**Inventar Server**

Haupt-Folder:

InventarServerMain

Hier startet das Program, lädt Libaries, MongoDB und den Server

public static void Main(string[] args): Ladet die Library fürs Excel lesen und speichert sich eine Instance auf das ganze Programm, damit innerhalb im public Kontext und außerhalb im statischen Kontext gearbeitet werden kann

public static MongoClient GetMongoDB(): Vereinfacht Zugriff auf die MongoDB Datenbank

public InventarServerMain(): Ladet MongoDB und startet den Server auf einem vorbestimmten Port (Standart-Port: 10000)

General-Folder:

AdminTest

Ermöglicht den Zugriff auf Admin-Datenbank-Funktionen

public static void CreateAdminUserDatabase(): Erstellt die Super-Admins Database, wird für Setup benötigt, da ansonsten keine neuen Datenbanken angelegt werden können

Client

Sorgt für die Command-based Kommunikationen zwischen dem Server und den Clients

public Client(TcpClient \_client, CommandManager \_cmdManager): Lädt Client und startet die weiter Kommunikationen in seinem eigenen Thread

private void ClientRoutine():

1. Startet RSA Kommunikation (siehe Punkt RSA)
2. Loggt sich bei einer Datenbank mit einem User ein und schaut ob der Login ok ist
3. Startet Schleife die erst endet, wenn die Verbindung beendet ist
   1. Liest Command aus und gibt in an die Zugehörige Command Klasse weiter

private void Close(): Schließt die Verbindung

Server

Startet den Server und gibt neue Verbindungen als Client weiter

public Server(string \_domain, int \_port): Startet den Server mit einer Domain auf einem vorgegebenen Port

public Server(int \_port): Startet den Server im localhost auf einem vorgegebenen Port

private void StartServer(string \_domain, int \_port):

1. Startet den Server mit einer Domain auf einem vorgegebenen Port
2. Startet die weitere Kommunikation in seinem eigenen Thread
3. Lädt alle Commands

private void ServerRoutine(): Akzeptiert neue Verbindungen und registriert diese als Clients

public static void WriteLine(string \_s, params object[] \_args): Schreibt Nachricht and Console, aber nur wenn sich das Programm im DEBUG mode befindet

StreamHelper

RSAHelper kann nur byte arrays schicken, der StreamHelper ermöglicht es nun komplexere Daten zu schicken

public StreamHelper(RSAHelper \_helper): Erstellt Objekte die beim umwandeln der Daten in byte arrays helfen

public void SendByteArray(byte[] \_bytes): Sendet eine byte array

public byte[] ReadByteArray(): Liest eine byte array

public void SendString(string \_s): Sendet einen String im ASCII Format

public string ReadString(): Liest einen String im ASCII Format

public void SendInt(int \_x): Sendet einen Integer

public int ReadInt(): Liest einen Integer

Security-Folder:

RSA

Sorgt für eine verschlüsselte Kommunikation, verwendet wird das RSA verfahren (siehe Punkt RSA)

public RSA(): Erstellt das RSA Verschlüsselungs-Objekt von C#, erzeugt einen Public-Key für die Kommunikation

public RSA(string publicKey): Macht das gleich wie nur RSA(), diesmal ist der Public-Key allerdings vorgegeben

public byte[] Encrypt(byte[] \_data): Verschlüsselt eine byte array

public byte[] Decrypt(byte[] \_data): Entschlüsselt eine byte array

RSAHelper

Initialisiert die Verschlüsselung und das Austausch-Protokoll wird hier eingeleitet (siehe Punkt RSA)

public RSAHelper(NetworkStream \_stream): Erstellt Objekte zur Ver- und Entschlüsselung

public void SetupServer(): Kümmert sich um die Server Seite des Austausch-Protokolls (siehe Punkt RSA)

public void SetupClient(): Kümmert sich um die Client Seite des Austausch-Protokolls (siehe Punkt RSA)

public byte[] ReadBytes(int \_amount): Liest eine gewisse Anzahl an unverschlüsselten Bytes

private int Read2Bytes(): Liest 2 bytes und wandelt diese in einen Integer um

private void WriteBytes(byte[] \_data): Schreibt eine unverschlüsselte byte array

private void Write2Bytes(int \_x): Schreibt die letzten 2 bytes des Integers (least significant bytes) unverschlüsselt

private void WriteEncryptedBytes(byte[] \_data): Schreibt eine verschlüsselte byte array, die maximal 86 bytes groß ist

private byte[] ReadDecryptedBytes(): Liest 128 bytes auf einmal, die auf 86 verschlüsselte bytes gesenkt werden

public void WriteByteArray(byte[] \_data): Schreibt beliebig große byte array, das große byte array wird in kleinere, maximal 86 bytes lange byte arrays aufgeteilt und einzeln geschickt (siehe Punkt RSA)

public byte[] ReadByteArray(): Liest eine beliebig großes byte array, die kleinen byte arrays vom Senden werden zu einem großen zusammengefasst

Validator

Wird verwendet um Emails, Usernamen und Passwörter auf Korrektheit zu überprüfen

public static bool ValidateUsername(string \_username):

1. Länge 2 bis 20 Zeichen
2. Erlaubt sind:
   1. Großbuchstaben
   2. Kleinbuchstaben
   3. Zahlen
   4. Underscore (\_)
3. Ein Name darf nicht mit Underscore (\_) starten

public static bool ValidateEmail(string \_email): Standard Email Validation laut C#

public static bool ValidatePassword(string \_password):

1. Länge 6 bis 20
2. Muss enthalten:
   1. Großbuchstaben
   2. Kleinbuchstaben
   3. Zahlen
   4. Sonderzeichen (!@#$%^&\*()\_+=\[{\]};:<>|./?,-)

Database-Folder:

Collection

Erleichtert den Zugriff auf die Collections von MongoDB

public Collection(IMongoCollection<BsonDocument> \_collection): Speichert die Collection für späteren Zugriff

public BsonDocument FindOne(string \_filterCriterium, string \_name): Sucht nach einem Dokument in der Collection, das Filter-Kriterium muss stimmen

public void AddOne(BsonDocument \_doc): Fügt ein Dokument der Collection hinzu

public List<BsonDocument> GetAll(): Gibt eine Liste aller Dokumente der Collection zurück

public void UpdateEntry(string \_filterCriterium, string \_name, BsonDocument \_newDoc): Ersetzt ein Dokumente, bei dem das Filter-Kriterium übereinstimmt

public void RemoveOne(string \_filterCriterium, string \_name): Entfernt ein Dokument von der Collection, bei dem das Filter-Kriterium übereinstimmt

Database

Erleichtert den Zugriff auf die Datenbanken von MongoDB

public Database(string \_name): Lädt eine Datenbank mit dem gleichen Namen

public LoginError Login(User \_u): Schaut ob der Login von einem User ok ist (siehe Punkt Secure Password Storing)

public string Hash(string \_salt, string \_password): Hashed ein Passwort, das Passwort wird vorher gepeppered und gesalted, im Moment wird 10000 mal gehashed

public string GenerateRandomSalt(): Erzeugt ein zufälliges, 32 byte langes Salt, dass in Base64 abgespeichert wird

public Collection GetCollection(string \_name): Lädt eine Collection mit dem gleichen Namen, falls die Collection nicht existiert wird sie neu erzeugt

public MongoClient Mongo(): Gibt die Haupt-Instanz von MongoDB zurück

public bool Exists(): Gibt an, ob die jetzige Datenbank überhaupt existiert

public void CreateNewDatabase(): Erstellt die Datenbank, erzeugt die Standart-Collections und erstellt eine Master-Item-Collection (siehe Punkt Datenbank Aufbau)

public ValidateError AddUser(string \_email, string \_username, string \_password): Überprüft ob die Email, der Nutzename und das Paswwort ok sind, falls ja wird ein neuer User erzeugt und zur User-Collection hinzugefügt

private ValidateError EmailOrUsernameUsed(string \_email, string \_username): Schaut ob ein anderer User schon die gleiche Email oder dem gleichen Nutzernamen verwendet hat

public List<UserData> GetUserData(): Gibt die sehbaren Daten aller Users zurück

public List<string> ListItemCollections(): Erzeugt eine Liste, die die Namen der Item-Collections speichert

public List<string> ListItemCollections(User \_u): Erzeugt eine Liste, die die Namen der Item-Collections speichert, allerdings erhält diese Liste nur die Collections für die der User auch Rechte hat

public ItemCollection GetItemCollection(string \_name): Gibt die Item-Collection mit dem gleichen Namen zurück#

public void AddItem(Item \_i): Fügt ein Item der Korrekten Item-Collection hinzu

DatabaseHelper

Bietet Funktionen die den Zugriff auf MongoDB vereinfachen

public static List<string> ListAllDatabases(): Gibt eine Liste der Namen aller Datenbanken zurück

public static List<string> ListUseableDatabases(): Gibt eine Liste der Namen aller benutzbaren Datenbanken zurück. MongoDB hat Standartmäßig bereits 3 Datenbanken, nämlich admin, config und local, diese sollen nicht gelistet werden

LoginError (enum)

Definiert die Fehler-Typen, die beim Zugriff auf eine Datenbank entstehen können

NONE: Kein Fehler

WRONG\_DATABASE: Datenbank existiert nicht

WRONG\_USERNAME: Username oder Email ist Falsch

WRONG\_PASSWORD: Passwort ist Falsch

UNKNOWN: Ein Unbekannter Fehler

Excel Helper

Ermöglicht das Lesen von Excel Dateien um das Hinzufügen von mehrere Items auf einmal zu ermöglichen

public static void Setup(): Registriert das Encoding für Excel

public static List<Item> LoadExcel(string \_name, string \_filename): Lädt die Items aus einer Excel Datei

Item

Speichert die Daten eines Items plus ein paar extra Details (siehe Punkt Datenbank Aufbau)

public Item(): Erstellt ein neues Item mit den Standart Werten

public Item(BsonDocument \_doc): Erstellt ein neues Item mit den Werten aus dem Dokument einer Item-Collection

public Item(object[] data): Erstellt ein neues Item von den Daten in einer Excel Zeile

public string[] ToStrings(): Wandelt das Item in ein string array um, um die Darstellung zu erleichtern

public override string ToString(): Stellt das Item als Text dar

public BsonDocument GetItemAsBson(): Generiert ein Dokument für die Item-Collection aus den Daten des Items

public void FromBson(BsonDocument \_doc): Erstellt ein neues Item mit den Werten aus dem Dokument einer Item-Collection

public void FromExcelCol(object[] data): Erstellt ein neues Item von den Daten in einer Excel Zeile

public void LoadImages(BsonDocument \_doc): Lädt die Bilder eines Items, werden Standartmäßig nicht geladen, da dies realitv lange Zeit braucht

public void GenerateID(): Erstellt eine neue ID für ein Item

public byte[] ToByteArray(): Wandelt das Item in eine byte array um

ItemCollection

Erleichtert den Zugriff auf die Item-Collections in der items-Collection

public ItemCollection(Database \_db, BsonDocument \_collection): Speichert eine Item-Collection

public ItemCollection(Database \_db, string \_itemCollectionName): Lädt eine Item-Collection und speichert sie

public string GetName(): Gibt den Namen der Item-Collection zurück

public string GetPermission(): Gibt die Permission der Item-Collection

public List<Item> GetItems(User \_u): Gibt alle Items einer Item-Collection zurück

public Item GetItem(User \_u, string \_id): Gibt ein Item mit der gleichen ID zurück

public void AddItem(Item \_i): Fügt ein Item der Item-Collection hinzu

public static BsonDocument CreateNew(string \_name, string \_perm): Erstellt eine neue Item-Collection

public BsonDocument Clone(string \_newName, string \_perm): Erstellt eine neue Item-Collection mit den gleichen Items

public bool RemoveItem(string \_itemCollection, string \_id, User \_u): Löscht ein Item mit der gleichen ID, falls der User genug rechte hat

public void RemoveCollection(string \_itemCollection): Löscht die ganze Item-Collection

User

Kümmert sich um das Speichern von Usern und das Rechte Management

public User(Database \_db, string \_username, string \_password): Lädt einen User von einer Datenbank

public User(string \_db, string \_username, string \_password): Lädt einen User von einer Datenbank mit dem angegebenen Namen

private void LoadPermissions(): Lädt die Permissions von einem User

public LoginError Login(): Loggt den User ein

public bool HasSpecificPermission(string \_perm): Überprüft ob ein User die angegebene Permission hat

public bool HasPermission(string \_perm): Überprüft ob ein User die angegeben Permission hat oder Admin ist oder die Permission eine Standart-Permission ist

public bool IsSuperAdminUser(): Überprüft ob ein User ein SuperAdmin ist

public bool IsAdmin(): Überprüft ob ein User ein Admin der Datenbank ist

public bool HasItemAddPermission(): Überprüft ob ein User die Rechte hat um Items hinzuzufügen

public void AddPermission(string \_permission): Fügt dem User eine neu Permission hinzu

public void RemovePermission(string \_permission): Entfernt dem User eine Permission

public BsonDocument AsBson(string \_email, string \_salt): Wandelt den User in ein speicherbares Dokument um

public (BsonDocument user, bool isEmail, bool isUsername) GetUser(): Lädt den User und speichert ob er über eine Email oder einen Nutzernamen geladen wurde

public UserData GetUserData(): Lädt die sichtbaren Daten eines Users

UserData

Speichert die sichtbaren Daten eines Users

public UserData(): Benötigt für die Wandlung in JSON

public UserData(string \_email, string \_username, List<string> \_permissions): Speichert die Daten eines Users

public static UserData FromBson(BsonDocument \_bd): Lädt die Daten eines Users von einem Dokument der Datenbank

public string ToJson(): Wandelt die Daten in JSON um

Commands-Folder:

CommandManager

Speichert die Commands

public CommandManager(): Lädt alle Commands

Command

Speichert die Command Daten und kümmert sich um die Ausführung

public Command(string \_command): Lädt den Command

public void Call(User \_u, StreamHelper \_helper, Client \_c): Führt den Command aus, kann nicht Überschrieben werden

public virtual void Execute(User \_u, StreamHelper \_helper,   
Client \_c): Führt den Command aus, kann Überschrieben werden

public bool SendPermissionMessage(User \_u, StreamHelper \_helper, bool \_hasRights): Sendet dem User den Fehler, dass er nicht genügend Rechte hat, falls er nicht genug Rechte hat

public bool SendPermissionMessage(User \_u, StreamHelper \_helper, string \_perm): Sendet dem User den Fehler, dass er nicht genügend Rechte hat, falls er nicht genug Rechte hat

public bool IsAdmin(User \_u, StreamHelper \_helper): Überprüft ob der User Admin Rechte hat

public bool HasItemAddPermission(User \_u, StreamHelper \_helper): Überprüft ob der User Recht für das Hinzufügen von Items hat

public void SendNoPermissionMessage(StreamHelper \_helper): Sendet die Nachricht, dass der User nicht genug Rechte hat

public void SendOKMessage(StreamHelper \_helper): Sendet die Nachricht, dass kein Fehler aufgetreten ist

public static byte[] Zip(string \_uncompressed): Komprimiert eine Nachricht

public static string Unzip(byte[] \_compressed): Ent-Komprimiert eine Nachricht

Jeder Command hat den gleichen Aufbau, der Constructor speichert den Namen des Commands und die Execute Funktion führt den Command aus

Die Funktionen der Commands werden nicht weiter erklärt, da sie überall gleich sind

LoginCommand

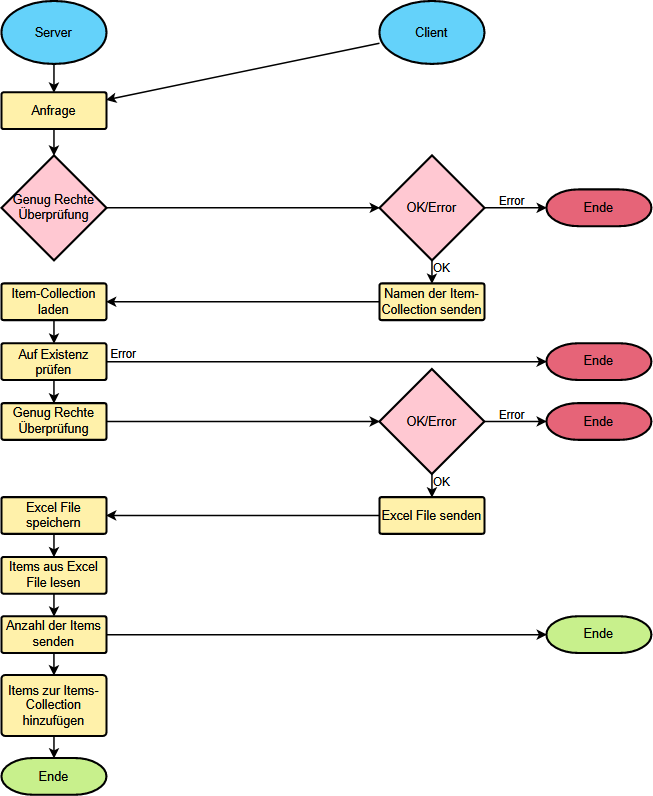
Leerer Command, kümmert sich um den Login

Da vor dem Command senden der User schon eingeloggt sein muss, ist dieser Command leer, da der Login bereits überprüft wurde

AddExcelItemsCommand

Fügt alle Items aus einem Excel Dokument der Datenbank hinzu

Flussdiagramm:



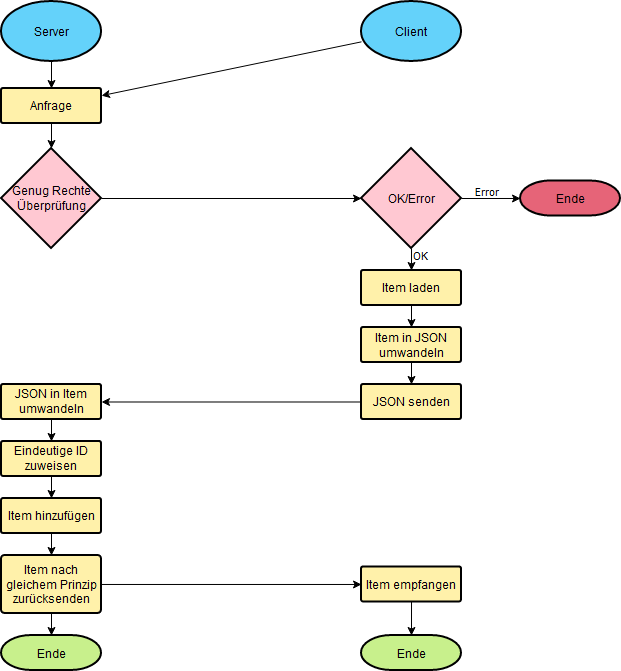
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User das Recht hat, um Items hinzuzufügen, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Client sendet den Namen der Item-Collection bei der die Excel-Items hinzugefügt werden sollen
4. Server lädt die Item-Collection und schaut ob sie existiert und ob der User die Rechte hat sie zu verändern, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
5. Client sendet das Excel-File, wird auf dem Server zwischengespeichert
6. Server lädt die Items aus dem File und sendet die Anzahl der Items an den Client zurück, Client Kommunikation ist nun vorbei
7. Server fügt die Items der Item-Collection hinzu, Server Kommunikation ist nun vorbei

AddItemCommand

Fügt ein Item der Datenbank hinzu

Flussdiagramm:



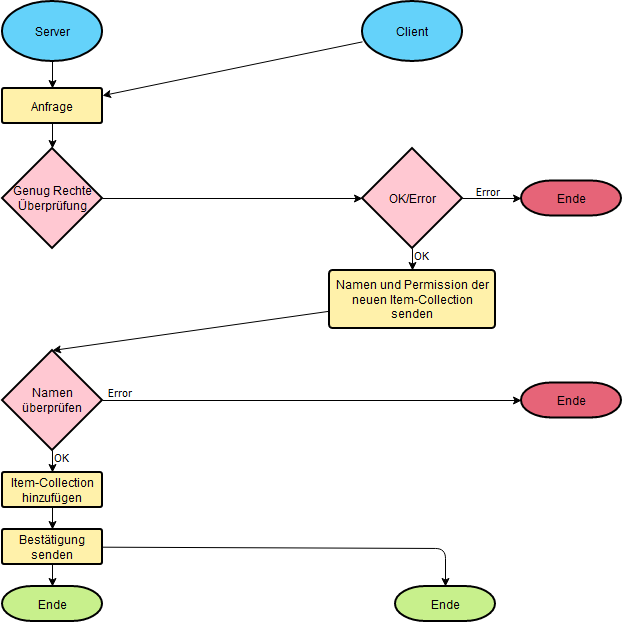
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User das Recht hat, um Items hinzuzufügen, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Client lädt das Item und wandelt es in JSON um
4. Der JSON-String wird an den Server gesendet
5. Der Server empfängt den String und wandelt ihn wieder in ein Item um
6. Das Item bekommt eine neue, eindeutige ID zur klaren Identifizierung
7. Das Item wird zur Datenbank hinzugefügt
8. Zur Bestätigung wird das neue Item, mit der neuen ID an den Client zurückgesendet

AddItemCollectionCommand

Fügt ein neue Item-Collection zur Datenbank hinzu

Flussdiagramm:



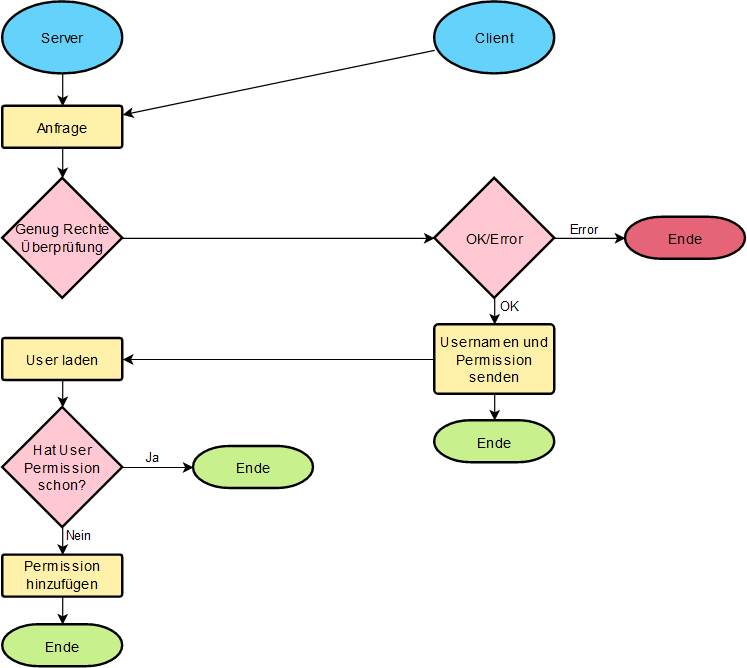
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User Admin Rechte hat, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Client sendet den Namen der neuen Item-Collection und die Permission, die diese haben soll
4. Server überprüft nun, ob es schon eine Item-Collection mit dem gleichen Namen gibt
5. Die Item-Collection wird hinzugefügt und der Client bekommt eine Bestätigung zurück

AddPermissionCommand

Fügt einem User neue Rechte hinzu

Flussdiagramm:



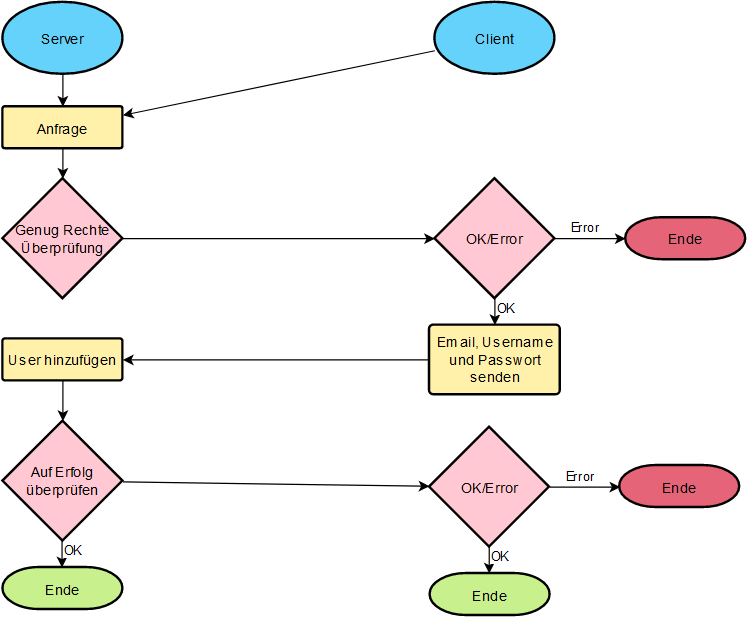
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User Admin Rechte hat, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Client sendet den Usernamen des Users, der die neuen Rechte erhalten soll
4. Client sendet die neuen Rechte des Users
5. Server lädt den User und überprüft, ob dieser existiert und ob er die Rechte nicht schon hat
6. Hat der User das Recht noch nicht, so wird das neue Recht dem User hinzugefügt

AddUserCommand

Fügt einen neuen User der Datenbank hinzu

Flussdiagramm:



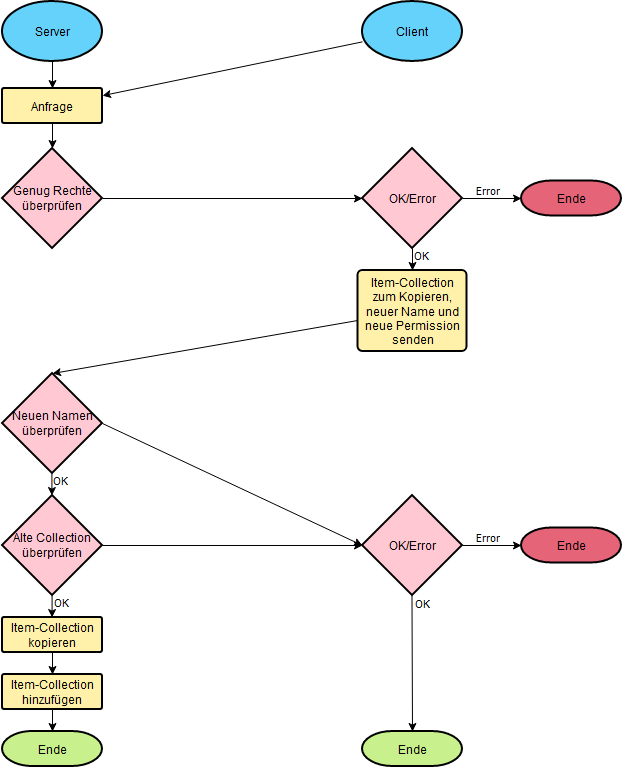
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User Admin Rechte hat, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Client sendet den Usernamen, die E-Mail und das Passwort des neuen Users
4. Die Daten des Users werden überprüft (siehe Klasse Validator)
5. Der Server fügt den User hinzu
6. Der Client wird benachrichtigt, ob der User erfolgreich hinzugefügt wurde

CopyItemCollectionCommand

Kopiert eine Item-Collection

Flussdiagramm:



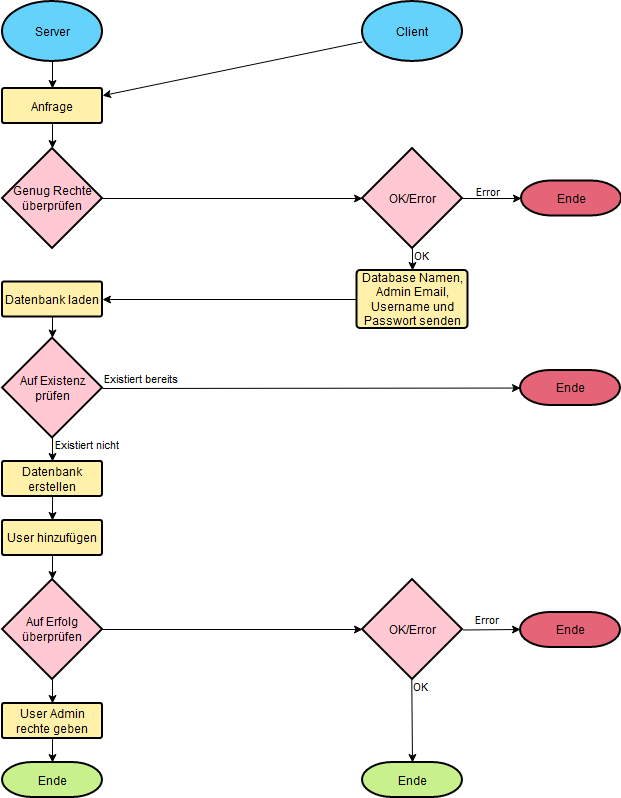
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User Admin Rechte hat, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Client sendet den Namen der Item-Collection zum Kopieren, den neuen Namen der Item-Collection und die Rechte der neuen Item-Collection
4. Der Server überprüft nun, ob es die neue Item-Collection nicht bereits gibt
5. Er wird auch überprüft, ob mit der alten Item-Collection nichts falsch ist
6. Waren alle Überprüfungen erfolgreich, so wird die alte Item-Collection dupliziert und danach zur Datenbank hinzugefügt

CreateNewDatabaseCommand

Erstellt eine neue Datenbank

Flussdiagramm:



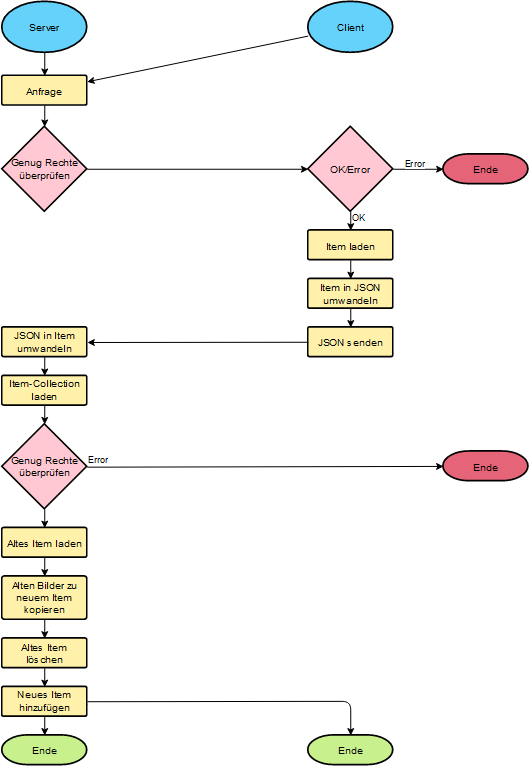
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User Super-Admin Rechte hat, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Client sendet den Namen der neuen Datenbank, die Admin-E-Mail, den Admin-Usernamen und das Admin-Passwort
4. Der Server lädt die Datenbank und überprüft, ob sie nicht bereits existiert
5. Existiert die Datenbank noch nicht, so wird sie neu erstellt und der Admin-User eingetragen
6. Die Daten des Admin-Users werden überprüft (siehe Klasse Validator)
7. War alles erfolgreich, so erhält der Admin-User noch die Admin-Rechte

EditItemCommand

Ändert ein Item, mit Ausnahme der Bilder

Flussdiagramm:



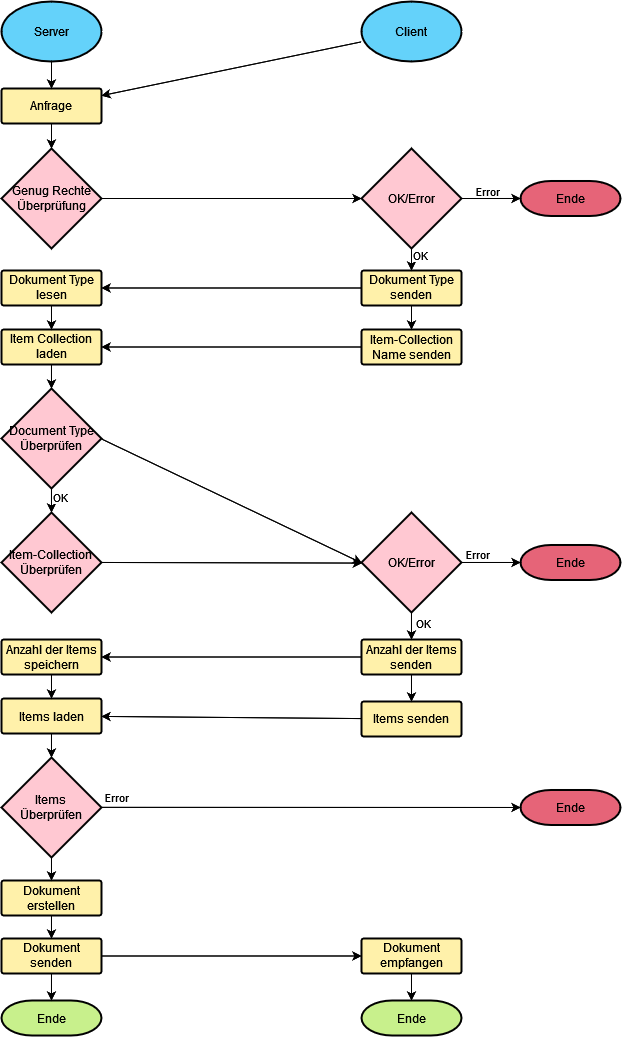
Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User das Recht hat, um Items hinzuzufügen, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Der Client lädt das Item und wandelt es in JSON um
4. Der JSON-String wird nun an den Server gesendet
5. Der Server wandelt den JSON-String wieder zu einem Item um und lädt die Item-Collection des Items
6. Nun wird auch überprüft, ob der User überhaupt genug Rechte für die Item-Collection hat
7. Hat der User genug Rechte, so lädt der Server das alte Item und kopiert die Bilder des alten Items zu dem neuen Item
8. Das alte Item wird gelöscht und das neue hinzugefügt

GeneratePDFCommand

Erzeugt PDFs für verschiedenste Dokumente

Flussdiagramm:



Funktionsweise:

1. Client stellt Anfrage
2. Server überprüft, ob der User Admin-Rechte hat, sendet entweder OK oder einen Error, wenn der User nicht genug Rechte hat
3. Der Client sendet, welches Dokument erzeugt werden soll und den Namen der Item-Collection aus der das Dokument erzeugt werden soll
4. Der Server überprüft, ob der Dokument-Type existiert und ob die Item-Collection existiert
5. Falls beide Überprüfungen erfolgreich waren, sendet der Client die IDs der Items, die in das Dokument kommen sollen
6. Der Server lädt diese Items und es wird überprüft, ob die Items existieren
7. War dies erfolgreich, so wird die PDF erstellt und an den Client gesendet

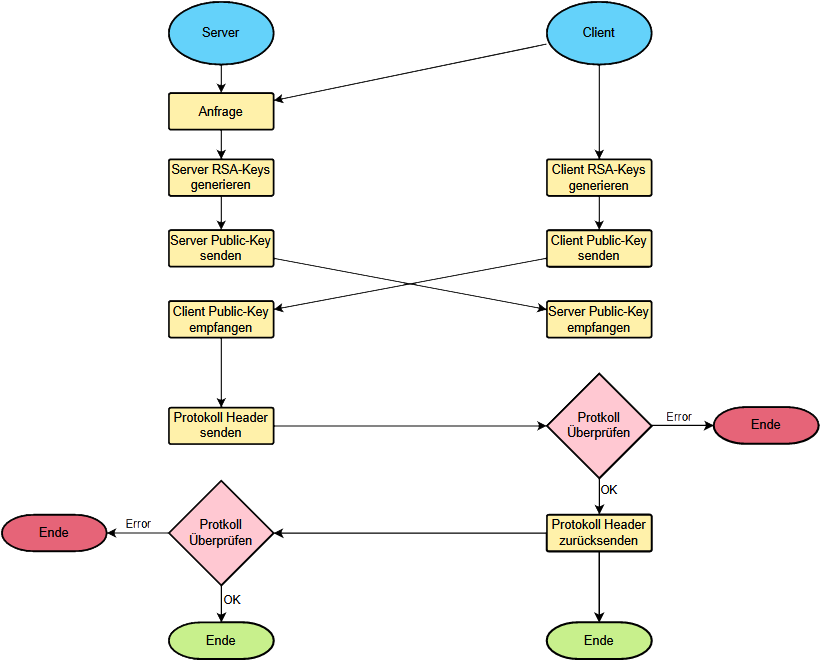
**RSA-Verschlüsselung**

RSA ist ein asymmetrisches kryptographisches Verfahren, das sowohl zum Verschlüsseln als auch zum digitalen Signieren verwendet werden kann.

Es verwendet ein Schlüsselpaar, bestehend aus einem privaten Schlüssel, der zum Entschlüsseln oder Signieren von Daten verwendet wird, und einem öffentlichen Schlüssel, mit dem man verschlüsselt oder Signaturen prüft.

Der private Schlüssel wird geheim gehalten und kann nicht mit realistischem Aufwand aus dem öffentlichen Schlüssel berechnet werden.

Dieses System wurde in dieser Diplomarbeit folgenderweise implementiert:



Erklärung

* Der Client öffnet eine Verbindung zum Server
* Der Server generiert nun seine RSA-Keys, die Erzeugung benutzt seinen eigenen Algorithmus (mehr siehe Unten)
* Der Client generiert parallel seine RSA-Keys
* Nun werden die Public-Keys ausgetauscht
* Die Public Keys können nun zum Verschlüsseln von Nachrichten verwendet werden
* Der Server sendet nun einige Header-Nachrichten an den Client, um das Protokoll zu überprüfen
* Der Client empfängt diese Nachrichten, überprüft sie und sendet seine eigenen Header-Nachrichten zurück
* Der Server empfängt sie, überprüft sie abermals
* Gab es keinen Fehler ist die Verbindung stabil und die weiter Kommunikation kann erfolgen

RSA-Erklärung

RSA basiert darauf, dass 2 Keys generiert werden, ein Public- und ein Private-Key.

Der Public-Key ist öffentlich, jeder kann ihn sehen und verwenden.  
Der Private-Key ist privat, nur der Erzeuger kann ihn verwenden.

Der Public-Key wird verwendet, um Nachrichten zu verschlüsseln. Diese verschlüsselten Nachrichten können nur mit dem zugehörigen Private-Key wieder entschlüsselt werden.

Die Gerissenheit dieser Verschlüsselung ist nun aber, dass es quasi unmöglich ist den Private-Key zu erhalten, der aber benötigt wird, um Nachrichten zu entschlüsseln.

Hat ein Angreifer den Public-Key bringt ihm das nichts, da er den Private-Key nicht kennt.

Key-Erzeugung

Der Public-Key ist ein Zahlenpaar (e, N) und der Private-Key ist ebenfalls ein Zahlenpaar (d, N), wobei N bei beiden Schlüsseln gleich ist. Man nennt N den RSA-Modul, e den Verschlüsselungsexponenten und d den Entschlüsselungsexponenten. Diese Zahlen werden durch das folgende Verfahren erzeugt:

* Wähle zufällig und stochastisch unabhängig\* zwei Primzahlen p und q. Diese sollen in der gleichen Größenordnung sein, aber nicht zu dicht beieinander liegen
* Berechne den RSA-Modul  
  N = p \* q
* Berechne die Eulesche ϕ-Funktion von N
* Wähle eine zu ϕ(N) teilerfremde Zahl e, für die gilt 1 < e < ϕ(N)
* Berechne den Entschlüsselungsexponenten d als multiplikativ Inverses (Kehrwert) von e bezüglich des Moduls ϕ(N)

stochastisch unabhängig... nicht gegenseitig beeinflussenden Zufallsereignissen

**Secure Password Storing**

Damit die gespeicherten Passwörter auch zu 100 Prozent sicher sind, wurde in dieser Diplomarbeit der neuste Sicherheits-Standard implementiert.

Die Passwörter sind zunächst einmal gehasched:

Ein normales Passwort wird zu einem Hash umgewandelt, dieser wurde mit dem Hashing Algorithmus SHA-256 möglich gemacht.

Das Spezielle an SHA-256 ist, dass das Hashen sehr schnell geht, den Hash in das ursprüngliche Passwort umzuwandeln ist aber so gut wie unmöglich.

Beispiel:

Passwort = Test123!  
Gespeichertes Passwort = Hash(„Test123!“) = d9b5f58f0b38198293971865a…

Dies ist aber noch nicht wirklich sicher, denn es gibt Tabellen, in denen die meistbenutzten Passwörter schon in ihrer gehashten Form gespeichert sind. Außerdem, falls 2 Nutzer das gleiche Passwort haben, erzeugt das Hashen den gleichen Hash und ein Angreifer kann erkennen, dass diese Personen das gleiche Passwort haben. Falls nun in einem anderen Datenbank-Leak das Passwort eines Nutzers geleaked wird, so weiß der Angreifer nun auch die Passworter der anderen Nutzer mit dem gleichen Hash.

Um dieses Problem zu lösen, salted man Passwörter. Das salten hängt einfach nur eine zufällige Zeichenkette an das Passwort des Nutzers und Hashed diese mit. Diese zufällige Zeichenkette nennt man Salt. Der Salt wird mit dem gehashten Passwort in der Datenbank gespeichert.

Beispiel:

Passwort = Test123!  
Salt = 3d5p2k9n2jj42  
Gespeichertes Passwort = Hash(„Test123!“ + „3d5p2k9n2jj42“) = e69188f95d0432453bf23fe85…

Nun gleicht kein Passwort dem anderen, dennoch gibt es noch Probleme. Hat ein Angreifer Zugang zur Datenbank, kann er ganz einfach die Salts auslesen und dann einfach die bekanntesten Passwörter durchprobieren, bis er einen Nutzer mit einem schwachen Passwort gefunden hat. Um das Ganze noch sicherer zu machen werden die Passwörter noch gepeppered. Bei diesem Verfahren wird genau wie beim salten bei jedem Passwort noch eine Zufällige Zeichenkette, der Pepper, angehängt. Diese Zeichenkette ist aber für alle User gleich und sollte möglichst geheim gehalten werden. Meist wird der Pepper irgendwo im tiefsten Inneren des Programmcodes gespeichert.

Beispiel:

Passwort = Test123!  
Salt = 3d5p2k9n2jj42  
Pepper = fj2dz4ls03da4  
Gespeichertes Passwort = Hash(„Test123!“ + „3d5p2k9n2jj42“ + „fj2dz4ls03da4“) = 9bdfa4c7120ca9a212fe8f880…

Jetzt ist die Datenbank schon sehr sicher. Damit die Datenbank geleaked werden kann, braucht man die Datenbank an sich und man muss das Programm Reverse-Engineeren um den Pepper zu finden. Hat man dies alles gefunden, ist des dennoch möglich alle Passwörter zu Brute-Forcen, das bedeutet alle möglichen Kombinationen durchprobieren.  
Dies dauert zwar recht lange, ist aber dennoch eine Sicherheitslücke.   
Um das zu verhindern, wird das Passwort nicht nur einmal, sondern unzählige Male gehashed. Kann ein Angreifer eine Million normal gehashte Passwörter in der Sekunde hashen, so kann er in unserem Fall nur 1000 oder noch weniger Passwörter in der Sekunde hashen.

Beispiel:

Passwort = Test123!  
Salt = 3d5p2k9n2jj42  
Pepper = fj2dz4ls03da4  
Gespeichertes Passwort = Hash(Hash(Hash...(„Test123!“ + „3d5p2k9n2jj42“ + „fj2dz4ls03da4“))) = f0ce007f7fb52b91ec874c294…