M223

NUCLEAR

Timo Schlumpf

Siemens

1 Table of Contents

1.1 Link zum Repository

https://github.com/seiemens/nuclear

2 Aufbau und Ablauf (Teil 1)

2.1 Ausgangslage

Inhaltlich geht es in der Projektarbeit des Moduls 223 um die Implementation / Erstellung einer Multiuser – Applikation, welche aus einer zentralen Datenbank ihre Daten bezieht, sowie ein zentrales Userverwaltungssystem hat. Dieses Projekt werden wir im Rahmen einer «IPA – Simulation» durchführen, um uns bereits mit den Bedingungen und Umständen unserer Abschlussarbeit bekannt zu machen.

2.2 Aufgabenstellung

2.2.1 Anforderungen

- Front- und Backend
- Zentrale Datenbank
- Mehrere User greifen gleichzeitig auf den gleichen Datenbestand zu
- Zentrale Benutzer- und Rechteverwaltung

2.2.2 Akzeptanzkriterien

- Pünktliche Abgabe gemäss ÜK-Kriterien
- Realisierung der Applikation
- Dokumentation der Arbeiten

2.2.3 Funktionale Anforderungen

- Es wird eine Datenbank verwendet
- Als Benutzer soll man sich registrieren können
- Als Benutzer soll man sich in sein bereits erstelltes Konto einloggen können
- Als Admin soll man sich mit vorgegebenen Daten einloggen können
- Benutzer benötigen ein Konto, um die Software zu benutzen
- Alle Konten (Benutzer & Admins) sollen sich via Passwort& E-Mail einloggen können, welche sie bei der Registration festlegen resp. bereits vorgegeben sind
- Benutzer sollen Dateien hochladen können
- Benutzer sollen Dateien aus nuclear löschen können
- Benutzer sollen Dateien via Link freigeben können
- Administratoren sollen eine Ansicht haben, in welcher sie User löschen können
- Hochgeladene Dateien sind für Admins nicht einsehbar
- Jegliche Benutzereingaben werden validiert
- Das Backend wird mit Rust erstellt
- Das Frontend wird mit Javascript / Typescript erstellt

2.2.4 Nicht-Funktionale Anforderungen

- Responsive UI
- Vorschau / Placeholder von kompatiblen Dateien
- Die Applikation soll vor gängigen Cyberattacken geschützt sein

2.2.5 Individuelle Bewertungskriterien

Zum gängigen Kriterienkatalog kommen wie bereits erwähnt noch 3 individuelle Bewertungskriterien, die aus einem Katalog herausgesucht wurden. Für dieses Projekt habe ich folgende ausgewählt:

2.2.5.1 1. Individuelles Kriterium

Nummer – Bezeichnung	
166 – Codingstyle: Lesbarer Code	
Leitfrage	
Ist der Code lesbar geschrieben, gut gegliedert und ist die Namensgebung gut gewählt?	
Gütestufe 3	Gütestufe 2
Die Namensgebung entspricht den Vorgaben oder ist einfach gut gewählt. Die Struktur des Codes ist ebenfalls gemäss möglichen Richtlinien oder einfach übersichtlich gemacht. Es ist eine gewisse Einheit zu sehen in der Art und Weise, wie der Code strukturiert ist (d.h. es ist überall etwa gleich gemacht)	Die Namensgebung ist ab und zu ungeschickt gewählt, Vorgaben sind teilweise berücksichtigt. Die Codestruktur ist uneinheitlich (so das Lesbarkeit leidet).
Gütestufe 1	Gütestufe 0
Die Namensgebung ist öfters verwirrend oder unpräzise. Dem Code fehlt es an einigen Stellen an klarer Struktur. Vorgaben sind nicht berücksichtigt.	Die Namensgebung ist verwirrend oder unpräzise. Der Code ist schlecht lesbar.

2.2.5.2 2. Individuelles Kriterium

Nummer – Bezeichnung	
123 – Kommentare Im Quellcode	
Leitfrage	
Wurde der Sourcecode der Applikation ausreichend kommentiert?	
Gütestufe 3	Gütestufe 2
Der Sourcecode der Applikation ist vollumfänglich kommentiert:1. Funktionen, Parameter, Rückgabewerte,2. Wichtige Stellen im Sourcecode,3. weitere zusätzliche/nützliche Kommentare.	Der Sourcecode der Applikation ist im Grossen und Ganzen kommentiert. Einer der genannten Punkte könnte präziser sein.
Gütestufe 1	Gütestufe 0
Der Sourcecode der Applikation ist nur teilweise kommentiert.	Der Sourcecode der Applikation ist unzureichend kommentiert.

2.2.5.3 3. Individuelles Kriterium

Nummer – Bezeichnung 194 – Plausibilisierung der Benutzereingaben	
Leitfrage Werden die Eingaben des Benutzers überprüft?	
Gütestufe 3 Alle Eingabefelder werden überprüft. Es Ist eindeutig gekennzeichnet, welche Felder Pflichtfelder sind. Für den Benutzer ist ersichtlich, welche Wertebereiche zulässig sind. Findet die Plausibilisierung eine Fehleingabe, so wird der Benutzer mit konkreten Hinweisen geführt, das entsprechende Feld wird aktiviert.	Gütestufe 2 Plausibilisierung findet statt, Feedback an Benutzer ist mangelhaft/nicht eindeutig/unvollständig. Nur korrekte Daten werden übermittelt.
Gütestufe 1 Eingaben werden plausibilisiert, aber bei Fehlern oder fehlenden Eingaben sind die bisher gemachten Eingaben verloren oder die fehlerhaften Eingaben werden trotzdem übermittelt. Oder: es werden nicht alle Eingaben überprüft, welche überprüft werden sollten.	Gütestufe 0 Es findet keine Plausibilisierung statt.

2.2.6 Technologien

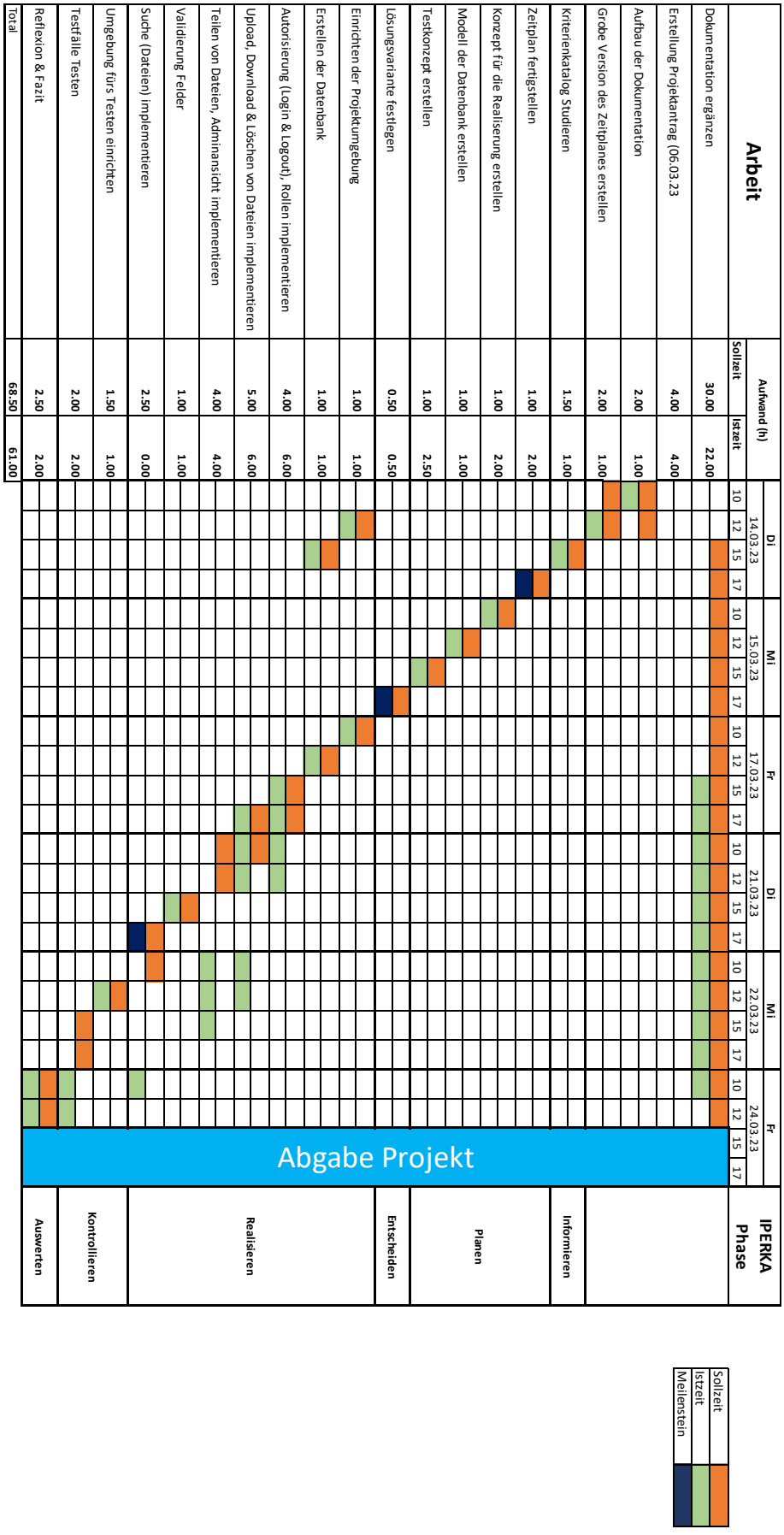
- Nuxt v3
  - Front-End Framework
  - Basiert auf Typescript / Javascript
- Rocket.rs
  - Serverseitiges Webframework
  - Basiert auf Rust, eine Low Level Programmiersprache
- MongoDB
  - Daten werden in JSON-ähnlichen Dokumenten verwaltet
  - NoSQL DB
  - Flexibel und skalierbar

2.3 Zeitplan

2.3.1 Vorgegebene Termine

Datum	Termin
27.02.2023	Infotag M223
14.03.2023	Projektstart
14.03.2023	1. Gespräch mit Experten
22.03.2023	2. Gespräch mit Experten
24.03.2023	Projektabschluss

2.3.2 Zeitplan



2.4 Arbeitsjournal

14.03.23	
Geplante Arbeiten (Zeitplan)	• Dokumentation erstellen • Zeitplan finalisieren • Kriterienkatalog studieren • Expertengespräch
Ausgeführte Arbeiten	Die Dokumentation wurde erstellt und mit einem Titelblatt & Inhaltsverzeichnis versehen. Der Zeitplan wurde fertiggestellt und hochgeladen. Die Projektstruktur wurde im Github erstellt, und in die Unterordner Frontend & Backend unterteilt. Der Kriterienkatalog wurde intensiv studiert.

Hilfestellungen	• Gespräch mit Lara
	  ○ Anpassungen:
	  ▪ Meilensteine

Erfolge & Misserfolge	+ Zeitplan fertig (ausser soll Zeiten) + Gut gestartet und bis jetzt im Zeitplan
Bemerkung	Meilenstein “Zeitplan fertigstellen” erledigt
Nächste Schritte	• Realisierungskonzept erstellen • Modell der Datenbank erstellen • Testkonzept erstellen • Lösungsvariante festlegen
Fazit	Der Anfang ist mir so weit gelungen, ich befinde mich ein bisschen vor meinem Zeitplan, was sehr gut ist. Technisch gab es ein paar Schwierigkeiten bzgl. Rust & Rocket.rs, da ich noch den Branch auf «0.5-.2rc» wechseln musste, da dieser gewisse Features enthält, die ich brauchen werde.

15.03.23	
Geplante Arbeiten (Zeitplan)	• Realisierungskonzept erstellen • Modell der Datenbank erstellen • Testkonzept erstellen • Lösungsvariante festlegen
Ausgeführte Arbeiten	Ich war mir seit Anfang an ziemlich sicher mit welchen Technologien ich das Projekt machen werde. Das Backend werde ich mit Rocket.rs machen, eine Webserver Library die in rust implementiert ist. Rocket.rs macht das Implementieren von Systemen sehr einfach, da es «out of the Box» bereits sehr viele Sicherheitsvorkehrungen und Hilfestellungen beinhaltet. Für das Frontend werde ich Nuxt3 verwenden, da ich mit diesem Framework bereits viel gearbeitet habe. Danach habe ich noch die Dokumentation erweitert, und das Arbeitsjournal erstellt.
Hilfestellungen	-
Erfolge & Misserfolge	+ Dokumentation wurde aktualisiert + Datenbank darf mit MongoDB gemacht werden, was den Aufwand reduziert • Testkonzept konnte noch nicht erstellt werden, da mir die Zeit fehlte
Bemerkung	-
Nächste Schritte	• Einrichten der Projektumgebung • Erstellen & Einrichten der DB • Impl. Auth – Prozess (Login & co.)
Fazit	Alles in allem bin ich noch gut in der Zeit. Ich konnte meine Projektumgebung einrichten und meine Datenbank erstellen. Jedoch muss ich mein Testkonzept noch nachholen, da dies noch fehlt.

17.03.23	
Geplante Arbeiten (Zeitplan)	• Einrichten der Projektumgebung • Erstellen & Einrichten der DB • Impl. Auth – Prozess (Login & co.) • Upload, Download & Delete von Dateien implementieren
Ausgeführte Arbeiten	Die Projektumgebung wurde eingerichtet und konfiguriert. Die Verbindung «DB – Backend – Frontend» wurde implementiert und optimiert, um einen effizienten Umgang mit den Models zu garantieren. Der Authentifizierungsprozess wurde im Backend sowie Frontend implementiert.
Hilfestellungen	-
Erfolge & Misserfolge	+ DB-Verbindung wurde erfolgreich implementiert + Landing Page erstellt
Bemerkung	•
Nächste Schritte	• Upload, Download & Delete von Dateien implementieren • Validierung von Usereingaben • Dateisuche (Frontend) implementieren
Fazit	Die ersten Entwicklungsarbeiten sind mir geglückt und konnten erfolgreich umgesetzt werden. Der Zeitplan und die Dokumentation wurden aktualisiert und die Homepage meiner Applikation steht bereits. Ich hatte ein paar kleine Probleme mit CORS (Cross Origin Resource Sharing), da Rocket.rs eine sehr strikte Policy hat. Auch habe ich ein bisschen mit den Auth Cookies gekämpft, da diese aufgrund von Cors nicht richtig gesetzt wurden. Beide Probleme sind aber nun gelöst und es funktioniert einwandfrei. Zum Dateien – CRUD bin ich noch nicht gekommen, aber werde nächsten Dienstag mit dem beginnen.

21.03.23	
Geplante Arbeiten (Zeitplan)	• Upload, Download & Delete von Dateien implementieren • Validierung von Usereingaben • Dateisuche (Frontend) implementieren
Ausgeführte Arbeiten	Up - & Download von Dateien wurde implementiert. Die Usereingaben werden an den nötigen Stellen validiert.
Hilfestellungen	-
Erfolge & Misserfolge	+ Erfolgreiche Implementation Up & Download + Eingabenvalidierung + Aktualisierung Dokumentation
Bemerkung	•
Nächste Schritte	• Share Funktion der Dateien • Adminpanel • Dateiensuche Implementieren • Testumgebung einrichten • Testfälle
Fazit	Heute war ich leider nicht so produktiv wie erhofft. Der Grund dafür ist meine Übermüdung, da ich in letzter Zeit aufgrund des Stresses nicht allzu viel & gut schlafe. Trotzdem konnte ich den Up & Download implementieren und mit dem Löschen von Dateien starten. Der Zeitplan & die Doku wurden nachgeführt und die Uservalidierung steht zu 90%. Das Dateisuche Feature habe ich auf Eis gelegt, da mir dazu im Moment die Zeit fehlt.

22.03.23	
Geplante Arbeiten (Zeitplan)	• Share Funktion der Dateien • Adminpanel • Dateiensuche Implementieren • Testumgebung einrichten • Testfälle • Expertengespräch
Ausgeführte Arbeiten	Die Sharefunktion wurde zu 90% implementiert. Die Testumgebung wurde eingerichtet und die Testfälle wurden erstellt.

Hilfestellungen	• Gespräch Lara
	  ○ Anpassungen:
	  ▪ Fokus auf Doku legen
	  ▪ Unfertiges Produkt weniger schlimm als unfertige Doku

Erfolge & Misserfolge	+ Finalisierung der Löschfunktion + Sharing zu 90% fertig + Dokumentation erweitert + Tagesjournale nachgeführt + Realisierungsteil Doku angefangen
Bemerkung	•
Nächste Schritte	• Sharefunktion der Dateien fertigstellen • Adminpanel • Dokumentation
Fazit	Heute habe ich hauptsächlich an der Delete & Sharefunktion gearbeitet, welche nun fertig resp. zum Grossteil fertig sind. Die Dokumentation wurde weitergeführt und am Gespräch erhielt ich einige gute Tipps, die ich umsetzen werde. Da ich Dokumentationstechnisch ein bisschen im Verzug bin, werde ich mich heute & am Freitag darauf fokussieren, da laut Lara ein nicht fertiggestelltes Produkt weniger schlimm ist.

24.03.23	
Geplante Arbeiten (Zeitplan)	• Sharefunktion der Dateien fertigstellen • Adminpanel • Dokumentation
Ausgeführte Arbeiten	Das Projekt wurde im aktuellen Stand zur Seite gelegt und der Fokus wurde auf die Doku gelegt.
Hilfestellungen	-
Erfolge & Misserfolge	+ Dokumentation erweitert
Bemerkung	•
Nächste Schritte	-
Fazit	Heute habe ich nur an der Doku gearbeitet.

3 Projekt (Teil 2)

3.1 Kurzzusammenfassung

3.1.1 Ausgangslage

Das Ziel ist es, ein «Google Drive Clone» zu erstellen. Bei diesem soll man in der Lage sein, Dateien hochzuladen, wieder zu löschen, herunterzuladen oder diese mit Freunden via Link zu teilen. Der User soll beliebig viele Dateien hochladen können, ohne dass dies die Performance des Systems beeinträchtigt. Der User wird durch ein Cookie – Authentication System authentifiziert, um sicherzugehen das er nur eigene Dateien sieht – und gleichzeitig Aussenstehenden den Zugriff komplett verweigert, bis sie sich registriert haben.

3.1.2 Umsetzung

Das Projekt wurde unter Verwendung der IPERKA-Projektmethodik durchgeführt.

Als Erstes stand die Informationsphase an, in dieser wurde gemäss der Aufgabestellung Anforderungen definiert.

Während der Planungsphase wurde ein Zeitplan, eine Grundstruktur für die Dokumentation erstellt.

In der Realisierungsphase wurde mithilfe des Nuxt-Webframework basierend auf Javascript / Typescript, HTML und CSS das Frontend und Mithilfe von Rocket.rs, einem Rust-basierendem Framework das Backend erstellt.

Die Kontrollphase bestand aus manuellen Tests.

Zum Schluss stand eine Auswertung an, in welcher Erfolge, Probleme und Verbesserungsmöglichkeiten analysiert wurden. Aufgrund der vorliegenden Fakten wurde ein Fazit gezogen.

3.1.3 Ergebnis

Mit der entstandenen Applikation lassen sich Dateien einfach online speichern, löschen und teilen. Die Daten sind verschlüsselt und sämtliche sensitive Daten (wie Passwörter) sind gehashed. Durch das zentrale Rollensystem sind Erweiterungen (Admin-Panel) einfach zu implementieren. Das Datenbankmodell ist selbsterklärend und beinhaltet nur die absolut essenziellsten Daten, um das System brauchbar zu machen. Um weiter Daten so sicher wie möglich zu halten, werden vom User nur die Daten benötigt, die auch absolut notwendig sind, wie zum Beispiel Name, E-Mail und ein Passwort.

3.2 Informieren

Das genaue Analysieren der Aufgabenstellung ist das Wichtigste in der ersten Phase der Projektplanungsmethode IPERKA.

3.2.1 Anforderungen

Hier sind alle funktionalen, nicht funktionalen und erweiterten Anforderungen basierend auf dem Projektantrag genau definiert. Basierend auf diesen Anforderungen wird im nächsten Teil die Implementation geplant.

3.2.1.1 Funktionale Anforderungen

- Benutzer: innen mit speziellen Berechtigungen (Administratoren) können bestehende User löschen, jedoch *nicht* erstellen.
- Benutzer benötigen ein Konto, um die Software verwenden zu können
- Nicht registrierte Benutzer: innen können sich registrieren, welches folgende Daten benötigt:
    - Name & Vorname
    - E-Mail
    - Passwort
- Bereits registrierte Benutzer: innen können sich mit den oben genannten Daten (E-Mail & Passwort) einloggen
- Als Administrator soll man sich mit vorgegebenen Daten einloggen können
- Angemeldete Benutzer: innen sollen in der Lage sein, via Upload-Menu Dateien von ihrem lokalen Gerät auf die Software hochzuladen
- Angemeldete Benutzer: innen können nur ihre eigenen Dateien sehen und manipulieren
- Angemeldete Benutzer: innen sollen hochgeladene Dateien wieder aus der Software entfernen können
- Angemeldete Benutzer: innen können ihre eigenen Dateien via Link herunterladen
- Angemeldete Benutzer: innen können ihre eigenen Dateien via Link teilen, um den Download für dritte zu ermöglichen

3.2.1.2 Erweiterte Anforderungen

Falls noch genügend Zeit besteht, werden folgende Funktionen implementiert:

- Ein Suchfenster in der Dateiübersicht, um die Bedienung zu vereinfachen
- Vorschau von kompatiblen Dateien, wie zum Beispiel Bilder

3.2.1.3 Nicht-Funktionale Anforderungen

- Responsives UI; Soll auf PC's sowie Tablets brauchbar aussehen
- Die Applikation soll vor gängigen Cyberattacken (MITM, SQL-Injection, etc.) geschützt sein
- Die Applikation ist Fehlertolerant; Sie stürzt bei Falscheingaben nicht direkt ab
- Passwörter und sensible Daten werden gehashed
- Keine Performanceeinbussen bei grösseren Datenmengen
- Einfache Wartbarkeit

3.3 Planen

Diese Phase ist ein essenzieller Teil des Projektes, da hier der gesamte Projektverlauf bestimmt wird. Der Zeitplan befindet sich im Kapitel «Zeitplan» und wird aus Redundanzgründen nicht noch einmal aufgeführt. Das Projekt wurde von Anfang bis Schluss mit der Projektplanungsmethode IPERKA geplant & durchgeführt. Unit Tests wurden nicht verlangt, jedoch sind manuelle Tests durchzuführen. Diese sind unter «Testkonzept» einsichtbar.

3.3.1 Sitemap

Eine Sitemap soll während dem Realisieren einen guten Überblick über sämtliche Unterseiten verschaffen und deren Verbindungen verdeutlichen.

- **Login / Register**
  - Der User kann sich hier einloggen / neu registrieren
- **Home**
  - Eine Landingpage, welche als «Eyecandy» dient
- **Dashboard** (User)
  - Hier sieht der User seine hochgeladenen Dateien und kann diese verwalten
- **Dashboard** (Admin) [NICHT IMPLEMENTIERT]
  - Ein Administrator kann hier die Benutzer verwalten
- **Profile** (User) [NICHT IMPLEMENTIERT]
  - Der User kann hier sein Profil einsehen und verwalten, wie z.B. Änderung der E-Mail

Sämtliche Punkte die mit [NICHT IMPLEMENTIERT] markiert wurden, konnten aus Zeitgründen nicht vollständig implementiert werden. Jedoch existiert die Struktur bereits, um einfache Wartbarkeit zu garantieren.

3.3.2 Testkonzept

Testfall #1	Login-Form
Beschreibung	User loggt sich in einen bestehenden Account ein.
Durchführung	• User klickt auf den Button «Login» • User gibt Angaben an • Klickt auf «Anmelden» • Wird auf Dashboard weitergeleitet
Erwartetes Resultat	Der Benutzer wird auf sein persönliches Dashboard weitergeleitet.

Testfall #2	Registrierformular
Beschreibung	Ein neuer User registriert einen neuen Account
Durchführung	• User klickt auf den Button «Register» • User gibt Angaben an • Klickt auf «Registrieren» • Eine Erfolgsmeldung erscheint, die ihn auf die Login Seite weiterleitet
Erwartetes Resultat	Ein neuer User wird in der Datenbank erstellt.

Testfall #3	File - Upload
Beschreibung	Ein User lädt eine neue Datei hoch
Durchführung	• User navigiert zum Dashboard • Klickt auf «Upload» • Wählt seine Datei aus • Schliesst den Prompt • Klickt auf «Send it»
Erwartetes Resultat	Die Datei wird auf dem Server generiert und in die Datenbank eingefügt. Die Datei ist nun auf dem Dashboard ersichtlich.

Testfall #4	File - Deletion
Beschreibung	Ein User löscht eine hochgeladene Datei
Durchführung	• User navigiert zum Dashboard • Wählt die zu löschende Datei aus • Klickt in dem Dateicontainer auf das «Trash» Icon
Erwartetes Resultat	Die Datei wird auf dem Server gelöscht. Der Datenbankeintrag wird entfernt und die Datei ist auf dem Dashboard nicht mehr sichtbar.

Testfall #5	File - Sharing
Beschreibung	Ein User teilt eine Datei via generierten Link
Durchführung	• User navigiert zum Dashboard • Wählt die zu teilende Datei aus • Klickt auf das «Share» Icon
Erwartetes Resultat	Die Datei wird auf dem Server präpariert, ein Link wird generiert und dem User in das Clipboard kopiert.

Testfall #6	File - Download
Beschreibung	Ein User lädt eine hochgeladene Datei herunter
Durchführung	• User navigiert zum Dashboard • User wählt die die Datei aus • User klickt auf das «Share» Icon • User fügt den Link in ein neues Browserfenster ein
Erwartetes Resultat	Die Datei wird auf den Computer des Users heruntergeladen.

Testfall #6	Profile Page
Beschreibung	User kann seine Daten im Profilepanel managen
Durchführung	• User loggt sich ein • Klickt auf «Profile»
Erwartetes Resultat	Der Benutzer wird auf sein Profil weitergeleitet. Hier kann er sämtliche Daten bzgl. Seinem Account einsehen & manipulieren.

Testfall #7	Login - Admin
Beschreibung	Admin loggt sich ein
Durchführung	• User klickt auf den Button «Login» • User gibt Angaben an • Klickt auf «Anmelden»
Erwartetes Resultat	Der Administrator wird auf das Admindashboard weitergeleitet

Testfall #8	Admin – Löschen eines Users
Beschreibung	Admin löscht einen User
Durchführung	• Admin klickt auf «Dashboard» • Admin wählt zu löschenden User aus • Admin klickt auf «löschen»
Erwartetes Resultat	Der User wird aus der Datenbank entfernt. Loginversuche mit diesen Angaben werden abgelehnt.

3.4 Entscheiden

In der Entscheidungsphase der Projektplanungsmethode IPERKA wird der Lösungsweg definiert. Dabei werden Lösungsvarianten verglichen und Risiken evaluiert.

3.4.1 Varianten

Für die Entwicklung des Frontend gibt es verschiedene Lösungswege. Zuerst musste ich mich für ein Framework entscheiden. Hier gab es folgende Optionen:

- Nuxt3 (Vue.js)
- Nuxt2 (Vuejs)
- Angular
- React

Angular und React fielen direkt weg, da mir mit diesen Frameworks die Erfahrung fehlt, ein solches Projekt umzusetzen. Der wesentliche Unterschied zwischen Nuxt2 und Nuxt3 besteht darin, das Nuxt2 «axios» verwendet, um http-Requests zu senden, für welche Nuxt3 eine eigene Library verwendet.

Hier ein Beispiel in axios:

```
axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
})
.then(function (response) {
    console.log(response);
})
.catch(function (error) {
    console.log(error);
});
```

Und hier ein Beispiel in Nuxt3 mit «$fetch()»:

```
const result =
await $fetch('http://localhost:4200/delete', {
       mode:"cors",
      method: "POST",
      credentials:"include",
      body: { id: this.id }
});
```

Auch wenn die Unterschiede nur minimal sind, war dies trotzdem eine grosse Entscheidung. Mit Nuxt2 bin ich bereits sehr vertraut und habe viel mit axios gearbeitet. Nuxt3 habe ich noch nie verwendet, jedoch hat es einige neue Funktionalitäten, welche das Entwickeln einiges einfacher machen sollen.

3.4.2 Risiken

Bei der Wahl der Datenbank gab es einige Kleinigkeiten, die durchdacht werden mussten. Die wichtigste Frage war SQL oder NoSQL, also PostgreSQL oder MongoDB; beide haben ihre Vorteile sowie Nachteile. Mit MongoDB bin ich einiges vertrauter und habe in der Vergangenheit mehr Projekte damit geschrieben als mit SQL, jedoch würde sich SQL auch anbieten, da es sich für mein spezifisches Use-Case eignet. Der Vorteil an MongoDB ist die «Document-Based Structure», oder Dokumentenstruktur, die Daten im JSON-Format speichert.

3.4.3 Entscheid

Aufgrund der Vorkenntnisse entschied ich mich für Nuxt3 für das Frontend, da die neuen Funktionalitäten den Entwicklungsprozess einiges einfacher machen. MongoDB ist meine Wahl für die Datenbank, da ich bereits viel Erfahrung darin habe. Für das Backend fiel der Entscheid auf Rocket.rs, ein Rust-Basiertes Webserver Framework. Diese Entscheidung stand schon vor dem Projektbeginn, da Rust mir sehr viel Spass macht.

3.5 Realisieren

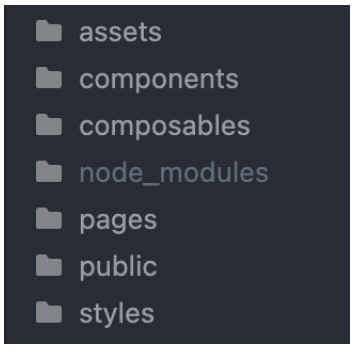
In der vierten Phase der Projektplanungsmethode IPERKA geht es um das Realisieren der Applikation. Hier werden komplexe Stellen erläutert und dargestellt. Im Code befinden sich auch Kommentare, sofern diese der Verständlichkeit helfen.

3.5.1 Frontend

Wie bereits erwähnt wurde das Frontend mit Nuxt3 erstellt.

Nuxt3 ist ein Javascript & Typescript-basiertes Frontend-Framework, das durch eine flexible Ordnerstruktur und «Page – Component» System überzeugt. Da das Framework Open Source ist, werden die meisten Sicherheitsprobleme direkt identifiziert und behoben, was es ein sehr sicheres Framework macht.

3.5.1.1 Ordnerstruktur



Um gute Wartbarkeit zu gewährleisten, habe ich mich für eine detaillierte & übersichtliche Ordnerstruktur entschieden. Assets, Components, Composables, Pages & dessen Styles befinden sich auf dem Rootlevel der Applikation.

Innerhalb des Component-Ordners befinden sich Komponenten, die in Seiten eingebettet werden. Components sowie Pages können via CSS gestaltet werden, wessen CSS-Files dann im «Styles»-Ordner abgelegt werden.

«Composables» ist ein neues Feature von Nuxt3; Es ermöglicht die Definition von globalen Funktionen und deren Verwendung via direkter Reference.

3.5.2 Design

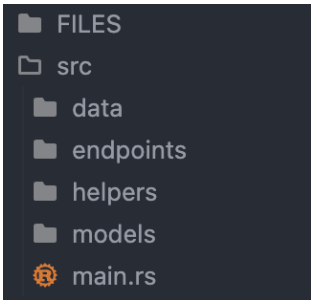
Das Design wurde nicht geplant, sondern einfach drauflos programmiert. Da der Aufbau der Seite (siehe Sitemap) von Anfang an ziemlich klar war, konnte ich mich komplett austoben. Für die Wahl der Farbpalette verwendete ich fffuel.co, ein Tool, das den Designprozess drastisch beschleunigt.

3.5.3 Backend

Rust für ein Backend zu verwenden ist keine optimale Wahl. Die Sprache ist noch relativ neu, und Frameworks sind deswegen noch nicht voll ausgereift. Rocket.rs ist da keine Ausnahme. Das Framework wurde in der Version «0.5-rc.2» verwendet. Dies ist der Release Candidate für die Version 0.5, also eine noch nicht definitive Version.

3.5.3.1 Ordnerstruktur

Die Ordnerstruktur ist gleich wie im Frontend sehr modular und übersichtlich aufgebaut.



3.5.3.1.1 Data

Der “data” Ordner enthält die Datenbankverbindung zum «Business-Layer». Diese wird benötigt, um sämtliche Daten einzufügen, modifizieren oder zu löschen. Dieser Ordner enthält nicht wirklich viel Code, abgesehen von den üblichen CRUD – Methoden.

3.5.3.1.2 Endpoints

Hier sind sämtliche Endpoints zu finden. Endpoints in rocket.rs sind sehr einfach zu definieren. Im Makro gibt man die http-Methode an, die man verwenden möchte. Darunter kommt die Funktionsdefinition, die bei einer Request aufgerufen wird.

```
/// create a new user
#[post("/user/new", data="<u>")]
pub async fn new_user(
    u:Json<User>,
    db: &State<Connector>
) -> Result<Json<InsertOneResult>, Status> {
    // parse the request data into a processable model
    let data = parse_user(u).unwrap();
    let user_detail = db.create_user(data).await;

    // rust "switch" case:
    //check if the user is valid or not
    match user_detail {
        Ok(user) => Ok(Json(user)),
        Err(_) => Err(Status::ImATeapot),
    }
}
```

Returnwerte können komplett unterschiedlich sein; Man legt sie im Returnwert der Funktion fest. In diesem Beispiel wird ein gültiges Usermodel zurückgeschickt, sofern der Vorgang erfolgreich war. Ist dies nicht der Fall, wird ein Status mit dem Code «418» zurückgeschickt.

3.5.3.2 Zentrale Funktionen

In die Helpers wurde der Grossteil des Aufwandes investiert. Hier befinden sich sämtliche Funktionen, die in irgendeiner Form benötigt werden. Um redundanten Code zu verwenden, sind sie hier zentralisiert und von jedem anderen Ordner aufrufbar. Funktionen wie CORS Fairing, Cookie Handling, Encryption / Decryption von Passwörtern, Parsing von Requestdaten und Tokenhandling sind essenziell und spielen eine zentrale Rolle für das ganze Projekt.

3.5.3.2.1 Parser

Die Parser sind das wichtigste Stück Code im ganzen Projekt. Sie sind dafür verantwortlich, die Daten aus den Requests in Models zu umformatieren und für das Backend brauchbar zu machen. Ausserdem wird das gesamte Filehandling über den Fileparser gemacht, welcher die Files von temporären Dateien zu fixen, benamsten Dateien macht und diese in die Datenbank einfügt.

3.5.3.2.2 Hashing

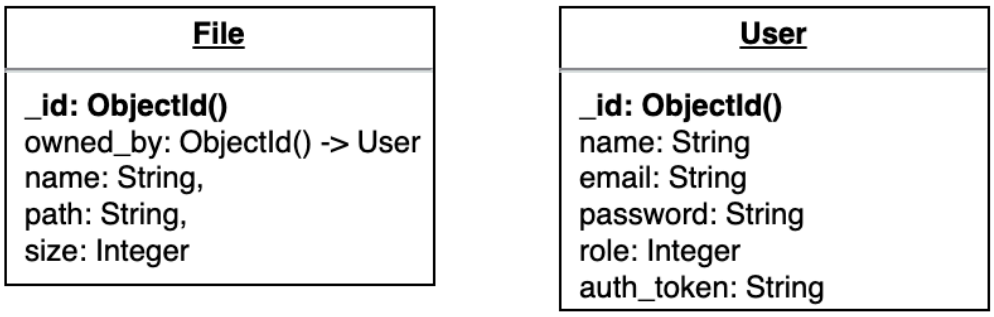
Wie bereits erwähnt wurden alle sensitiven Daten gehasht. Hashing ist ein Algorithmus, welcher einen Input (z.B. ein Passwort), einen Salt und optional einen Pepper zusammenfügt und diesen Wert als einen Chararray mit fixer Länge zurückgibt. Der Argon2 Algorithmus ist der Standard für Passwort-Encryption, welcher im Vergleich zu MD5 & SHA256 noch nicht «gecrackt» wurde, weshalb ich ihn verwendet habe.

```rust
pub fn encrypt(pw: String) -> String {
    dotenv().ok();
    let salt = env::var("SALT").expect("SALT
REQUIRED");
    // argon2 config could be altered for more security
    let config = Config::default();

    let hash = argon2::hash_encoded(
      pw.as_bytes(),
      salt.as_bytes(),
      &config)
    .unwrap();

    return hash;
}
```

3.5.4 Diagramme

Da MongoDB keine relationale Datenbank ist, wurde kein traditionelles ERD angefertigt. Anstelle wurde ein Modeldiagramm eingefügt, das die verschiedenen Datentypen darstellt.

3.5.4.1 Modeldiagramm



3.6 Kontrollieren

In der Phase Kontrollieren der Projektmethode IPERKA geht es darum, das Projekt auf die vorausgesetzte und erwartete Funktionalität zu testen. Hier werden Korrekturen, sowie gefundene Fehler beschrieben und die komplette Applikation aufgrund des Testprotokolls getestet.

3.6.1 Testdurchführung

Testfall #1	Login-Form
Beschreibung	User loggt sich in einen nicht bestehenden Account ein.
Erhaltenes Resultat	• Der User wird nicht eingeloggt oder auf das Dashboard weitergeleitet
Status	Bestanden

Testfall #2	Register Form
Beschreibung	User erstellt einen bereits vorhandenen User
Erhaltenes Resultat	• Der Neue user wird nicht erstellt und es erscheint keine «success» Meldung
Status	Bestanden

Testfall #3	File Upload
Beschreibung	User lädt eine Datei hoch
Erhaltenes Resultat	• Die Datei wird hochgeladen und auf dem Dashboard angezeigt
Status	Bestanden

Testfall #4	File Deletion
Beschreibung	User löscht eine hochgeladene Datei
Erhaltenes Resultat	• Die Datei wird von der DB und dem Dashboard entfernt
Status	Bestanden

Testfall #5	File Sharing
Beschreibung	Eine hochgeladene Datei wird vom User via Link geteilt
Erhaltenes Resultat	• Der Link wird dem User ins Clipboard kopiert, jedoch wird keine Datei heruntergeladen
Status	Fehlgeschlagen

Testfall #6	File Download
Beschreibung	Eine hochgeladene Datei wird vom User heruntergeladen
Erhaltenes Resultat	• Der Link wird kopiert und die Datei im Backend an die Response geknüpft, jedoch nicht heruntergeladen
Status	Fehlgeschlagen

Testfall #7	Admin Login
Beschreibung	Ein Administrator loggt sich mit den Angaben ein
Erhaltenes Resultat	• Der Admin wird angemeldet, jedoch auf kein Dashboard weitergeleitet.
Status	Fehlgeschlagen

Testfall #8	Admin – Löschen eines Users
Beschreibung	Ein Administrator löscht einen bestehenden User aus der DB
Erhaltenes Resultat	• Der User wird nicht gelöscht, da kein Adminpanel implementiert ist
Status	Fehlgeschlagen

3.7 Auswerten

Das Auswerten ist der letzte Schritt der IPERKA Projektplanungsmethode. In diesem Teil wird reflektiert, Erfolge und Misserfolge erkannt, sowie Verbesserungsmöglichkeiten gesucht und zum Schluss ein Fazit daraus gezogen.

3.7.1 Erfolge

Ein grosser Erfolg ist, dass ich alle meine persönlichen Ziele erreicht habe. Da ich nicht zum ersten Mal ein solches Projekt mache, legte ich mehr Wert auf kleine Details (wie zum Beispiel den Pfeil auf der Landing Page) und probierte die Seite insgesamt interessanter und intuitiver zu gestalten. Ich würde behaupten, dies ist mir auch ziemlich gut gelungen. Auch bin ich mit meiner Arbeitsverteilung zufrieden, da ich alles in einem so kleinen Zeitrahmen auf ca. 90% Fertigkeit gebracht habe.

3.7.2 Misserfolge

Misserfolge gab es nicht viele, da ich sehr speditiv und zielstrebig gearbeitet habe. Jedoch hat das Backend sowie Frontend einige Funktionalitäten, die ich aufgrund fehlender Zeit nicht fertigstellen konnte, wie zum Beispiel ein Adminpanel oder eine Dateisuche. Auch funktioniert die Sharefunktion noch nicht zu 100%, was ein bisschen Schade ist.

Ein weiterer Punkt ist die nicht komplette Einhaltung des Zeitplans. Geplant war, dass der Freitag für Reviews & Verbesserungen zu verwenden. Da ich leider durch «Rumgeplempere» massiv Zeit verlor, konnte dies nicht ganz umgesetzt werden. Auch bin ich mit meinen Arbeitsjournalen nicht zufrieden, da diese meistens ein bis zwei Tage im Verzug entstanden.

3.7.3 Verbesserungsmöglichkeiten

Aufgrund meiner Misserfolge habe ich eine Liste zusammengestellt, die diese zusammenfasst:

- **Planungsphase**
  - Zeitplan realistischer gestalten
  - Journale täglich ausfüllen
  - Ziele realistischer setzen

3.7.4 Fazit

Alles in allem bin ich mit meiner Leistung zufrieden. Ich werde bei der IPA definitiv mehr Zeit in die Planung investieren & diese strikter durchführen (tägliche Arbeitsjournale, usw.), um Zeitverzug frühzeitig zu bemerken und verhindern.

3.8 Quellenverzeichnis

- Nuxt3: https://nuxt.com/
- Rust: https://www.rust-lang.org/learn
- Rocket.rs: https://rocket.rs/
- Stackoverflow: https://stackoverflow.com/
- CORS Fairing: https://stackoverflow.com/questions/62412361/how-to-set-up-cors-or-options-for-rocket-rs
- Argon2: https://www.argon2.com/
- fffuel: https://fffuel.co

3.9 Glossar

Begriff	Erklärung
API	Application Programming Interface: - Eine Schnittstelle, die die Kommunikation zwischen Systemen ermöglicht.
Hash	- Ist ein Wert, welcher aus dem Input generiert wird. Dieser hat eine fixe Länge und ist nicht entzifferbar.
CORS	Cross Origin Resource Sharing: - Ist ein Header-basierter Mechanismus welcher sogenannte «Origins» (Ursprünge) definiert und bei Requests / Responses mitschickt. Stimmt dieser Origin nicht überein, wird eine Preflight – Request (Sicherheitscheck) vorausgeschickt.

3.10 Code-Snippets mit Relevanz

3.10.1 # /src/helpers/parser.rs

```
/// Slams the user data through a model to make it easier to
use.
pub fn parse_user(u: Json<User>) -> Result<User, Error> {

    // generate a token if no token has been supplied.
    let token:String;
    if u.auth_token == None {
        token = token::generate(64);
    }else{
        token = u.auth_token.clone().unwrap();
    }

    // create a new User object and parse the data
    let data = User {
        _id: u._id.to_owned(),
        name: u.name.to_owned(),
        password: endecr::encrypt(u.password.to_owned()),
        email: u.email.to_owned(),
        role:u.role.to_owned(),
        auth_token:Some(token.to_owned()),
    };
    return Ok(data);
}

/// Parses the file data into an insertable type
pub async fn parse_file(file: &mut TempFile<'_>, owner:String)
-> File {
     // set in here and not in `connector.rs` since its only
     called here.
     let path = env::var("STOREPATH").expect("STOREPATH HAS TO
     BE SET");

     // filter out the filename and extension
     let name = file.raw_name().unwrap().as_str().unwrap();
     let extension = file.content_type().unwrap().0
     .extension().unwrap().as_str();
     let fullname = format!("{name}.{extension}");

     // make a persistent file out of the "temporary" one.
    let path = format!("{path}/{fullname}");
    let _ = file.persist_to(path.clone()).await;
```

```
    let data = File {
        _id: Some(ObjectId::new()),
        name: Some(fullname),
        path: Some(path),
        owned_by: Some(owner.to_owned()),
        size: Some(file.len().to_owned()),
    };

    data // <- Equal to "Return data;"
}
```

3.10.2 # /src/helpers/cookies.rs

```
/*
----- BISCUIT GENERATOR -----
-> We love cookie clicker, don't we?
*/

/// generate a new cookie based on the parameters
pub fn cookie(name: String, value: String) -> Cookie<'static>
{
    let cookie = Cookie::build(name, value)
    .path("/")
    .max_age(Duration::hours(3))
    .secure(true)
    .http_only(true)
    .same_site(SameSite::None)
    .finish(); // Setting the expiry date to 'None' sets it to
expire when the session gets closed.
    println!("{:?}", cookie);
    return cookie;
}
```

```
/// Used to extract value from cookie.
pub fn get_cookie_value(jar: &CookieJar<'_>, name: String) ->
String {
     let c = jar.get(&name);

     if c.is_some() {
          // cookies format: "key=value"
          // they have to be split into an array to get value
        return String::from(jar.get(&name).map(|cookie|
        cookie.value()).unwrap());
     }else {
        return String::from("");
     }
}
```

3.10.3 # /src/models/mod.rs

```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq)]
pub struct User {
    pub _id: Option<ObjectId>,
    pub name: Option<String>,
    pub password: String,
    pub email: String,
    pub role: Option<i32>, // 0 | 1: Admin | User
    pub auth_token: Option<String>, //generated by backend
}

#[derive(Serialize, Deserialize, Clone, Debug, PartialEq)]
pub struct File {
    pub _id: Option<ObjectId>,
    pub owned_by: Option<String>,
    pub name: Option<String>,
    pub path: Option<String>,
    pub size: Option<u64>,
}

#[derive(FromForm, Debug)]
pub struct Upload<'r>{
    pub file:TempFile<'r>
}
```

3.10.4 # /src/data/connector.rs

```
/*
----- SUMMARY -----
-> This is basically the Base Layer of the Backend.
*/

extern crate dotenv;

use std::{env, fs};
use dotenv::dotenv;
use mongodb::{
    bson::{doc, oid::ObjectId},
    Client,
    Collection,
    error::Error, results::{DeleteResult, InsertOneResult},
};
use rocket::{futures::TryStreamExt, http::Status};

use crate::{models::{File, User}};

pub struct Connector {
    user_col: Collection<User>,
    file_col: Collection<File>,
}

impl Connector {
    /*
    ----- INITIALIZATION OF DB CONNECTION -----
    */
    pub async fn init() -> Self {
        dotenv().ok();
        //change the var 'key' to change the uri (contained in
.env file)
        let uri = env::var("MONGOURI").expect("MONGOURI HAS TO
BE SET");

        let client = Client::with_uri_str(uri).await.unwrap();
        let db = client.database("nuclear");
        let user_col: Collection<User> =
db.collection("users");
        let file_col: Collection<File> =
db.collection("files");
        Connector {
            user_col,
```

```
            file_col,
        }
    }
}

/*
----- AUTH CHECKER -----
*/
impl Connector {
    /// Verify the authenticity of a request.
    pub async fn verify_auth(&self, token: String) ->
Result<User, bool> {
        let filter = doc! {"auth_token":token};

        let result = self.user_col.find_one(filter,
None).await;

        if let Ok(None) = result {
            return Err(false);
        } else {
            return Ok(result.unwrap().unwrap());
        }
    }

    /// verify that its an admin
    pub async fn _verify_admin(&self, token: String) ->
Result<User, bool> {
        let filter = doc! {"auth_token":token, "role":1};

        let result = self.user_col.find_one(filter,
None).await;

        if let Ok(None) = result {
            return Err(false);
        } else {
            return Ok(result.unwrap().unwrap());
        }
    }
}
```

```rust
/*
----- USER - RELATED METHODS -----
*/

impl Connector {
    /// insert a new user into the DB
    pub async fn create_user(&self, mut u: User) ->
Result<InsertOneResult, Error> {
        // 'oid switch' -> Generate an ObjectId if its empty.
        if u._id == None {
            u._id = Some(ObjectId::new());
        }

        let new = User {
            _id: u._id,
            name:u.name,
            password: u.password,
            email: u.email,
            role:u.role,
            auth_token: u.auth_token,
        };
        let user = self
            .user_col
            .insert_one(new, None)
            .await
            .ok()
            .expect("Error creating user");
        println!("{:?}", user);
        Ok(user)
    }

    /// get user based on password & email | used for login
(mainly)
    pub async fn get_user(&self, u: User) ->
Result<Option<User>, Error> {
        let filter = doc! { "email": u.email, "password":
u.password };
        let result = self.user_col.find_one(filter,
None).await?;
        match result {
            None => Ok(None),
            Some(res) => Ok(Some(res)),
        }
    }
```

```
    /// [ADMIN] - Delete User
    pub async fn delete_user(&self, u: User) ->
Result<DeleteResult, Error> {
        let filter = doc! {"_id":u._id};
        let result = self.user_col.delete_one(filter,
None).await?;
        return Ok(result);
    }

    /// [ADMIN] - Return a vector containing all users for
management.
    pub async fn _get_users(&self) -> Result<Vec<User>,
Status> {
        let mut cursor = self.user_col.find(None,
None).await.unwrap();

        let mut array: Vec<User> = Vec::new();

        while let Ok(Some(user)) = cursor.try_next().await {
            array.push(user);
        }

        return Ok(array);
    }
}
```