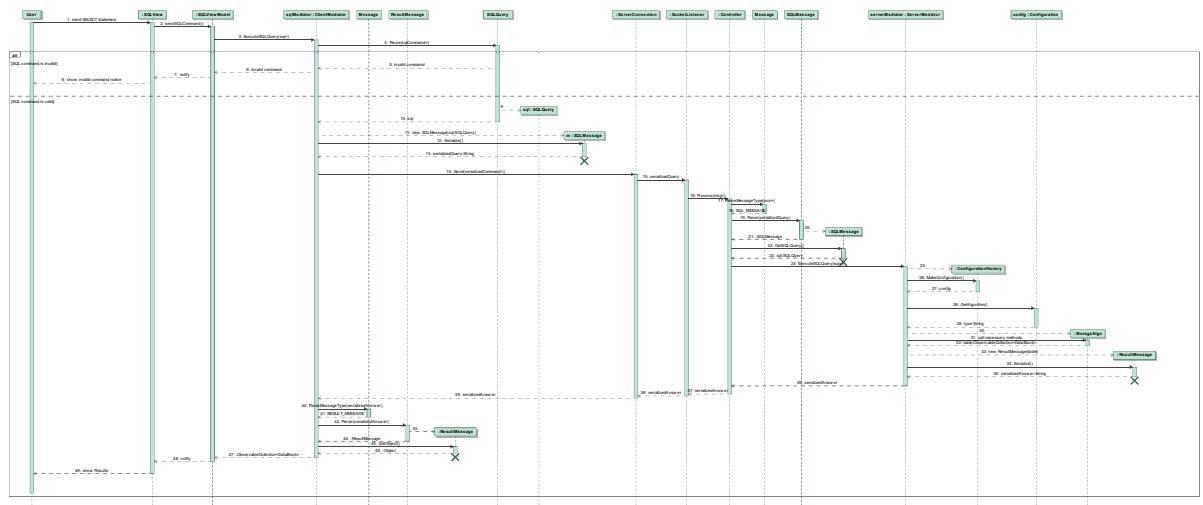


Abbildung 8: Sequenzdiagramm des Anwendungsfalls: Ausführen einer SQL-Anfrage

### 3.3 Anwendungsfall: Benutzer speichert Datei im Cloud Tab

Ein Benutzer fügt in der Cloud Tab Ansicht eine Datei hinzu. Diese wird dem Server übertragen und dort in serialisierter Form in die Datenbank eingespeist.

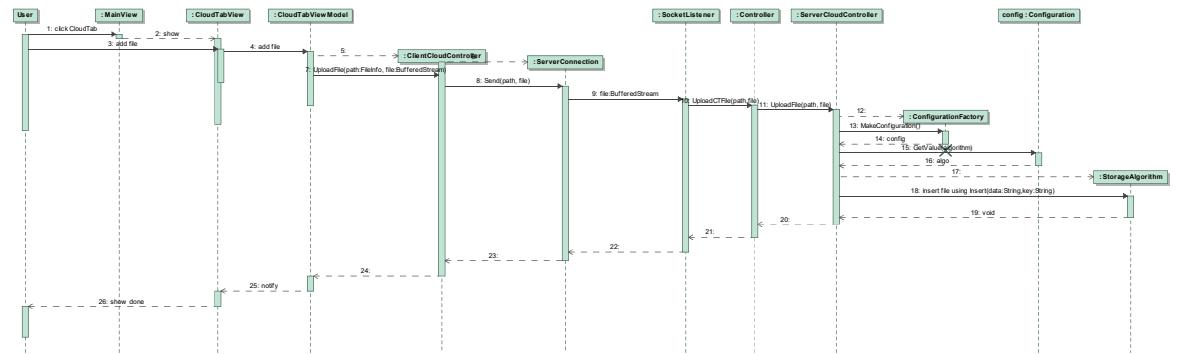


Abbildung 9: Sequenzdiagramm des Anwendungsfalls: Benutzer speichert Datei im Cloud Tab

