HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Architekturbeschreibung

Dennis Grabowski, Julius Zint, Philip Matesanz, Torben Voltmer

Masterprojekt „Entwicklung und Analyse einer sicheren Web-Anwendung“
Wintersemester 18/19

9. Dezember 2018

Inhaltsverzeichnis

1. Architekturaufbau

1.1. Klassendiagramm

Das in Abbildung B dargestellte Architekturdiagramm verzichtet zugunsten der Übersichtlichkeit auf die Abhängigkeiten zwischen den Klassen. Es dient lediglich zur Veranschaulichung der Top-Level-Architektur und zur Illustration der wichtigsten Klassen. Der Zusammenhang der farblich abgetrennten Gruppierungen wird zusätzlich textuell erklärt.

Zu jedem Controller gibt es einen Manager in der Domänenlogik. Dieser enthält jegliche Logik zur Abarbeitung der Anfrage und liefert auftretende Fehler über eine Exception zurück. Ein Datenbankzugriff erfolgt somit niemals von einem Controller, sondern immer aus dem zugehörigen Manager. Beim Sessionmanager handelt es sich um einen Spezialfall, da das Session-Konzept ein integraler Bestandteil der Anwendung ist, da die Prüfung auf vorhandene Authentifizierung Grundlage aller Aktionen ist[1].

Das Gleiche gilt für das `PasswordSecurityModule`, welches beim Login, Erstellen eines Nutzers sowie dem „Passwort zurücksetzen“ benötigt wird, da in ihr alle Passwort-Operationen gekapselt sind. Daher wurden diese beiden Klassen unter „Cross-Cutting-Concerns“ zusammengefasst. Zugriffe auf diese Klassen können sowohl von Controllern als auch von Managern in der Domänenlogik erfolgen.

Im Folgenden geben Sequenzdiagramme weitere Einsicht in das Zusammenspiel dieser Klassen.

Die Klassen, die in der zweiten und dritten Iteration dazu gekommen sind, können der Abbildung C und D entnommen werden. Der Aufbau dieser Klassendiagramme gleicht dem Aufbau des ersten.

1.2. Sequenzdiagram: Benutzer erstellen

Beim Anlegen eines neuen Benutzers wird zuerst durch ein HTTP GET das Formular angefragt. Dies ist notwendig um ein gültiges CSRF-Token zu erhalten, welches beim anschließenden HTTP POST mitgeliefert werden muss. Auch hier wird bereits, wie aus dem Sequenzdiagramm E ersichtlich, überprüft ob der anfragende User authentisiert ist. Ist das der Fall, wird ein HTML Formular zurückgeliefert, die ausgefüllt und durch eine weitere Anfrage wieder an den Server geschickt wird. Hier wird zuerst über die in Play integrierte Möglichkeit der Annotations eine erste Validierung der Eingaben vorgenommen. Somit werden Eingaben mit nicht unterstützten Zeichen oder ungültiger Länge früh herausgefiltert und dem Benutzer durch Anzeigen von Fehlermeldungen direkt am

[1] Siehe Anforderungsbeschreibung 4.2: Sämtliche Funktionen von HsH-Helfer sind nur angemeldeten Benutzern zugänglich.

Eingabefeld mitgeteilt. Sind alle Eingaben korrekt, wird der Aufruf an den Manager weitergeleitet. Dieser befragt zuerst die Policy, ob der authentisierte Benutzer autorisiert ist, einen neuen Benutzer anzulegen. Nach erfolgreicher Prüfung generiert er ein neues Passwort, hashed es und legt in einer Transaktion einen neuen Benutzer an. Das Klartext-Passwort wird zurückgegeben und dem Administrator angezeigt, um dies dem neuen Benutzer zukommen zu lassen.

1.3. Sequenzdiagram: Login

Das Login-Sequenzdiagramm F zeigt den Fall, dass der Benutzer bereits mehrfach das Passwort falsch eingegeben hat und ein Captcha lösen muss. In der Antwort, die auf die erste GET Anfrage folgt, sind sowohl das CSRF-Token als auch das HTML zum Lösen des Captchas enthalten. In der darauffolgenden POST Anfrage werden die Captcha-Information, sowie der Benutzername und das Passwort aus dem Formular mitgeliefert. Bei beiden Anfragen wird noch sichergestellt, dass der Benutzer nicht bereits eine gültige Session besitzt, um mehrfache Logins zu vermeiden. Die Eingaben werden, wie auch schon bei Nutzer erstellen, zuerst durch Annotations validiert und anschließend an den `LoginManager` weitergereicht. Diese berechnet unabhängig von der Existenz des Nutzerkontos den Hash des Klartext-Passworts und vergleicht diesen mit dem Hash, der zum Benutzerkonto abgespeichert wurde, falls vorhanden. Anschließend wird die Login-Firewall befragt, ob es Unstimmigkeiten für diesen Benutzer oder die anfragende IP-Adresse gibt. In dem hier vorgestellten Fall liefert diese `Verify` zurück, was bedeutet, dass dieser Benutzer ein Captcha lösen muss. Anschließend wird überprüft, ob es erfolgreich gelöst wurde und dann wird beim `SessionManager` durch den Aufruf von `startNewSession` eine neue Session erzeugt. Damit wurde der Login erfolgreich abgeschlossen.

1.4. Dateiaustausch

Alle Dateien, die von Benutzern hochgeladen werden, werden in der Datenbank als `BLOB` gespeichert. So können konsistente Zustände mittels Transaktionen sichergestellt werden. Außerdem ist HsHelper dadurch unabhängiger vom Dateisystem, das auf dem Host-System verwendet wird.

Dadurch, dass wir Java-seitig ein Byte-Array (`byte[]`) als Datentyp für den Dateiinhalt verwenden, besteht eine Obergrenze von $2^{31} - 1$ Bytes (2,147 Gigabyte) [2] für die Dateigröße, da die Länge sowie der Index eines Byte-Arrays als Integer verwaltet wird.

[2] Siehe 4.2.1 Integral Types and Values https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.2

Um die Umgehung des Speicherplatzlimits durch Ausnutzung des Kommentarfeldes und Dateinamens zu verhindern, zählen sowohl der Datei-Inhalt, als auch Dateiname und Kommentar zum Speicherplatz, den eine Datei belegt. Dabei nehmen wir an, dass jedes Zeichen des Dateinamens und des Kommentars jeweils 1 Byte Speicherplatz verbraucht und machen keine Annahmen über die verwendete Zeichenkodierung. Somit werden auch obskure Unicodezeichen dem Nutzer mit jeweils 1 Byte in Rechnung gestellt. Wir halten dies für eine faire Annahme, vor allem unter Berücksichtigung eines typischen Nutzers, der die Unterschiede von Unicode zu anderen Zeichenkodierungen nicht kennt.

Beim Herunterladen von Dateien wird der Content-Disposition Modus auf `attachment` und Content-Type auf `application/octet-stream` gesetzt. So wird erzwungen, dass der Browser immer den „Speichern“-Dialog anzeigt statt die Datei direkt (im Browser) darzustellen[3].

1.5. Logging

Aufgrund der Bedrohung *T11: Abstreitbarkeit illegaler Handlungen* wurde ein Logging implementiert, mit dem sichergestellt wird, dass genau das nicht passieren kann.

Im Zuge dessen wurden 2 verschiedene Logs mithilfe der Bibliothek „Logback“, die Play standardmäßig inkludiert, implementiert. Einmal ein generelles Zugriffslog, in welchem jegliche Zugriffe auf unsere Seite dokumentiert werden, und ein domänenspezifisches Log, in dem spezielle Loggingausgaben für die einzelnen, vorhandenen Aktionen persistiert werden. Das Zugriffslog wird in der Datei `logs/access.log` unter Beachtung eines angepassten *Apache Log Format* persistiert. Das Format ist wie folgt aufgebaut:

```
[Timestamp] - IPAdresse - Username "HTTPVerb HTTPPath"
```

Für den Fall, dass kein Username vorhanden ist, beispielsweise ein Angreifer oder Endpoints, die auch ohne Authentifizierung erreichbar sind, wird „unauthorized user“ geloggt.

Im domänenspezifischen Log, dem `logs/application.log`, wird nach dem vorgegebenem Format von „Logback“ persistiert. Dieses sieht wie folgt aus:

```
[Timestamp] - [LogLevel] - from JavaClass in JavaThread
domainspecificLogMessage
```

Um den Schaden eines Log Poisoning zu mitigieren, loggen wir nur die IDs der einzelnen Entitäten sowie Teile dieser Entitäten, die validiert wurden. Konkret heißt das, dass wir Benutzername, Gruppenname und Dateiname loggen. Diese werden nämlich durch die Eingabevalidierung mithilfe eines Regex geprüft, so dass nur erlaubte Zeichen durchkommen.

[3] Das wäre zum Beispiel bei Textdateien, Bildern und PDFs möglich.

Um einen weiteren Schutz zu bieten, haben wir Play's Loggerschnittstelle erweitert, so dass wir weitere, potentiell gefährliche Zeichen aus einer zu loggenden Ausgabe herausfiltern können. Bisher filtern wir nur Zeilenumbrüche heraus, damit ein Angreifer nicht in der Lage ist, ein fiktives Logstatement zu injizieren, um beispielsweise einem anderen Nutzer eine illegale Aktion unterzujubeln.

Generell loggen wir folgende Aktionen:

- Erstellen von Nutzern, Gruppen sowie Dateien
- Löschen von Nutzern, Gruppen sowie Dateien
- Änderungen an Dateikommentar, -inhalt und -Berechtigungen
- Änderungen an der Mitgliederliste einer Gruppe
- Versuch, bestehende Einschränkungen zu brechen (Speicherplatzlimit versucht zu überschreiten, Gruppenname existiert bereits...)

2. Kryptografisch relevante Informationen

Folgender Abschnitt enthält gesammelte Informationen über die kryptographischen Algorithmen, die im HsH-Helper Anwendung finden. Die für die Algorithmen benötigten Geheimnisse und deren Verwaltung wird ebenfalls hier dokumentiert, da die Sicherheit von der Geheimhaltung dieser abhängt.

2.1. Passwörter

Die Passwörter, die Benutzer verwenden um sich beim HsH-Helper zu authentisieren, werden grundsätzlich nur als Hash persistiert. Konkret wird die Hashfunktion bcrypt zum Hashen der Passwörter verwendet.

Um ein Passwort mit bcrypt zu hashen, muss zunächst ein Salt generiert werden. Neben dem Salt und dem gehashten Passwort wird bei bcrypt außerdem noch die bcyrpt Version und die Anzahl der Runden gespeichert. Über die Anzahl der Runden kann die Laufzeit, die bcrypt benötigt, um ein Passwort zu hashen, beeinflusst werden.

Die verwendete Bibliothek ist jBCrypt von mindrot.org (org.mindrot.jbcrypt) in der Version 0.4. Sie verwendet intern die Java Klasse SecureRandom als CSPRNG.

Auch zur Generierung der initialen (und temporären) Benutzer-Passwörter beim Anlegen eines neuen Benutzerkontos wird SecureRandom als Basis verwendet. Dafür zuständig ist die Klasse `PasswordGenerator`. Die generierten Passwörter können die Zeichen `a-z`, `A-Z`, `0-9` und die Sonderzeichen `!%?#-_*+` enthalten.

Wir prüfen bei einer Nutzer-Initiierten Passwortänderung, ob das entsprechende Klartextpasswort „schwach“ ist. Wir definieren ein schwaches Passwort als eines, das in einer Blacklist der Anwendung gelistet ist. Die von uns ausgelieferte Blacklist basiert auf einer im Internet erhältlichen Liste der am häufigsten verwendeten Passwörter [1] und umfasst eine Million Einträge. Bei der Initialisierung der Anwendung laden wir diese Liste in ein HashSet und nutzen dieses zur effizienten Abfrage. Die hierzu genutzte contains() Methode läuft in konstanter Zeit bzw. O(1) [2]. Durch eine missbräuchliche Nutzung dieser Funktionalität würde sich so kein DOS-Vektor ergeben.

[1] https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt
[2] https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

2.2. CSRF-Tokens

Die CSRF-Tokens, die das Play Framework generiert, bestehen aus 3 Teilen:

1. Einem Token, dass für jede Session mit der Java Klasse SecureRandom als CSPRNG generiert wird.
2. Einer auf der Systemzeit basierendem Nonce, dass für jeden Request generiert wird.
3. Einer auf HMAC-SHA1 basierenden Signatur des Tokens und der Nonce.

Beim Starten einer Play-Session wird ein CSRF-Token in das Session Cookie geschrieben. Dieses CSRF-Token ändert sich während einer Play-Session nie. Für jedes Formular wird ein eigenes CSRF-Token generiert, dass sich innerhalb einer Play-Session nur durch die Nonce (und damit auch der Signatur) unterscheidet. Durch Vergleichen des Tokens aus dem CSRF-Token der Play-Session und des Tokens aus dem CSRF-Token in einem HTTP Request kann die Gültigkeit eines CSRF-Token festgestellt werden.

2.3. Play Cookies

Das Play Framework nutzt zur Persistierung eines Cookies in dem Client-Browser das „JSON Web Token“-Format. Ein JWT besteht aus 3 Teilen: einem Header, einer Payload sowie einer Signatur, siehe A. Jeder dieser Teile ist ein namenloses JSON-Objekt. Innerhalb des Headers ist gelistet, welcher Algorithmus genutzt wurde, um das JWT zu signieren. Die Payload ist ein einfaches JSON-Objekt, in welchem Daten gespeichert werden können, die zu signieren sind. Diese Daten werden innerhalb des JWT-Formats als „Claims“ bezeichnet. Zusätzlich werden innerhalb dieser Payload persistiert, zu welchem Zeitpunkt das JWT erstellt wurde, ab welchem Zeitpunkt es gültig ist und wann es ausläuft. Für die Signatur werden Header sowie Payload durch „base64url“ kodiert und dann zusammen mit einem Punkt als Trennelement konkateniert. Die dadurch entstehende Zeichenkette wird zusammen mit einem geheimen Schlüssel an den kryptografischen Signatur-Algorithmus weitergegeben, der zuvor im Header genannt wird. Play verwendet standardmäßig zur Signierung den HMAC-SHA256 Algorithmus. Dadurch kann die Integrität der Claims geprüft werden und zeitgleich können sie authentifiziert werden. Um das JWT nun bilden zu können, werden alle 3 Teile durch „base64url“ seperat kodiert und durch einen Punkt getrennt und konkateniert: `encodeBase64Url(header) + '.' + encodeBase64Url(payload) + '.' + encodeBase64Url(signature)`.

Bei Werkseinstellungen verwendet Play ein Cookie namens *PLAY_SESSION*, in dessen Payload ein zusätzliches JSON-Objekt namens `data` sowie die Zeitstempel für Erstellungsdatum sowie Beginn der Gültigkeit gespeichert sind. In dem JSON-Objekt `data` persistiert Play das CSRF-Token, erlaubt einer Applikation allerdings dort auch Daten abzulegen. Wie später zu sehen ist, baut unser Session-Konzept darauf auf.

Neben dem Session Cookie gibt es noch das Flash Cookie *PLAY_FLASH*, was allerdings in der aktuellen Iteration keinerlei Anwendung findet.

Die Dokumentation von Play vermerkt, dass das Flash Cookie nicht signiert wird:

> The Flash scope works exactly like the Session, but with two differences:
> - data are kept for only one request,
> - the Flash cookie is not signed, making it possible for the user to modify it.
> [1]

Dies hat sich allerdings bei einem Test als falsch herausgestellt. Das Flash Cookie wird in unserer Anwendung sowohl signiert als auch validiert, wenn es vom Client bei einer weiteren Anfrage wieder an den Server geschickt wird.

2.4. Schlüsselmanagement

Für das System sind mehrere Schlüssel und Zugangsdaten notwendig:

- Der private Schlüssel zum Signieren der Play Cookies
- Der private Schlüssel für reCAPTCHA
- Der öffentliche Schlüssel für reCAPTCHA
- Zugangsdaten für den E-Mail Server
- Zugangsdaten für die Datenbank

Der öffentliche reCAPTCHA-Schlüssel wird benötigt, um das reCAPTCHA einbinden zu können und wird vom Client an Google übertragen. Löst der Client ein Captcha korrekt, erhält er von Google eine Nonce, die er an uns weiterleitet. Diese wird von unserem Server in Verbindung mit dem Private Key an Google geschickt. Google teilt uns dann mit, ob diese Nonce valide ist.

Der öffentliche Schlüssel ist fest mit „unserer Domain“ (localhost) verbunden. Das heißt, dass der Captcha-Code nicht auf einer anderen Domain eingebunden werden kann, um Noncess zu erzeugen.

Daher werden alle Schlüssel sowie Zugangsdaten in der `secrets.conf` Datei auf jedem Entwicklungsrechner der Teammitglieder persistiert, welche nicht versioniert wird. Die `application.conf` bindet diese Datei mit dem `include`-Statement ein und somit werden die Schlüssel in der Anwendung verwendet. Dies hat den offensichtlichen Vorteil das Zugangsdaten nicht im Repository einsichtlich sind, aber Anwendungseinstellungen jedoch schon.

3. Sicherheitsmaßnahmen

3.1. Login-Firewall

Der Login einer Web-Anwendung ist potentiell anfällig für Brute-Force-Angriffe, die darauf abzielen, eine valide Kombination aus Benutzername und Passwort zu finden. Aus diesem Grund muss die Anwendung fehlerhafte Zugriffsversuche protokollieren und wenn eine Häufung selbiger auftritt, Maßnahmen ergreifen, um den Angriff zu verlangsamen oder gar zu stoppen. Die zu ergreifende Maßnahme sollte so gewählt werden, dass sie nicht selbst einen Angriffsvektor darstellt, wie z.B. das vollständige Sperren eines Accounts nach N falschen Logins - man könnte so legitime Nutzer vorsätzlich an der Benutzung der Anwendung hindern (Denial of Service).

Aus diesem Grund verfügt unsere Anwendung über eine Komponente[1], die falsche Logins protokolliert und eine der folgenden Maßnahmen ergreift:

- Spezifischen Benutzer-Account in Captcha-Mode versetzen
- Spezifische IP-Adresse in Captcha-Mode versetzen
- Spezifische IP-Adresse von Logins ausschließen

Beim Captcha-Mode handelt es sich um eine Maßnahme, bei der ein Login lediglich möglich ist, wenn zugleich ein Google reCAPTCHA gelöst wird. Eine vollständige Sperrung eines Accounts findet durch unsere Firewall so nicht statt. Es wurde die Annahme getroffen, dass das automatisierte Lösen der reCAPTCHAs zu aufwändig & teuer für einen Angreifer ist. Zugleich ist das „unnötige“ Lösen eines Captchas für das Opfer ein vertretbares Hindernis.

Lediglich IP-Addressen werden als schärfste Maßnahme vollständig von Logins ausgeschlossen. Sofern ein Angreifer nicht die gleiche IP-Adresse wie sein Opfer verwendet, kann er sich höchstens selbst vom Dienst ausschließen. Sollte eine Situation vorliegen, bei der sich eine Vielzahl an Nutzern eine IP-Adresse teilt, besteht die Möglichkeit, diese von der vollständigen Sperre auszunehmen [2].

Die interne Datenstruktur der Login Firewall ist darauf optimiert, möglichst wenig Speicher zu verwenden und nutzt Buckets, um aggregierte Informationen über Login-Versuche zu speichern. Die Buckets bilden ein fixes Zeitinterval ab. Führt ein Angreifer eine Million fehlerhafte Logins in zehn Minuten durch, werden lediglich zwei Datenbank-Einträge erstellt: Einen mit Bezug auf den betroffenen Nutzer-Account und einen mit Bezug auf die IP des Angreifers. Beide Einträge verfügen über eine Zählvariable, die jeweils die Anzahl der fehlerhaften Logins beinhaltet.

Der Verweis auf den Nutzer-Account findet über die numerische und eindeutige ID statt, über die jeder Account verfügt. Jener Fallgestaltung, bei der ein Login bei einem

[1] policy.ext.loginFirewall
[2] siehe conf/ip_whitelist.txt

nicht-existenten Account erfolgt, wurde ebenfalls Rechnung getragen. Hierbei ist es erforderlich, dass die Firewall exakt so funktioniert, wie bei existierenden Accounts, um ein Informationsleck zu verhindern: Würde die erste Maßnahme nie oder anders greifen als bei tatsächlich existenten Accounts, wäre es möglich zu prüfen, ob ein Account beziehungsweise der dazu korrospondierende Benutzername tatsächlich existiert[3].

Zu diesem Zweck wird bei nicht-existenten Accounts der Benutzername mittels Hashing auf eine „virtuelle“ User-ID gemappt. Die User-ID wird über ein Long repräsentiert, es wird jedoch ausschließlich der positive Zahlenraum verwendet. Vom Username wird der MD5-Hash gebildet, die ersten 8 Byte werden als Long verwendet und gegebenfalls invertiert. Der so resultierende negative Long-Wert wird in diesem Fall als Grundlage für die Protokollierung fehlerhafter Logins verwendet. Dass MD5 kryptografisch bereits längst als geknackt gilt, ist irrelevant. Wir machen uns lediglich die gute Streuung und hohe Geschwindigkeit von MD5 zu eigen.

3.2. Session-Konzept

Integraler Bestandteil unserer Anwendung ist eine Session-Funktionalität. Theoretisch bietet Play bereits eine Implementierung unter Verwendung von JWT mit der es möglich ist, Daten in Form von signierten Cookies im Client-Browser zu speichern (vgl. 2.3). Die Sicherstellung der Authentifizierung ist jedoch für nahezu jede Funktionalität unserer Anwendung von Bedeutung. Aus diesem Grund wurde das Session-Konzept soweit abstrahiert, dass eine Java-Annotation über einer Controller-Klasse oder einzelnen Controller-Methoden ausreicht, um zu garantieren, dass ausschließlich authentifizierte Nutzer auf selbige zugreifen können.

Die Annotation bewirkt, dass der korrospondierende Request vor Erreichen der dazugehörigen Controller-Methode abgefangen wird und einer Prüfung unterzogen wird, ob der Absender authentifiziert ist. Ist dies nicht der Fall, wird die weitere Ausführung abgebrochen und der Nutzer auf die Login-Seite weitergeleitet. Ansonsten wird der Request an den entsprechenden Controller weitergereicht.

Zur Umsetzung dieser Funktionalität setzen wir auf dem Session-Konzept von Play auf. Initialisiert ein Nutzer eine neue Session, so persistieren wir seine IP-Adresse, seine User-ID, das Initialisierungsdatum und eine kryptografisch sicheren „Universally unique identifier (UUID)“ als Identifikator dieser neuen Session in unserer Datenbank. Um diese Session sicher beim Client speichern zu können, verwenden wir das bereits genannte Cookie *PLAY_SESSION*. Innerhalb des Payloads dieses JWT befindet sich ein JSON-Objekt namens „data“ hinterlegt, zu welchem man weitere Applikationsdaten als Schlüssel-Wert-Paare hinzufügen kann. Das machen wir uns zu eigen und persistieren das Schlüssel-Wert-Paar („HsHSession“: „UUID“), durch welches wir den Nutzer authentisieren können. Wir verlassen uns daher nicht auf die Existenz dieses Cookies,

[3] Diese Angriffsform nennt man auch „Side-Channel Attack“

sondern nutzen dieses nur, um einen signierten Speicher innerhalb des Client-Browsers verwenden zu können.

Die UUID selbst dient als 2nd Layer of Defense: Sollte die JWT-Signatur geknackt werden, muss der Angreifer noch eine in der Datenbank enthaltene UUID erraten. Selbst wenn er diese Hürde meistert, muss er noch die dazugehörige IP-Addresse kennen, sowie über diese auf unsere Anwendung zugreifen können und dies in einem engen Zeitfenster schaffen, da die Lebenszeit einer Session begrenzt ist. Wie lange eine Session gültig ist kann von HsHelper Benutzern eingestellt werden. Die Einstellung wirkt sich auf alles Sessions eines Benutzers aus. Maximal ist eine Session 24 Stunden gültig.

Die Anwendung verfügt über eine Ansicht, die dem Nutzer sämtliche eigene, „aktive“ Sessions anzeigt und ihm die Möglichkeit bietet, diese nach Bedarf einzeln zu entfernen. Ein Logout-Vorgang führt ebenfalls zur Entfernung der korrospondierenden Session. Um eine Vermüllung der Datenbank zu vermeiden, läuft im Hintergrund ein Prozess, der in festen Zeitintervallen Sessions aus der Datenbank entfernt, die „veraltet“ sind, denn wenn der Benutzer sich nicht ausloggt, sondern einfach den Browser schließt, kann unsere Anwendung keine Kenntnis davon erlangen.

3.3. Content Security Policy

Play verwendet standardmäßig eine sehr restriktive Content Security Policy, die XSS-Angriffe maßgeblich erschwert: Inline-Javascript wird vom Client-Browser grundsätzlich blockiert, es ist lediglich möglich, ein externes Script in eine HTML-Seite einzubinden, das jedoch von der gleichen Top-Level-Domain stammen muss. Zugleich muss dieses Script explizit Javascript als Content-Type im HTTP-Header benennen.

Da unsere Anwendung zum jetzigen Zeitpunkt nur mittelbaren Gebrauch von JavaScript macht, haben wir uns dazu entschlossen, die Content Security Policy zu verschärfen: JavaScript wird grundsätzlich nicht mehr ausgeführt, es sei denn, es handelt sich um die JavaScript-Libraries von reCAPTCHA. Zu diesem Zweck haben wir die entsprechenden Quell-URLs auf die Whitelist gesetzt und somit gleichzeitig die Annahme getroffen, dass wir Google als Host-Provider der Bibliotheken vertrauen[4].

Ebenso haben wir uns entschlossen, Browser, die Content Security Policy nicht unterstützen, von unserer Anwendung auszuschließen. Hiervon sind nur hinreichend alte Browser betroffen[5].

[4] Erforderliche Urls für die CSP Whitelist: https://developers.google.com/recaptcha/docs/faq
[5] Siehe „Browser compatibility“ https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

3.4. Verschlüsselung von Netzwerkdienst-Zugangsdaten

Um zu verhindern, dass irgendjemand Zugriff auf die gespeicherten Netzwerkdienst-Zugangsdaten erhält, werden diese verschlüsselt gespeichert. Die Verschlüsselung erfolgt faktisch mit dem Klartext-Nutzerpasswort als Secret.

Für jeden Nutzer wird bei Erstellung ein kryptografisch sicherer Schlüssel erzeugt ($CredentialKey^{Plaintext}$). Dieser Schlüssel wird mit einem aus dem Klartext-Passwort des Nutzers generiertem Schlüssel verschlüsselt. Unsere Anwendung speichert den resultierenden $CredentialKey^{Ciphertext}$.

Meldet sich ein Nutzer an, wird $CredentialKey^{Ciphertext}$ mittels des beim Login eingegebenen Klartext-Passworts entschlüsselt. Gleichzeitig wird von unserer Anwendung ein kryptografisch sicheres Secret erzeugt. Wir verschlüsseln $CredentialKey^{Plaintext}$ erneut mit dem erstellten Secret und speichern den resultiernden Ciphertext in der Datenbank in Verbindung mit der Session. Das Secret wird auf PC des Nutzers im Cookie gespeichert und von unserer Anwendung verworfen.

Hat sich ein Nutzer angemeldet, kann mittels des Session-Secret und dem der Session zugehörigen Datenbankeintrag der $CredentialKey^{Plaintext}$ entschlüsselt werden. Ein erneuter Zugriff auf das Klartext-Passwort des Nutzers ist nicht erforderlich. Erhält ein Angreifer Zugriff auf das im Cookie gespeicherte Session-Secret, ist dieses nur in Verbindung mit der Session verwendbar. Timed die Session aus, ist das Session-Secret wertlos. Dem Nutzer wird niemals der eigentliche Credential Key offenbart.

Diese Implementierung hat den Vorteil, dass die Änderung des Passworts keine Neu-Verschlüsselung der dazugehörigen Netzdienst-Zugangsdaten erfordert. Es muss lediglich der Credential Key mit dem alten Passwort ent- und mit dem neuen Passwort verschlüsselt werden. Vorhandene Sessions funktionieren weiterhin. Ein Passwort-Reset ist allerdings nicht ohne Datenverlust möglich: Bei diesem liegt das Klartextpasswort nicht vor, eine Entschlüsselung des Credential Keys ist somit nicht möglich, er muss neu generiert werden und mit dem neuen Passwort verschlüsselt werden. Vorhandene Netzdienst-Zugangsdaten können folglich nicht mehr entschlüsselt werden und werden somit im selbigen Prozess entfernt.

Der von Nutzer durchzuführende Single Sign-On erfolgt asynchron: Lädt der Benutzer die Seite mit seinen Zugangsdaten und den Login-Buttons, werden keine Zugangsdaten entschlüsselt. Dies erfolgt ausschließlich, wenn ein Nutzer auf einen Login-Button klickt. In diesem Fall wird die Ausführung des Klick-Events unterbrochen, die entsprechenden Zugangsdaten werden von der Anwendung abgefragt und in die Form des Login-Buttons eingefügt. Erst dann wird die Ausführung des Klick-Events fortgesetzt.

Als Chiffre wird AES-256 im CBC Modus mit PKCS5 Padding verwendet. Dazu gibt es eine native Java Implementierung.

Um aus dem Passwort des Benutzers einen kryptografisch sicheren Schlüssel zum verschlüsseln des CredentialKey zu generieren wird PBKDF2 benutzt. Die Rundenzahl

beträgt 65536. Der Salt ist 8 byte Lang und wird mittels SecureRandom generiert. Auch dazu gibt es eine native Implementeriung in Java.

3.5. Eingabevalidierung

Für einige Parameter gibt es festgelegte Constraints, die in allen entsprechenden Dtos verwendet werden. Das sind:

- username
  - Maximale Länge: 20
  - Regex: [a-z][a-z0-9.]+
- password
  - Minimale Länge: 6
  - Maximale Länge: 72 (Beschränkung von bcrypt)
  - Darf nicht in Blacklist enthalten sein (conf/password_blacklist.txt)
- groupname
  - Minimale Länge: 3
  - Maximale Länge: 20
  - Regex: [a-zA-Z][a-zA-Z0-9.]+
- email
  - Regex: Siehe Play Implementierung von `EmailValidator`
- filename
  - Maximale Länge: 40
  - Regex: [a-zA-ZäüöÄÜÖß0-9.,#%+&!“:; -]+

3.6. Ausgabecodierung

Die Zeichen, die ein HsHelper Benutzer in das Kommentarfeld einer Datei eintragen kann, unterliegen keiner Beschränkung. Aus diesem Grund müssen Sonderzeichen maskiert werden. Wenn in Play Daten vom Typ String in einem Template verwendet werden, geschieht die Maskierung der Zeichen automatisch. So wird zum Beispiel der String `<script>` durch `&lt;script&gt;` ersetzt.

4. Implementierte Zusatzfunktionalität

4.1. Nutzer kann mit seinen aktiven Sessions interagieren

Im Sinne von Ka-Ping Yee, der innerhalb seines Papers „User interaction design for secure systems“ [3] zehn Designprinzipien beschrieben hat, die Benutzer bei Bedienung eines sicheren Systems benutzerfreundlich unterstützen sollen, haben wir mithilfe dieser neuen Funktionalität die Designprinzipien „Revocability“ und „Visibility“ abgebildet.

Ein Nutzer ist in der Lage, seine eigenen, aktiven Sessions in tabellarischer Form zu betrachten. Während dieser Ansicht kann der Nutzer einzelne Sessions invalidieren, sofern er es für nötig hält. Wir halten das für eine nützliche Zusatzfunktionalität, da ein Nutzer unsere Applikation von verschiedenen Geräten nutzen kann.

4.2. Nutzer kann vorherige Logins betrachten

Da wir bereits anzeigen, welche Sessions gerade noch aktiv sind, und sich ein Nutzer von verschiedenen Geräten zeitgleich einloggen kann, ist es sinnvoll, dem Nutzer die Möglichkeit zu verschaffen, ihm seine bisherigen, erfolgreichen Logins anzuzeigen. Dadurch ist er in der Lage, ggf. Logins von anderen Geräten sowie User-Agents zu identifizieren, die er weder autorisiert noch eventuell selber getätigt hat. So ist ihm auch möglich, eine Session, die gegebenfalls von einer bösartigen Quelle kommt, zu identifizieren. Ähnlich wie die zuvor genannte Zusatzfunktionalität soll diese Funktionalität dazu dienen, dem Nutzer zu ermöglichen, mehr Information und somit mehr Klarheit über die Aktivität seines Kontos zu erhalten. Auch diese Funktionalität bildet das Designprinzip „Visibility“ ab.

4.3. Nutzer kann einsehen, welcher Nutzer zuletzt eine Datei überschrieben hat

Dadurch, dass jeder Nutzer, der Schreibrechte für eine Datei besitzt, diese auch überschreiben kann, ist es nicht klar ersichtlich, von wem der Dateiinhalt tatsächlich geschrieben wurde. Aus diesem Grund speichern wir zusätzlich für jede Datei, wer den tatsächlichen Inhalt geschrieben hat. Diese Information verwenden wir in Form einer visuellen Darstellung: Dateien, die von einem selbst geschrieben wurden, sind grün hinterlegt („vertrauenswürdig“). Dateien, die von Dritten geschrieben wurden, werden rot hinterlegt („potentiell gefährlich“). Zugleich wird auf der Detailseite der entsprechenden Datei angezeigt wer diese geschrieben hat und wann dies erfolgte. Über diese Funktionalität

wird zugleich Threat (Freigabe einer Datei mit gleichem Namen) mitigiert. Eine solche Datei wäre rot gefärbt und somit deutlich von der eigenen, namensgleichen Datei zu unterscheiden.

4.4. Warnung bei Aktionen mit schweren Seiteneffekten

Die Aktionen „Benutzer löschen“ und „Gruppe löschen“ haben schwerwiegende Seiteneffekte: Wird ein Nutzer gelöscht, werden seine Dateien und Gruppen sowie sämtliche diesbezügliche Freigaben unwiederruflich entfernt. Das gleiche gilt für die Löschung einer Gruppe und diesbezügliche Freigaben. Diese Aktion war mittels eines Mausklicks möglich, es fand keinerlei Warnung statt. Eine solche Implementierung begünstigt Fat-Finger-Fehler[1]. Aus diesem Grund haben wir uns entschieden von dem Benutzer eine explizite Bestätigung zu verlangen. Die Auswirkungen seiner geplanten Aktion werden ihm ausdrücklich angezeigt.

4.5. Nutzer kann sein Passwort ändern

Ein angemeldeter Benutzer kann sein Passwort ändern. Dazu muss er sein derzeitige Passwort und zwei mal das gewünschte neue Passwort eingeben. Diese Funktion wurde implementiert, da ein Benutzer sein Passwort sonst nur über die „Passwort vergessen“-Funktion ändern kann.

4.6. Suchfunktionalität

Im Rahmen der Design-Erstellung haben wir das Bedürfnis einer Suchfunktionalität entdeckt. Die Dateien sind die Zentralen Elemente des HsH-Helpers und sollten so einfach wie möglich zugänglich sein. Folglich haben wir uns dazu entschlossen, in unser Design ein Suchfeld zu implementieren, das von jeder Stelle der Anwendung aus zugänglich ist.

4.7. Einstellbares Session-Timeout

Im Zuge unserer Session-Implementierung wurde ein Timeout eingeführt: Eine Session sollte nur für einen begrenzten Zeitraum gültig sein. Dieser Timeout wurde vom Entwickler willkürlich gesetzt und sollte im Rahmen einer späteren Diskussion angepasst werden. Im Laufe der Diskussion stellten wir fest, dass die Bestimmung eines Werts schwierig ist. Wir haben uns dazu entschlossen, dem Nutzer selbst einen gewissen Grad

[1] https://en.wikipedia.org/wiki/Fat-finger_error

an Freiheit einzuräumen: Er darf in einem fest definierten Interval ein eigenes Timeout wählen, das für die eigenen Sessions gilt (5 Minuten - 1 Tag). Der Default-Wert ist das zulässige Minumum.

4.8. Zusätzliche Sichten

Gemäß der Anforderungsbeschreibung wurde lediglich eine Sicht zum Listen von Dateien verlangt: Sie soll anzeigen, welche von anderen hochgeladenen Dateien der aktuelle Nutzer lesen oder schreiben darf. Die Beschränkung auf diese Sicht ist aus Usability-Aspekten nicht ausreichend: Der Benutzer sollte zumindest noch die eigenen Dateien sehen können. Folglich haben wir die folgenden Sichten zusätzlich implementiert:

- Alle Dateien, die aktuelle Benutzer lesen und/oder schreiben kann.
- Alle Dateien, die dem aktuellen Benutzer gehören.
- Alle Dateien, die dem aktuellen Benutzer freigegeben wurden (d.h. keine eigenen Dateien sind).
- Alle Dateien, die der aktuelle Benutzer Dritten freigegeben hat inkl. Berechtigungen.
- Alle Dateien, die einer bestimmten Gruppe zugeordnet sind.

Ergänzend zu den Datei-Bezogenen Sichten haben wir noch folgende Sichten implementiert:

- Alle Benutzer, die Administrator-Privilegien haben.
- Alle Gruppen, deren Owner der aktuelle Benutzer ist.
- Details zum Quota-Verbrauch des aktuellen Nutzer aufgeschlüsselt nach Kategorien.

4.9. Zwei Faktor Authentifizierung

Um die Sicherheit der Anwendung weiter zu erhöhen hat sich das Team dazu entschieden Zwei-Faktor-Authentifizierung zu unterstützen. Hierbei handelt es sich um ein Time-Based-One-Time-Password, welches in RFC 6238 [2] definiert wurde. Der Nutzer kann dazu in seinen persönlichen Einstellungen, durch Einscannen des Barcodes, das Shared-Secret in seine mobile Anwendung importieren. Ein weiterer Klick auf „Aktivieren“ legt fest, dass ab der nächsten Anmeldung, zusätzlich zum Passwort, das Einmalpasswort mit angegeben werden muss. Nach Absprache mit dem Projektleiter wurde sich dazu

entschieden, das Administratoren die Zwei-Faktor-Authentifizierung für Nutzer deaktivieren können auch wenn dies die gewonnene Sicherheit wieder etwas abschwächt. Für Nutzer die nicht auf einen Administrator zurückgreifen möchten, besteht die Möglichkeit ein Backup des Geheimnisses zu erstellen, indem Sie den QR-Code ausdrucken und diesen an einem sicheren Ort aufbewahren. Der Nutzer hat ebenso wie der Administrator die Möglichkeit die Zwei-Faktor-Authentifizierung wieder zu deaktivieren.

5. Verworfene Entwürfe

5.1. Trennung der Zuständigkeiten des Controllers

Innerhalb unserer ersten, prototypischen Implementation hatten wir einen Ansatz verfolgt, bei dem unsere `Controller`-Klassen für das Bearbeiten der Requests, das Validieren der Formulare, das Weiterleiten von Fehlern innerhalb eines Formulars, die Autorisierungsprüfung, die Ausführung der domänenspezifischen Operation sowie das Absenden des generierten HTMLs zuständig war. Auch wenn uns bewusst war, dass dies nur ein Ansatz für das Prototyping war, ist schnell offensichtlich geworden, dass dieser uns langfristig gesehen Probleme bereiten würde. Die Testbarkeit solcher Controller-Methoden litt schwer, da es kompliziert war, für einen einzelnen Unit-Test alle Klassen zu mocken, die benötigten Voraussetzungen zu erstellen, und zu prüfen, ob die richtigen Nachbedingungen gelten. Zusätzlich musste jeder dieser Tests entweder gegen eine `FakeApplication` oder gegen ein laufendes Testsystem laufen.

Um diese Testbarkeit wieder zu gewinnen, haben wir die Zuständigkeiten feiner aufgeteilt und die `Manager`-Klassen ins Leben gerufen, die von nun an die domänenspezifischen Operationen übernehmen. Die `Controller` beschäftigen sich seither nur noch mit dem Bearbeiten der Requests, dem Weiterleiten von Fehlern innerhalb eines Formulars zum Nutzer sowie dem Absenden des generierten HTMLs.

5.2. Session-Konzept / Datenschutz

Aufgrund von Datenschutzbedenken haben wir zunächst die Speicherung der mit einer Session-Verknüpften IP-Addresse im JWT-Cookie selbst forciert. Wir waren der Annahme, die IP Addresse sei ein personenbezogenes Datum und die Speicherung von selbigen in unserer Datenbank zumindest fragwürdig. Die Verlagerung der Speicherung auf den Cookie-Speicher des Client-Browsers wäre eine potentielle Umgehungsstrategie für diese rechtliche Problematik gewesen: Die IP-Adresse wird zwangsläufig vom Nutzer bei jedem Request „übertragen“. Herr Prof. Dr. Peine hat uns jedoch mitgeteilt, dass Datenschutzbedenken für dieses Projekt nicht von Bedeutung sind. Aus diesem Grund wurde diese Idee verworfen, wir speichern die IPs in unserer Datenbank.

5.3. Session-Konzept / Numeric IDs

Ursprünglich verfügte eine in unserer Datenbank abgebildete Session über eine numerische ID, die inkrementell aufsteigend war. Das JWT-Token beinhaltet den Schlüssel der korrospondierenden Session-Entität in der Datenbank. Würde ein Angreifer die JWT Signatur knacken, könnte er leicht passende IDs erraten: Er müsste sich anmelden,

wüsste dann, welche ID „aktuell“ ist und könnte anhand dieser Information passende IDs ausprobieren. Wir haben diese Implementierung zugunsten von kryptografisch sicheren UUIDs verworfen, die ein „second layer of defense“ darstellen (vgl. 3.2).

5.4. Asynchroner Datei-Upload

Aufgrund von Usability-Erwägungen wurde ein asynchroner Datei-Upload mittels Javascript angepeilt: Es ist eine Vorgabe des Dozenten, dass beim Upload zugleich Dateiname und Kommentar sowie die Berechtigungen angegeben werden müssen. Treten bei diesen Daten Fehler auf, ist es erforderlich, die entsprechende Datei erneut auszuwählen und hochzuladen. Um diesen Makel zu verhindern, war es angedacht, dass das Auswählen einer Datei unmittelbar einen Upload inkl. Fortschrittsanzeige in Gang setzt. Die Daten würden von unserer Anwendung temporär vorgehalten werden und im Formular mittels eines Hidden-Inputs eingebunden. So wäre selbst bei fehlerhaften Formulareingaben kein mehrfach-Upload erforderlich. Diese Idee wurde verwurfen: Herr Prof. Dr. hat ein dertiges Konzept als zu komplex bewertet und uns mitgeteilt, dass die Usability bei einer erzwungenden Mehrfachauswahl der Datei nicht negativ beeinträchtigt wäre.

5.5. Aufteilung von Datei-Upload und Rechtevergabe

Der Datei-Upload sollte ursprünglich keine unmittelbare Rechtevergabe beinhalten. Ein Upload sollte in einer Datei resultieren, über die ausschließlich der Owner verfügen kann bzw. das alleinige Nutzungsrecht verfügt. Das Rechtemanagement sollte ihm nach dem Dateiupload möglich sein. Im Gespräch mit Herrn Prof. Dr. Peine wurde klar, dass eine solche Umsetzung die Anforderungen nicht erfüllt. Es wurde ausdrücklich von ihm gefordert, dass der Upload in einem Request erfolgt und dieser die Datei sowie die korrospondierenden Rechte beeinhalten muss. Folglich wurde das Konzept verworfen.

5.6. Darstellung der Dateianzahl von Nutzern

Auf der Warnseite, die beim Löschen eines Nutzers angezeigt wird (vgl. 4.4) wurde ursprünglich auch die Anzahl jener Dateien angezeigt, die dem entsprechenden Benutzer gehören. Herr Prof. Dr. Peine hat uns darauf hingewiesen, dass ein Administrator über keine Berechtigung verfügt, die es ihm gestattet eine derartige Information zu erlangen. Die Anzahl der Dateien wurde durch eine allgemeingültige Umschreibung ersetzt („..alle Dateien des Nutzers..“).

5.7. Manager API ohne Übergabe des aktuellen Benutzers

In der ersten Iteration erforderte die Schnittstelle der Manager eine Übergabe der Id des aktuell angemeldeten Benutzers. Dies ist nicht nur umständlich, da es bei den meisten Aufrufen geschehen muss (abgesehen von Spezialfällen wie Login), sondern auch relevant für die Sicherheit der Anwendung. Wird beim Aufruf fälschlicherweise nicht die Id des aktuellen Benutzers übergeben, kann es so zu einer Autorisierungsverletzung kommen. Nach der Überarbeitung beziehen alle Manager den aktuellen Benutzer aus dem Session Manager und somit wird diese Art von Fehlern automatisch vermieden.

5.8. Passwort-Zurücksetzen durch unmittelbares setzen eines neuen temporären Passworts

Die Passwort-Zurücksetzen Funktionalität wurde initial implementiert, indem das Passwort des Nutzers durch ein Zufallspasswort ersetzt wurde. Dieses Passwort wurde dem Nutzer per E-Mail mitgeteilt.

In Hinblick auf die von uns gewählte Verschlüsselungsmethode für Netzdienst-Zugangsdaten haben wir diesen Ansatz verworfen: Das Secret hierfür ist faktisch das Klartext-Passwort des Benutzers. Eine Funktionalität, die jedem offen steht und das Passwort des Benutzerkontos ändert, führt in diesem Kontext zu Problemen: Ein Nutzer kann sich zwar noch einloggen, jedoch nicht mehr seine Zugangsdaten entschlüsseln.

Aus diesem Grund haben wir uns dazu entschlossen, die andere Variante der Anforderungsbeschreibung zu implementieren.

6. Literatur

[1] *Cryptography API: Next Generation*. URL: https://www.playframework.com/documentation/2.6.x/JavaSessionFlash (besucht am 10. 11. 2018).

[2] *Time Based One Time Password*. URL: http://www.rfc-editor.org/info/rfc6238 (besucht am 04. 12. 2018).

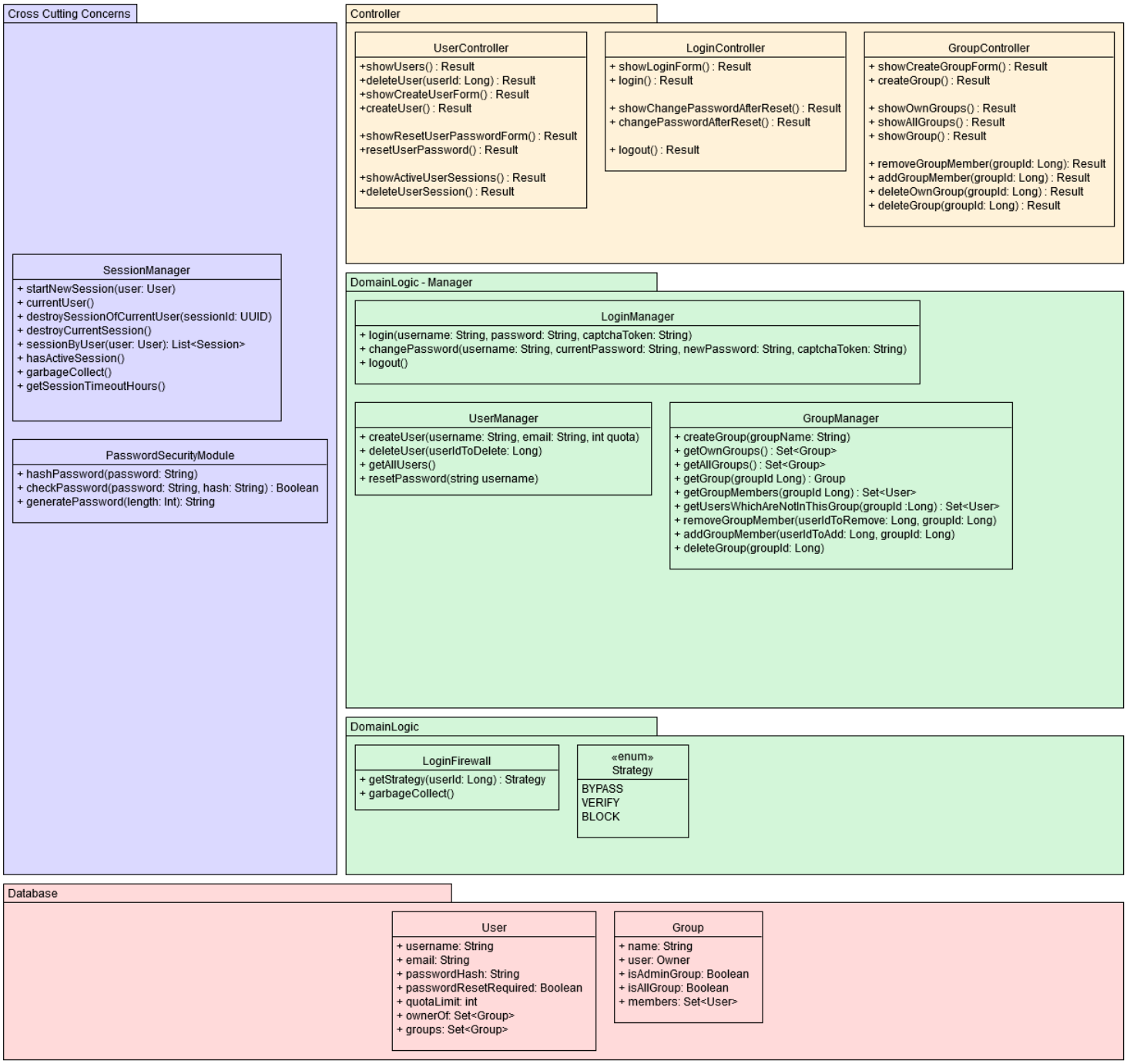
[3] Ka-Ping Yee. „User interaction design for secure systems“. In: *International Conference on Information and Communications Security*. Springer. 2002, S. 278–290.

Appendix

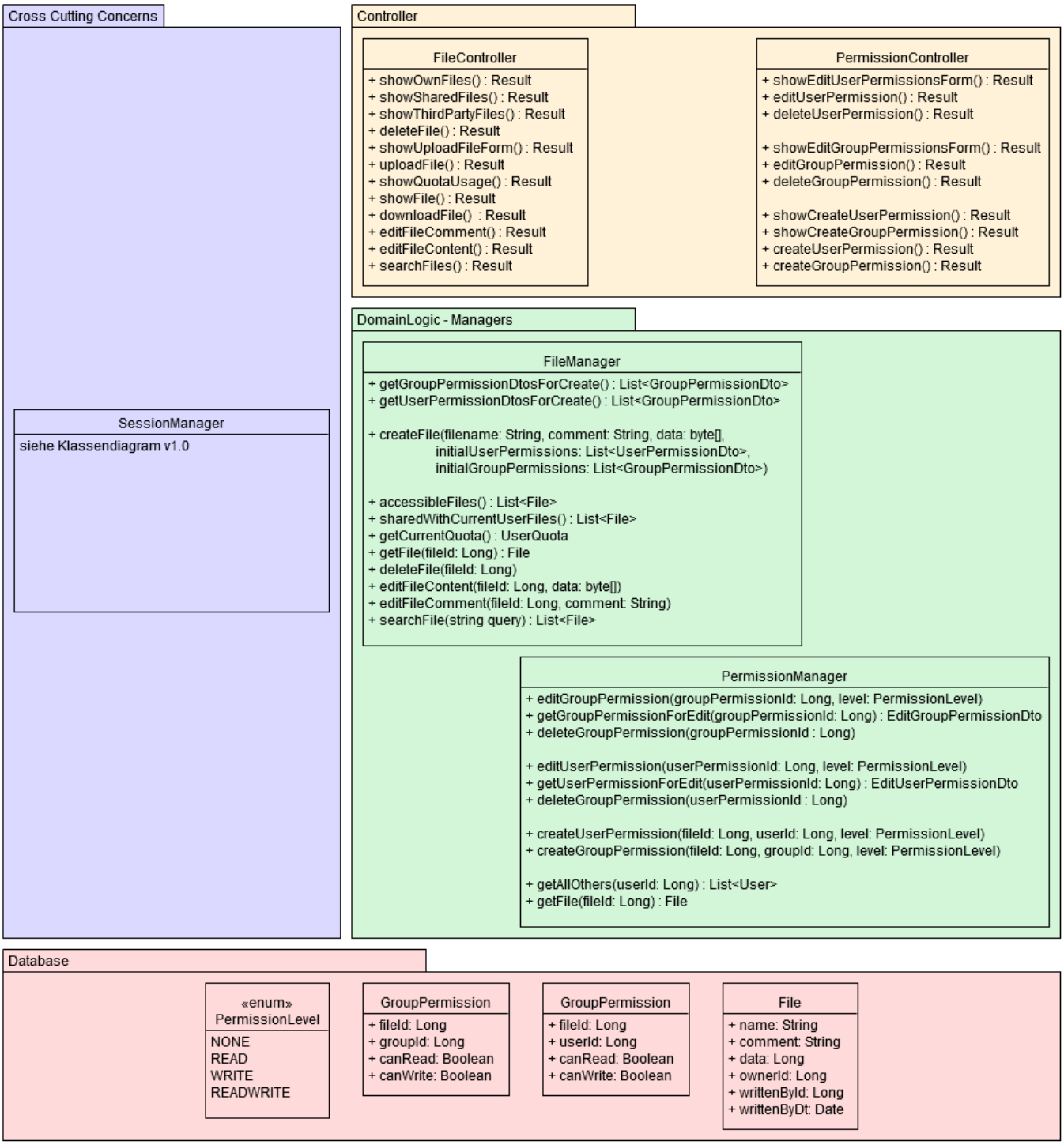
A. Aufbau eines JSON Web Token

```
Header:
{
        "alg" : "HS256",
        "typ" : "JWT"
}
Payload:
{
        "data" : {
                csrfToken": "e80706be9dfc0a7bf0e7246374be6dc3fa42001e
                    ↪ -1542826455895-98a06066234c951115dbd75e"
        },
        "nbf": 1542826455, // not before
        "iat" 1542826455, // issued at timestamp
        "exp": 1642863567 // expiration time
}
Signatur:
HMAC-SHA256(
        encodeBase64Url(header) + '.' +
        encodeBase64Url(payload),
        secret
)
```

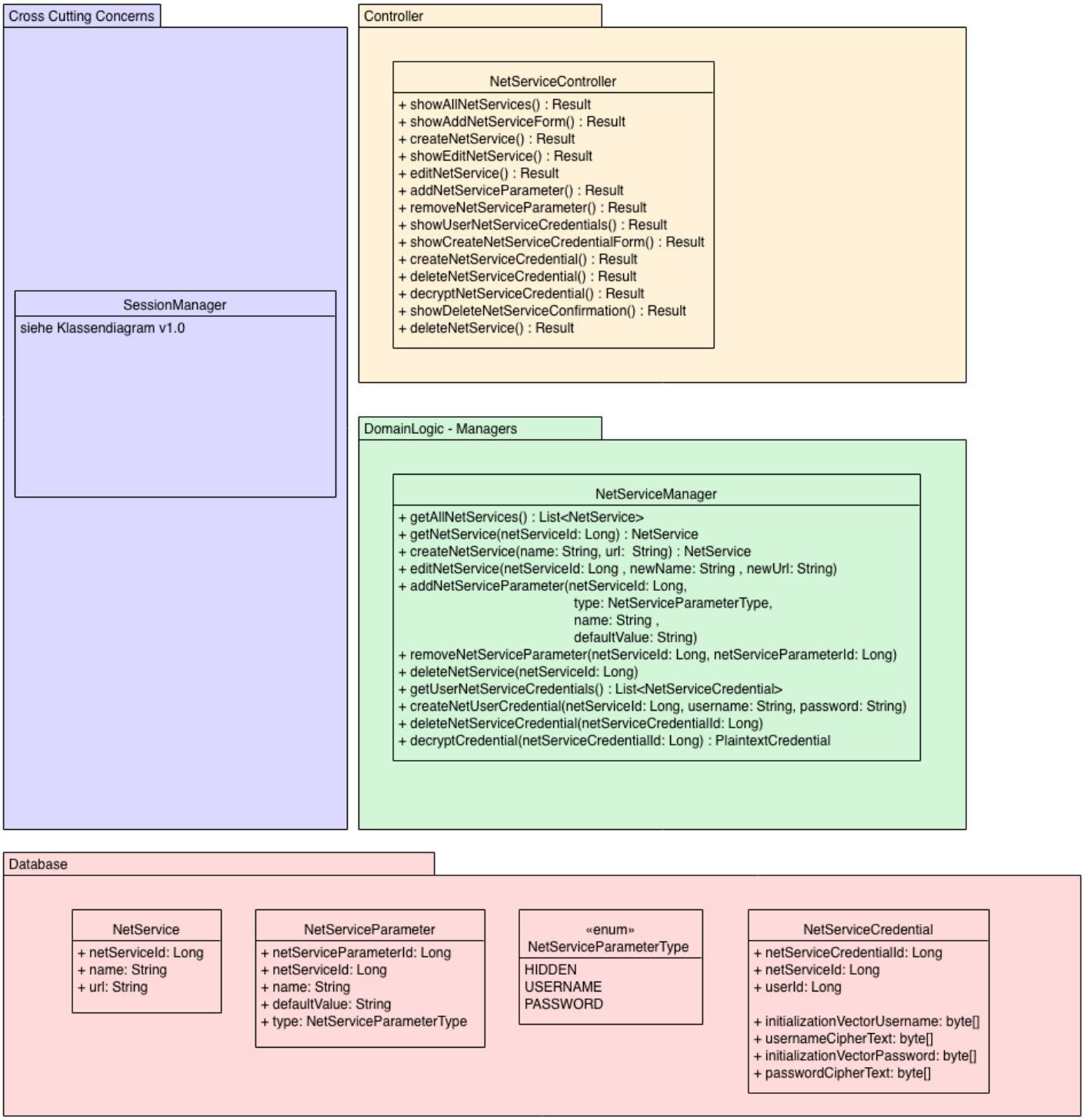
B. „High-Level“-Klassendiagramm unseres Systems (Erste Iteration)



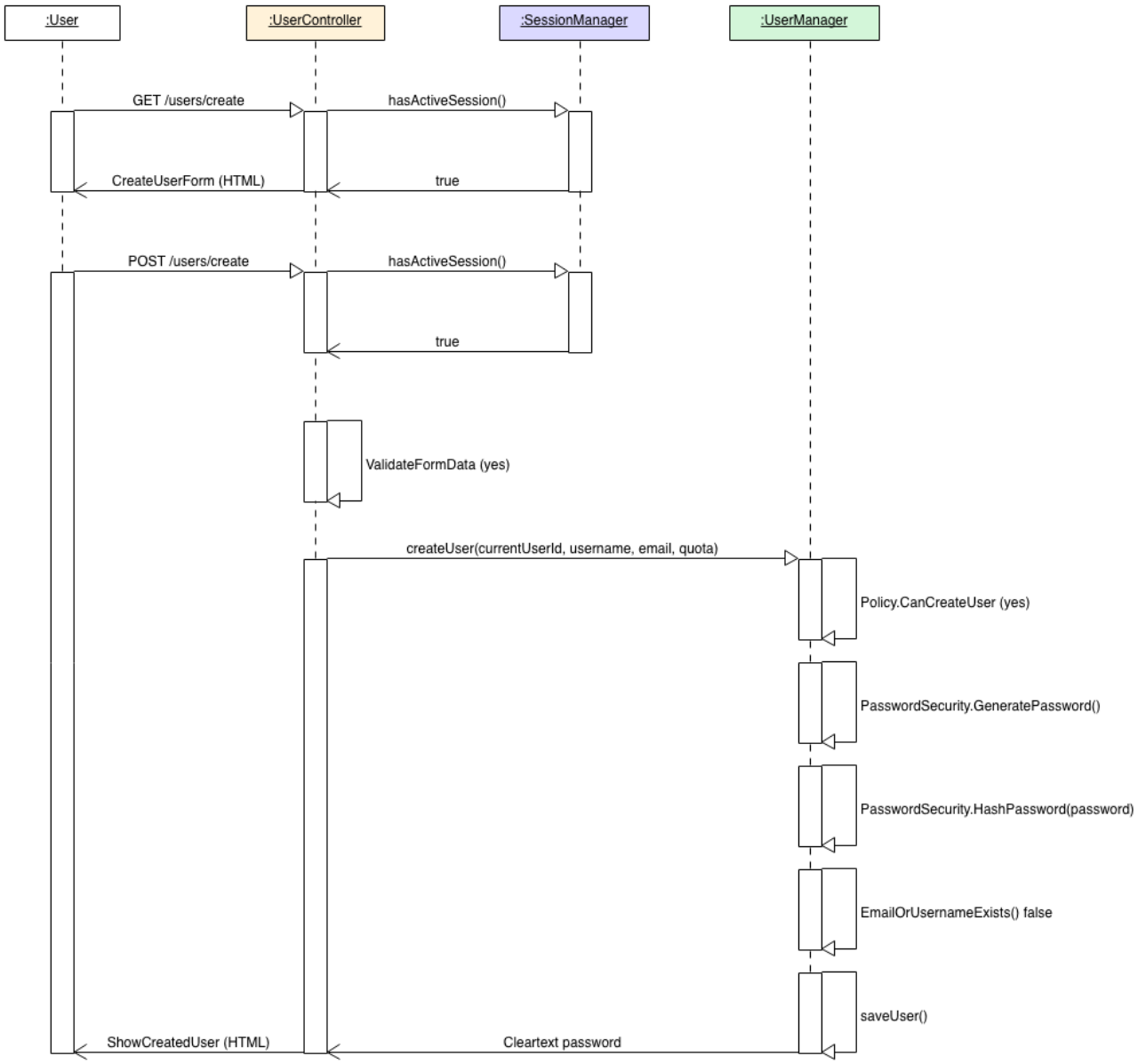
C. „High-Level“-Klassendiagramm unseres Systems (Zweite Iteration)



D. „High-Level“-Klassendiagramm unseres Systems (Dritte Iteration)



E. Sequenzdiagramm für „Benutzer erstellen“-Prozess



F. Sequenzdiagramm für „Login“-Prozess