

Advanced Software Engineering - Dokumentation

docketStore



Dozent/in	Lars Briem
Abgabedatum	31.05.2021
Student	Nico Diefenbacher
Matrikelnummer	5949420

Inhaltsverzeichnis

Vorwort	4
App Grundlagen	4
Welche Frameworks werden eingesetzt ?	7
NestJS-Framework	7
Angular	7
MongoDB	7
Jest und Supertest	8
Installation & Start der App	8
Die Dokumenttypen	10
Von der Klasse zum Schema	10
Vom Schema zum Model	11
Programmierprinzipien	12
DRY	12
DRY bei Tests	12
DRY bei Services und Controller	13
SOLID	16
Liskov Substitution Principle	16
Interface Segregation Principle	16
Single Responsibility Principle	16
Open-Closed-Prinzip	17
GRASP	18
Indirection	18
Controller	19
Protected Variations	19
Entwurfsmuster	20
Dependency Injection	20
Decorators	21
E2E-Tests	21
ATRIP	21
Code Coverage	22
Mocks	23
Refactoring	23

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle sinngemäß und wörtlich übernommenen Textstellen aus fremden Quellen wurden kenntlich gemacht.

Gender Disclaimer

Ausschließlich zum Zweck der besseren Lesbarkeit wird auf die geschlechtsspezifische Schreibweise verzichtet. Alle personenbezogenen Bezeichnungen in dieser Hausarbeit sind somit geschlechtsneutral zu verstehen.

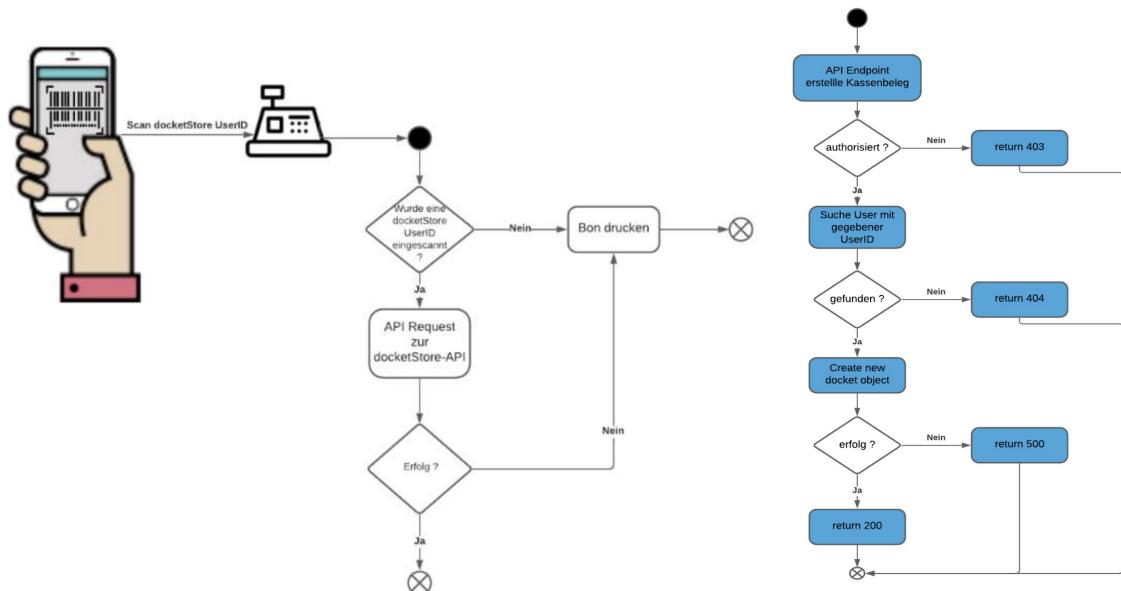
1. Vorwort

Ursprünglich hatte ich dir ein anderes Projekt vorgeschlagen. Ich hoffe das ist kein Problem. Zu Beginn der Arbeit an diesem Projekt habe noch nie mit den Frameworks oder mit den Programmiersprachen gearbeitet. Ich habe es als Gelegenheit gesehen Neues zu erlernen, da meine Programmiererfahrungen generell geringer ausfallen. Aus diesem Grund möchte ich schon im Voraus sagen, dass der Programmcode an einigen Stellen suboptimal ist, da mir das Wissen noch etwas fehlt.

2. App Grundlagen

docketStore ist eine Web-Applikation, mit der (digitale) Kassenbelege importiert, empfangen und verwaltet werden können. Es besteht die Möglichkeit Kassenbelege als PDF-File oder als Snapshot per Kamera in die Datenbank von docketStore zu importieren. Außerdem steht ein Endpoint bereit, der für externe Unternehmen das Ausstellen von Kassenbelegen ermöglichen soll. Jedes Unternehmen, das eine elektronische Registrierkasse führt muss der Belegausgabepflicht nachkommen. Aus rechtlichen Gründen muss immer sichergestellt sein, dass der Kassenbeleg dem Kunden auch wirklich ausgestellt wird.

Die untenstehenden Grafiken verdeutlichen die Funktionsweise von docketStore. Der Kunde lässt seinen eindeutigen Barcode von docketStore an der Kasse abscannen. Durch einen eindeutigen Prefix (aktuell noch nicht) des Barcodes erkennt die Kasse, dass es sich bei dem eingescannten Barcode um eine docketStore UserID handelt. Sobald solch ein Barcode abgescannt wurde, schickt die Kasse einen API Request an das docketStore Backend. Lediglich wenn das Backend von docketStore beim Ausstellen eines Kassenbeleges den Statuscode 200 zurückgibt wird der Kassenbeleg nicht ausgedruckt.



Neben dem Empfangen und Importieren von Kassenbelegen besteht die Möglichkeit einzelne Kassenbelege mit Tags zu markieren. Diese Tags dienen dazu, dass für spätere Zwecke wie beispielsweise bei Garantiefällen der Kassenbeleg nicht aufwändig gesucht werden muss. Mittlerweile gibt es einen elektronischen Kassenbeleg-Standard (EKaBS), der vorgibt wie die Inhalte von Kassenbelegen in die digitale Kassenbeleg-Dokumente eingebettet werden sollen. Aufgrund dessen wurde auf die Extraktion der Inhalte von Kassenbelegen bewusst verzichtet. Die Tags werden aus docketStore verschwinden, sobald der EKaBS in docketStore implementiert ist. Der EKaBS wird allerdings in der Wirtschaft noch nicht verwendet.

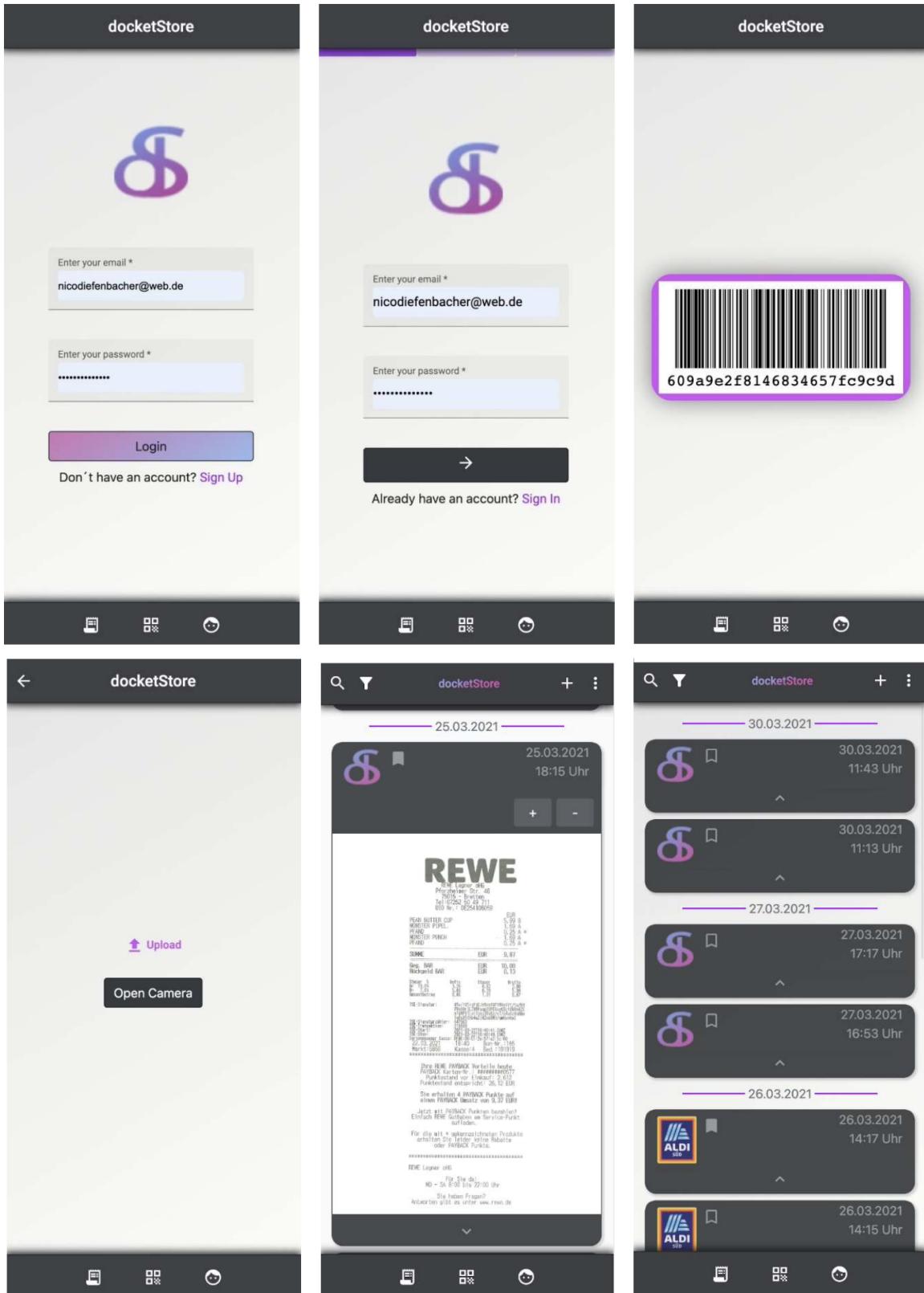
Zukünftig soll es auch möglich sein einzelne Kundenkonten verschiedener Unternehmen zu integrieren, sodass die Kassenbelege von den API's bezogen werden können. So ist das zumindest vom EKaBS vorgesehen. Allerdings gibt es beispielsweise weder von Rewe noch Lidl eine öffentliche Dokumentation der API's. Diesbezüglich muss noch abgewartet werden inwiefern die Kundenkonten zukünftig per API ansprechbar sind.

Die Applikation besteht aus einem Frontend und einem Backend. Da das Backend aus circa 2000 Zeilen Code besteht soll das Frontend nicht mitbewertet werden. Um das Backend zu Testen wird eine Postman Collection bei der Abgabe beigelegt.

Language	files	blank	comment	code
JSON	8	0	0	13230
TypeScript	39	438	9	1967
XML	5	0	0	365
Markdown	1	20	0	55
SUM:	53	458	9	15617

Abbildung - Das Backend hat circa 2000 Zeilen TypeScript Code.

Außerdem ist das Frontend noch nicht fertig, sodass es für eine Demonstration sinnvoll wäre. Um dennoch einen Eindruck vom Frontend zu bekommen habe ich hier ein paar Screenshots.



2.1. Welche Frameworks werden eingesetzt ?

Die Applikation besteht aus einem Frontend und einem Backend. Das Backend wurde mit NestJS entwickelt und arbeitet mit einer MongoDB.

2.1.1. NestJS-Framework

Die Anforderungen an den Programmentwurf konnten mit dem NestJS Framework erfüllt oder vereinfacht werden.

- **Objektorientierte Mainstream-Programmiersprache**
JavaScript beziehungsweise TypeScript ist eine objektorientierte Programmiersprache, welche überwiegend für die Webentwicklung verwendet wird.
- **DRY**
NestJS setzt auf NodeJS auf und abstrahiert viele Dinge die häufig Anwendung finden, sodass der Entwickler sich auf das Wesentliche konzentrieren kann.
- **SOLID**
NestJS arbeitet mit Modulen, Services, Controller, Middleware uvm. Dies ist besonders von Vorteil für das Single-Responsibility-Prinzip und das Open-Closed-Prinzip.
- **E2E-Tests**
NestJS nutzt Jest als Test-Framework. fokussiert Einfachheit und bietet die Möglichkeit E2E-Tests durchzuführen. In Verbindung mit Supertest können die API Endpunkte angesprochen werden. Jest bietet die Möglichkeit einzelne Services zu mocken.
- **Dependency Injection**
NestJS bietet die Möglichkeit für Dependency Injection auf sehr einfachem Weg.

2.1.2. Angular

Das Frontend wurde aufgrund von privatem Interesse in Angular entwickelt und soll auch nicht Bestandteil der Bewertung sein.

2.1.3. MongoDB

docketStore benötigt eine Datenbank um die Informationen dauerhaft speichern zu können. Dafür kann theoretisch jede beliebige Datenbank verwendet werden. Aufgrund der guten Kompatibilität mit NestJS wurde eine MongoDB gewählt. MongoDB ist eine dokumentbasierte NoSQL-Datenbank. Dadurch können Sammlungen von JSON-ähnlichen Dokumenten verwaltet und auf natürliche Weise modelliert werden.

Die Daten können zwar in komplexen Hierarchien verschachtelt werden, sind dabei aber immer abfragbar und indizierbar. Das NPM-Package “mongoose” wird verwendet um die MongoDB anzusprechen.

2.1.4. Jest und Supertest

Jest wird in docketStore zum automatisierten Ausführen von Tests genutzt. Mit Supertest werden Requests an die zu testenden Endpunkten ausgeführt. Unit-Tests sind auch Bestandteil der Anwendung, werden aber aufgrund von Zeitmangel erst nach der Abgabe dieser Hausarbeit implementiert. Hauptsächlich werden E2E-Tests durchgeführt. Wie bereits erwähnt achtet Jest auf die Einfachheit. Dadurch können Tests übersichtlich und mit geringer Komplexität geschrieben werden.

Um die in den Tests benötigten Abhängigkeiten einer Klasse minimal zu simulieren, werden diese gemockt. Das Jest-Framework eignet sich dafür, NestJS-Services zu mocken. Allerdings wird beim E2E-Test versucht so wenig Services wie möglich zu mocken, um einen realitätsnahen E2E-Test durchführen zu können. In den Unit-Tests können Mocks häufiger benutzt werden, da es lediglich um das Testen einer Unit geht. Die erwarteten Ergebnisse mit “expect” aus Jest kontrolliert.

3. Installation & Start der App

Zur Installation aller benötigten Programme muss lediglich das installationsskript ausgeführt werden. Linux erkennt dies öfter nicht als ausführbare Datei, weshalb der Befehl “chmod +x dateiname” ausgeführt werden muss.

```
chmod +x docketstore_installation_script.sh
```

Danach können Skripte ausgeführt werden

```
./docketstore_installation_script.sh (installiert Angular, NPM, MongoDB, NestJS und die Applikation an sich)
```

Um die Applikation zu starten führe einfach folgenden Befehl aus:

```
npm run start
```

Wenn das Backend gestartet wird, ist dieses über <http://localhost:3000/> erreichbar.

Nachdem der Benutzer angemeldet ist kann der User

- Kassenbelege importieren
- Kassenbelege empfangen
- Kassenbelege mit Tags markieren
- Tags von Kassenbelegen entfernen
- Kassenbelege Löschen
- Tags Erstellen
- Tags Löschen
- Tags umbenennen

4. Die Dokumenttypen

Dieses Kapitel möchte ich nutzen um die Dokumenttypen der Datenbank zu erläutern. Es existieren aktuell die Dokumente User, Docket, Tag und MailVerification.

In einem UserDokument werden alle relevanten Informationen zum User gespeichert. Dabei existieren zwei Typen den App-User und den B2B-User. Die Unterscheidung erfolgt über Rollen. Das DocketDokument beinhaltet alle Informationen über einen Kassenbeleg. Dazu gehören die Tags, mit denen der Kassenbeleg markiert wurde sowie die JSON-Spezifikation des elektronischen Kassenbeleg-Standards und das Kassenbeleg PDF-Document.

Das MailVerificationDocument wird beim erstellen eines Accounts bzw. beim Ändern der E-Mail-Adresse erzeugt und beinhaltet den Bestätigungscode den der User per Mail zugesendet bekommt um seine E-Mail-Adresse zu bestätigen. Das TagDocument beinhaltet lediglich einen Namen. Dockets können dann mit einem oder mehreren Tags markiert werden.

Zukünftig soll es auch die ExternalAPIAccountDokumente geben. Diese speichern Credentials für externe Kundenkonten, um über die jewilige API die digitalen Kassenbelege beziehen zu können.

4.1. Von der Klasse zum Schema

NestJS bietet Dekorierer mit denen eine Klasse schnell zu einem Datenbankschema konvertiert werden kann. Die Klasse wird mit `@Schema()` dekoriert. Der Dekorierer `@Prop()` sagt aus, dass das Attribut Teil des Schemas sein soll. Mittels `SchemaFactory.createForClass()` wird aus der Klasse das Schema erstellt. Alternativ kann auch `new Schema()` von "mongoose" aufgerufen werden.

```
export type MailDocument = Mail & Document;

@Schema()
export class Mail {
    @Prop()
    receiverId: string;
    @Prop()
    code: string;
    @Prop()
    dateCreated: Date;
}

export const MailSchema = SchemaFactory.createForClass(Mail);
```

4.2. Vom Schema zum Model

Nun existiert ein Schema, welches in jedes Modul importiert werden kann. Die Schemas werden folgendermaßen in ein Modul importiert.

```
MongooseModule.forFeature({ models: [{name: 'Users', schema: UserSchema}]}),
MongooseModule.forFeature({ models: [{name: 'Mails', schema: MailSchema}]}),
```

Nachdem ein Schema in einem Modul importiert wurde, kann es in sämtliche Provider injiziert werden. Beispielsweise wird im AuthModul das UserSchema im AuthService genutzt. Das Schema kann über den @InjectModel() Dekorierer injiziert werden.

```
@Injectable()
export class AuthService {
    private logger = new Logger('AuthService');

    constructor(
        private readonly mailService: MailVerificationService,
        private readonly userService: UserService,
        @InjectModel('Users') private readonly userModel: PassportLocalModel<User>,
        @Inject('JWT_CONFIG') private readonly JWT_CONFIG: JwtConfig
    ) {}
}
```

Jedes Model bietet alle grundsätzlichen Datenbankoperationen wie Create, Delete, Update und Find, dank “mongoose”. Diese Operationen sind in fast allen Anwendungen notwendig und müssen deshalb nicht erneut programmiert werden. Jedes Dokument besitzt eine eindeutige String-ID.

5. Programmierprinzipien

In diesem Kapitel werden die Programmierprinzipien GRASP, SOLID und DRY in docketStore beschrieben.

5.1. DRY

Don't Repeat Yourself ist ein wichtiges Prinzip in der Softwareentwicklung. Denn dieses Prinzip versucht den Programmcode klein zu halten, da wiederholende Ereignisse separat ausgelagert werden sollen.

5.1.1. DRY bei Tests

Beim Testen der Software habe ich Mock Objekte erstellt, um wiederkehrende Objekte nicht erneut schreiben zu müssen. Beispielsweise wurden für das Testen des Authentifizierungsmoduls verschiedene Data Transfer Objekte (DTOs) erstellt, um einen User erstellen zu können. Außerdem wurden Mocks von Services erstellt, die wiederkehrend verwendet werden. Beispielsweise wird der MailService in jedem Fall gemockt, da das Versenden von Willkommen Mails mit einem Bestätigungscode für das Testen der Software nicht erforderlich ist.

```
export const createAppUserDto = new CreateAppUserDto({ username: "nicodiefenbacher@web.de", password: "password1", lastName: "Diefenbacher", phoneNumber: "015904379121" })
export const loginAppDto = {username: "nicodiefenbacher@web.de", password: "password123"}
export const falseLoginDto1 = {username: "nicodiefen@web.de", password: "password12"}
export const falseLoginDto2 = {username: "nicodiefen@web.de", password: "password123"}
export const falseLoginDto3 = {username: "nicodiefenbacher@web.de", password: "password13"}
export const createB2BUserDto = new CreateB2BUserDto({ username: "nicodiefuse@web.de", password: "password123", lastName: "Diefenbacher", phoneNumber: "015904379121", contactEmail: "nicodiefuse@web.de", companyName: "TestCompany" })

export const createB2BUserErrorDto = new CreateB2BUserDto({ username: "blabla@web.de", password: "password123", lastName: "Diefenbacher", phoneNumber: "015904379121", contactEmail: null, companyName: "TestCompany", category: "test" })

export const loginB2BDto = {username: "nicodiefuse@web.de", password: "password123"}

export const registerCodeMock = "123456789"
export const mailServiceMock = {
    sendWelcomeEmail: () => Promise.resolve({ value: true }),
}

export const codeGeneratorServiceMock = {
    generateCode: () => registerCodeMock,
}
```

Neben den Mocks habe ich auch eine Klasse TestHelper geschrieben, die Methoden enthält, um eine TestSuite zu initialisieren. Beispielsweise wird für das Importieren von Dockets einen User benötigt. Statt den User in jeder TestSuite in der er benötigt wird zu erstellen, habe ich diesen Ablauf ausgelagert, um in den Test Files eine bessere Übersicht zu erreichen. Ein weiteres Beispiel hierfür ist die Methode createTag(). Im Docket E2E Test existiert ein Testcase, der ein Docket mit einem Tag markiert. Dafür wird ein Tag benötigt. Beim Weiterentwickeln der Software wird womöglich in anderen Tests ebenfalls ein Tag benötigt. Durch diese Methode kann der Tag einfach erstellt werden.

```
beforeAll( fn: async () => {
    testHelper = new TestHelper();
    await testHelper.createTestingModule();
    await testHelper.dropTestDatabase();
    await testHelper.createAndActivateAppUser();
    await testHelper.createAndActivateB2BUser();
    await testHelper.loginAppUser();
    await testHelper.loginB2BUser();
    await testHelper.createTag()

});
```

5.1.2. DRY bei Services und Controller

Oftmals werden gleiche Codestücke mehrmals verwendet. Daher ist es sinnvoll diese gleichen Codestücke zu separieren. Außerdem kann dieses Prinzip auch der Übersichtlichkeit des Codes dienen.

Es existieren aktuell zwei Endpunkte für das Erstellen von Usern. Einen für das Erstellen eines AppUsers und einen für das Erstellen eines B2BUsers. Für das Erstellen eines B2B Users werden zusätzliche Informationen benötigt. Außerdem werden B2B-User nicht per Mail aktiviert, sondern erst nach einer Prüfung von docketStore. Dadurch kann sichergestellt werden, dass das auch wirklich das Unternehmen ist, was es vorgibt zu sein. Allerdings wird der B2B-User aktuell noch nicht benötigt, weshalb die Aktivierung noch per Mail stattfindet. Da für die unterschiedlichen Usertypen verschiedene Algorithmen notwendig sind werden zwei Endpunkte benötigt. Allerdings durchlaufen beide Endpunkte einen ziemlich ähnlichen Algorithmus. Aus diesem Grund habe ich die Codestücke in Methoden zusammengefasst, um eine bessere Übersichtlichkeit zu erhalten und den Programmcode zu reduzieren. Sobald der B2B-User relevant ist werden sich die Algorithmen auch mehr unterscheiden.

```

async registerAppUser(userToRegister: CreateAppUserDto): Promise<IResponse> {
    const result: IResponse = {
        status: HttpStatus.OK,
        success: false,
        data: {message: ""}
    }
    const user = await this.userService.findOne({email: userToRegister.username});
    if (user) {
        result.status = HttpStatus.BAD_REQUEST
        result.data.message = "User with given e-mail already registered";
    } else {
        const user = await this.userModel.register(new this.userModel(
            {
                username: userToRegister.username,
                email: userToRegister.username,
                contact: userToRegister.contact,
                status: "pending",
                roles: ['app_user'],
                // @ts-ignore
                lastLogin: new Date()
            }, userToRegister.password);
        if (user.errors) {
            result.data.message = user.errors;
        } else if (user) {
            const created_user = await this.userService.findById(user.id);
            let mailVerificationDto: MailVerificationDto = {
                receiver: created_user,
                code: this.mailService.generateCode(10),
                dateCreated: new Date()
            };
            result.success = await this.mailService.create(mailVerificationDto);
            result.data.message = "User successfully created!";
        }
    }
    return result;
}

```

```

@Post( path: "/app/register")
public async registerAppUser(@Res() res, @Body() createUserDto: CreateAppUserDto): Promise<any> {
    const userExists = await this.userService.existsUsername(createUserDto.username)
    if (userExists) {
        return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.BAD_REQUEST));
    } else {
        const user = await this.userService.createUser(createUserDto);
        if (user) {
            const mailVerification = await this.mailVerificationService.create(user._id);
            if (mailVerification) {
                const sentMail = await this.mailService.sendWelcomeEmail(user._id, user.username, user.firstName);
                if (sentMail) {
                    return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id))
                }
            } else {
                return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id))
            }
        } else {
            return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.INTERNAL_SERVER_ERROR))
        }
    }
}

```

Die obere Grafik repräsentiert den selben Endpunkt (registerAppUser) wie die untere Grafik. Allerdings wurde in der unteren, also der neueren Version, das DRY-Prinzip angewandt. Beispielsweise wurde das Erstellen des userModels in den UserService ausgelagert, da beim erstellen eines B2B-Users ebenfalls ein userModel erstellt werden muss. Auch das erstellen der MailVerificationDto wurde in den MailVerificationService ausgelagert, da vom Endpunkt lediglich die erstellte UserID benötigt wird, um eine MailVerificationDto zu erstellen.

Des Weiteren werden einzelne Routen oder ganze Controller über einen RolesGuard und einen RolesDecorator protected. Stattdessen hätte man bei jedem Endpunkt im Algorithmus die Rollen des Users abfragen müssen. Das DRY-Prinzip würde sich etwas verletzt fühlen, wenn die Rollenabfrage vor jedem eigentlichen Algorithmus angeführt werden muss.

```
export enum Role {
  B2B_USER = 'b2b_user',
  APP_USER = 'app_user',
  MODERATOR = 'MODERATOR',
}

export const Roles = (...roles: string[]) => SetMetadata(metadataKey: 'roles', roles);
```

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<string[]>(`metadataKey: 'roles', context.getHandler());
    if (!roles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    const hasRole = () =>
      !!user.roles.find(role => !!roles.find(item => item === role));

    return user && user.roles && hasRole();
  }
}
```

5.2. SOLID

In diesem Abschnitt werden Code-Beispiele der SOLID-Prinzipien in docketStore gezeigt.

5.2.1. Liskov Substitution Principle

```
@Injectable()
export class JwtAuthGuard extends AuthGuard( type: 'jwt' ) {}
```

Einzelne Endpunkte können mit einem AuthGuard versehen werden, um unberechtigter Nutzung zu widerstehen. Um nicht jedes mal AuthGuard('jwt') im Code hinterlegen zu müssen, habe ich eine neue Klasse erstellt, die von AuthGuard('jwt') erbt. Dies reduziert u.A. auch die Fehleranfälligkeit, da der Type nur einmal angegeben werden muss und nicht öfters.

5.2.2. Interface Segregation Principle

docketStore besitzt zwar Interfaces, jedoch keine, die Funktionalität weitergeben. Demnach konnte das Interface Segregation Principle nicht angewendet werden.

5.2.3. Single Responsibility Principle

Nach diesem Prinzip soll jedes Klasse nur eine Zuständigkeit haben. Aus diesem Grund wurden einige Klassen bzw. Services aufgespalten, um die Zuständigkeiten einer Klasse zu reduzieren. Beispielsweise existiert in der Anwendung ein MailService. Dieser MailService war ursprünglich für die MailVerificationDokumente (CRUID) zuständig sowie das Versenden von Willkommen Mails. Beim Testen der Software wurde mir dann klar, dass es verschiedene Zuständigkeiten sind. Aus diesem Grund habe ich den Service in einen MailVerificationService und in einen MailService geteilt. Dadurch konnte ich beim Testen der Software den MailService, der die Mails versendet, mocken und den MailVerificationService vollständig testen ohne ihn zu mocken.

Ein weiteres Beispiel für das Single-Responsibility-Prinzip ist der CodeGeneratorService. Aktuell benötigt die Software lediglich an einem Punkt in der Software einen zufällig generierten Code. Die Methode war ursprünglich im MailService enthalten. Nun habe ich dafür einen eigenen CodeGeneratorService, um die Zuständigkeiten einer Klasse so gering wie möglich zu halten.

Der Commit kann hier eingesehen werden:

<https://github.com/ndthmdg98/docketStore-backend/commit/bd2b4d5d557d20f4468fd7ea3a c1f3120c8bcef1>

Die Aufgaben der Services lauten wie folgt:

- **AuthService**: Beinhaltet alle relevanten Authentifizierungsmethoden login, register, verify.
- **UserService**: DatenbankService für UserDokumente.
- **MailService**: Beinhaltet Methoden zum Versenden von Mails (sendWelcomeMail). Dieser Service Nutzt das MailerModul, dass im Modul importiert werden muss.
- **TagService**: DatenbankService für TagDokumente
- **DocketService**: DatenbankService für DocketDokumente
- **ExternalAPIService**: DatenbankService für ExternalAPIAccountDokumente. Außerdem beinhaltet er aktuell noch die fetch()-Methode, mit der die Externen API's angesprochen werden sollen. Allerdings wäre es im Sinne der Single-Responsibility besser dafür einen eigenen Service zu schreiben.
- **Code-GeneratorService**: Service um Random Codes zu erzeugen.
- **MailVerificationService**: DatenbankService für MailVerificationDokumente
- **JwtStrategy**: Strategy zur Jwt Authentifizierung.
- **LocalStrategy**: Strategy für Username & Password Authentifizierung

Im Bezug zum Single Responsibility Prinzip sollte der AuthService in einen LoginService, RegisterService und VerifyAccountSevice aufgeteilt werden.

5.2.4. Open-Closed-Prinzip

Jeder Service hat seine eigene Zuständigkeit, wobei es bei dem ein oder anderen Service noch Verbesserungsbedarf gibt. Jede Methode und jeder Service wurde von mir so geschrieben, dass so wenig Abhängigkeiten wie möglich zwischen den Services gibt, um so den Service nach außen hin für Erweiterungen zu öffnen und die Modifikationen innerhalb der Klasse so gering wie möglich zu halten.

Der UserService ist beispielsweise nur vom UserModel abhängig. Beim LocalStrategyService wird die Methode validateUsernameWithPassword() aus dem AuthService aufgerufen. Statt findByUsername() in dieser Methode aufzurufen, hätte man auch direkt das userModel ansprechen können. Allerdings ist dann der AuthService abhängig vom userModel. So ist der AuthService nicht abhängig vom UserModel.

```

async validateUsernameWithPassword(username: string, password: string): Promise<boolean> {
    const user = await this.userService.findByUsername(username);
    const authenticationResult: AuthenticationResult = await user.authenticate(password);
    if (authenticationResult.error) {
        return false;
    } else if (authenticationResult.user) {
        return true;
    }
}

async findByUsername(username: string): Promise<any> {
    return await this.usermodel.findOne({ conditions: {username: username}}).exec();
}

```

5.3. GRASP

In diesem Kapitel werden die eingesetzten GRASP-Regeln aufgezeigt, die zu geringer Kopplung und hoher Kohäsion führen.

5.3.1. Indirection

In docketStore könnte jeder Controller die DatenbankSchemas direkt aufrufen, um Objekte abzuspeichern, zu bearbeiten oder zu löschen (vgl. ursprüngliche

```

@Post( path: "/app/register")
public async registerAppUser(@Res() res, @Body() createUserDto: CreateAppUserDto): Promise<any> {
    const userExists = await this.userService.existsUsername(createUserDto.username)
    if (userExists) {
        return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.BAD_REQUEST));
    } else {
        const user = await this.userService.createUser(createUserDto);
        if (user) {
            const mailVerification = await this.mailVerificationService.create(user._id);
            if (mailVerification) {
                const sentMail = await this.mailService.sendWelcomeEmail(user._id, user.username, user.firstName);
                if (sentMail) {
                    return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id))
                }
            } else {
                return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id))
            }
        } else {
            return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.INTERNAL_SERVER_ERROR))
        }
    }
}

```

registerAppUser-Methode). Stattdessen werden diese Operationen an die zugehörigen Services delegiert, die diese Aufgaben dann übernehmen. Dies sorgt für eine bessere Schichtenarchitektur.

5.3.2. Controller

docketStore besitzt insgesamt 3 Controller. Einen AuthController, DocketController und einen TagController. Jeder Controller besitzt mehrere Endpunkte. Sie vermitteln die Benutzereingaben zur zugehörigen Businesslogik, indem sie ihre Aufgaben an die Services der zu Grunde liegenden Applikationsschicht delegieren.

DocketController: `/docket` - Mit diesem Controller können Kassenbelege importiert, erstellt, gelöscht und gesucht werden. Außerdem können Dockets mit Tags markiert werden.

TagController: `/tag` - Mit diesem Controller können Tags erstellt und gelöscht werden.

AuthController `/auth` - Dieser Controller ist vollständig für die Authentifizierung zuständig.

Alle Endpunkte der verschiedenen Controller können aus der beigelegten Postman Collection oder aus dem Screenshot entnommen werden.

```
DocketController {/docket}:
  Mapped {/docket/import, POST} route
  Mapped {/docket/create/:receiverId, POST} route
  Mapped {/docket, GET} route
  Mapped {/docket/:id, GET} route
  Mapped {/docket/tag/:tagId, GET} route
  Mapped {/docket/:docketId/mark/:tagId, PUT} route
  Mapped {/docket/:docketId/unmark/:tagId, PUT} route
  Mapped {/docket/:id, DELETE} route
TagController {/tag}:
  Mapped {/tag, POST} route
  Mapped {/tag, GET} route
  Mapped {/tag/:id, GET} route
  Mapped {/tag/:id, PUT} route
  Mapped {/tag/:id, DELETE} route
AuthController {/auth}:
  Mapped {/auth/app/register, POST} route
  Mapped {/auth/b2b/register, POST} route
  Mapped {/auth/login, POST} route
  Mapped {/auth/:user/:code, GET} route
  Mapped {/auth/profile, GET} route
```

5.3.3. Protected Variations

Das Abspeichern und Laden der Objekte aus der Datenbank wird von den jeweiligen Services übernommen. Diese werden per DI in den Service injiziert. Dadurch beschränkt sich die Abhängigkeit der Datenbank auf die jeweiligen Services. Diese geringe Kopplung sorgt dafür, dass die Datenbank gewechselt werden kann, ohne dass der Applikations-Code groß angepasst werden muss.

6. Entwurfsmuster

6.1. Dependency Injection

In Folgenden wird anhand von Beispielen das Prinzip der Dependency Injection in docketStore aufgezeigt.

Jeder Datenbank Service besitzt beispielsweise eine Abhängigkeit zum jeweiligen DatenbankSchema. Diese DatenbankSchemas werden im Konstruktor injected. Beispielsweise wird hier im AuthService die JwtConfig und das UserModel injected. Diese Injection Token werden in der Moduldefinition angegeben und können im gesamten Modul verwendet bzw. injiziert werden.

```
providers: [
    AuthService,
    UserService,
    LocalStrategy,
    JwtStrategy,
    MailVerificationService,
    MailService,
    CodeGeneratorService,
    {
        provide: 'JWT_CONFIG',
        useValue: JWT_DI_CONFIG
    },
    {
        provide: 'MAIL_CONFIG',
        useValue: MAIL_DI_CONFIG
    }
],
```

Ausschnitt aus der AuthModul Definition. Die Dependency Injection Token werden definiert.

```
@Injectable()
export class AuthService {
    private logger = new Logger('AuthService');

    constructor(
        private readonly mailService: MailVerificationService,
        private readonly userService: UserService,
        @InjectModel('Users') private readonly userModel: PassportLocalModel<User>,
        @Inject('JWT_CONFIG') private readonly JWT_CONFIG: JwtConfig
    ) {
    }
}
```

Ausschnitt aus dem AuthService, der die Injection Token aus dem AuthModul nutzt.

6.2. Decorators

Wie bereits beim Kapitel DRY erwähnt, werden Decorators in docketStore genutzt. Der RolesDecorator wird genutzt, um notwendige Rollen für das Zugreifen auf eine Ressource festlegen zu können. Vom RolesGuard werden die im Decorator angegeben Rollen und die des Users verglichen.

7. E2E-Tests

In diesem Kapitel werden die E2E-Tests genauer beschrieben.

7.1. ATRIP

Jeder Test von docketStore muss den ATRIP-Anforderungen entsprechen. Dies wird im Folgenden an Beispielen gezeigt.

Automatic: Es soll automatisiert ausgewertet werden, ob der Test erfolgreich war oder fehlgeschlagen ist. Durch `expect(response).toBe(value)` wird automatisch sichergestellt, dass die Response auch wirklich das zurückgibt was erwartet wird.

```
it(`should register app user`, fn: () => {  
  return request(app.getHttpServer())  
    .post('/auth/app/register')  
    .send(createAppUserDto)  
    .expect(status: 200)  
    .then(res => {  
      const response = res.body;  
      expect(response.data).toBeDefined()  
      expect(response.success).toBeTruthy()  
      expect(response.statusCode).toBe(expected: 200)  
      createdAppUserId = response.data;  
    })  
});
```

Abbildung - Register User Test

Thorough: Alle Funktionalitäten soll getestet werden, vor allem die Haupt-UseCases der Anwendung. Es wurden für alle Haupt-UseCases Tests geschrieben. Diese sollten funktionieren oder auch nicht je nach Eingabe.

Professional: Die Tests sind nach ihren Funktionalitäten bzw. nach Ihrem Resultat benannt. Die Ausführungszeit ist kurz sodass die Tests zwischendurch ausgeführt werden können.

Repeatable: Bedeutet, dass die Tests bei jedem ausführen die gleichen Ergebnisse erzeugen. Dadurch wird gewährleistet, dass immer die gleichen Objekte zum Testen verwendet werden.

Independent: Die Tests sind unabhängig voneinander und bauen nicht aufeinander auf. Die Tests testen jeweils nur eine Funktionalität, nach welcher sie benannt sind, sodass bei einem Fehlschlag direkt gesehen werden kann, an welcher Codestelle der Fehler liegt.

```

it(`name: 'should not import a docket to the authorized (b2b) user`, fn: async () => {
  const file = fs.readFileSync(path: "/Users/nicodiefenbacher/WebstormProjects/docketStore/docket.pdf");
  return request(app.getHttpServer()) SuperTest<Test>
    .post(url: '/docket/import') Test
    .set('Authorization', 'bearer ' + b2bJwtToken) Test
    .attach(field: 'file', file, options: 'file.pdf') Test
    .expect(status: 403) Test
    .then(res => {
      const response = res.body;
      expect(response.error).toBe(expected: "Forbidden")
      expect(response.statusCode).toBe(expected: 403)
    })
})
}

```

Abbildung - Ein Test der den ATRIP Anforderungen entspricht

7.2. Code Coverage

Bisher wurden lediglich für den AuthController und für den DocketController E2E-Tests geschrieben. Es wurde noch nicht für die komplette Businesslogik (Application) Tests geschrieben. Allerdings wird die Code Coverage noch weiter ausgebreitet, sodass alle relevanten Teile getestet sind.

 auth.controller.ts	86% line...
 auth.module.ts	78% line...
 auth.service.ts	100% lin...
 code-generator.service.ts	25% line...
 jwt-auth.guard.ts	100% lin...
 jwt-strategy.service.ts	91% line...
 local.strategy.ts	66% line...
 local-auth.guard.ts	
 mail-verification.service.ts	81% line...
 user.service.ts	70% line...

 docket.controller.ts	75% line...
 docket.module.ts	100% lin...
 docket.service.ts	70% line...

Die Coverage der Tests kann durch Ausweitung der TestCases erhöht werden.

7.3. Mocks

In den Tests werden Mocks verwendet. Beispielsweise wird beim Registrieren eines Users eine Bestätigungsmaile gesendet, welche mittels Mocks umgangen wird, da es nicht relevant für das Funktionieren des Codes ist.

```
export const mailServiceMock = {
    sendWelcomeEmail: () => Promise.resolve({ value: true}),
};
```

Abbildung - Mail-Service wird gemockt

```
it(`name: 'should register app user', fn: () => {
    return request(app.getHttpServer()) SuperTest<Test>
        .post(url: '/auth/app/register') Test
        .send(createAppUserDto) Test
        .expect(status: 200) Test
        .then(res => {
            const response = res.body;
            expect(response.data).toBeDefined()
            expect(response.success).toBeTruthy()
            expect(response.statusCode).toBe(expected: 200)
            createdAppUserId = response.data;
        })
});
```

Abbildung - Das TestAppModul wurde mit dem gemockten Service ersetzt und returned nun einfach true beim Aufruf von sendWelcomeMail().

8. Refactoring

Wie bereits am Anfang erwähnt, habe ich Javascript bzw. Typescript vor diesem Projekt noch nie zuvor verwendet. Generell fällt meine Programmiererfahrung gering aus. Aus diesem Grund wurden der Code von mir ständig verbessert, da ich ständig neue Dinge gelernt habe. Im Folgenden werden ein paar Code Smells, die im Laufe der Programmierung von docketStore aufgetreten sind, aufgezeigt.

```

@Post(path: "/app/register")
public async registerAppUser(@Res() res, @Body() createUserDto: CreateAppUserDto): Promise<any> {
    const userExists = await this.userService.findByUsername(createUserDto.username)
    if (userExists) {
        return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.BAD_REQUEST, message: "User with Given Mail already exists"));
    } else {
        const user = await this.userService.createUser(createUserDto);
        if (user) {
            const mailVerification = await this.mailVerificationService.create(user._id);
            if (mailVerification) {
                const sentMail = await this.mailService.sendWelcomeEmail(user._id, user.username, user.firstName, mailVerification.code);
                if (sentMail) {
                    return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id));
                }
            } else {
                return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id));
            }
        } else {
            return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.INTERNAL_SERVER_ERROR));
        }
    }
}

@Post(path: "/b2b/register")
public async registerB2bUser(@Res() res, @Body() createUserDto: CreateB2BUserDto): Promise<any> {
    const userExists = await this.userService.findByUsername(createUserDto.username)
    if (userExists) {
        return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.BAD_REQUEST, message: "User with Given Mail already exists"));
    } else {
        const user = await this.userService.createUser(createUserDto);
        if (user) {
            const mailVerification = await this.mailVerificationService.create(user._id);
            if (mailVerification) {
                const sentMail = await this.mailService.sendWelcomeEmail(user._id, user.username, user.firstName, mailVerification.code);
                if (sentMail) {
                    return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id));
                }
            } else {
                return res.status(HttpStatus.OK).json(APIResponse.successResponse(user._id));
            }
        } else {
            return res.status(HttpStatus.OK).json(APIResponse.errorResponse(HttpStatus.INTERNAL_SERVER_ERROR));
        }
    }
}

```

Aktuell existieren noch zwei Endpunkte für das erstellen von Usern. Allerdings bin ich der Meinung, dass das auch in einem Endpunkt geht, da der Code zu 90% gleich ist. Jedoch werden verschiedene Data Transfer Objekte (DTOs) benötigt mit verschiedenen Validationskriterien für verschiedene Usertypen. Inwiefern eine DTO für verschiedene User Typen möglich ist, weiß ich noch nicht. Ich habe dies aus Zeitproblemen vernachlässigt.

Beim Testen der Software habe ich wie bereits erwähnt einen Testhelper geschrieben, der eine bessere Übersicht in den Testfiles ermöglichen soll. Ursprünglich diente der TestHelper nur zum erstellen des Testmoduls. Irgendwann habe ich gemerkt, dass die ganzen Requests, die bei den E2E-Tests wiederkehrend aufgerufen werden ausgelagert werden können. Folgende Abbildung zeigt einen Test, bei dem der Request im Test File aufgerufen wird. Diese Aufrufe sind konstant bis auf die Eingaben. Aus diesem Grund können die Requests ausgelagert werden und über den TestHelper aufgerufen werden, um eine bessere Übersichtlichkeit in den Tests zu haben.

```

it(`name: `should register app user`, fn: () => {
    return request(testHelper.app.getHttpServer())
        .post({ url: '/auth/app/register' })
        .send(createAppUserDto)
        .expect({ status: 200 })
        .then(res => {
            const response = res.body;
            expect(response.data).toBeDefined()
            expect(response.success).toBeTruthy()
            expect(response.statusCode).toBe(expected: 200)
            createdAppUserId = response.data;
        })
});

```

Abbildung - Request im Test

Allerdings werden noch nicht alle Tests nach diesem Schema aufgerufen. Im Testhelper können Methoden für jeden beliebigen Endpoint geschrieben werden, sodass die Testfiles lediglich das überprüfen der erwarteten Werte übernimmt.

```

async deleteDocket(docketId: string) {
    const url = `/docket/${docketId}`;
    const res = await request(this.app.getHttpServer())
        .delete(url)
        .set('Authorization', `bearer ${this.appJwtToken}`)
        .send();
    return res.body;
}

it(`name: `should delete a docket`, fn: async () => {
    const body = await testHelper.deleteDocket(docketId)
    expect(body.statusCode).toBe(expected: 200)
    expect(body.success).toBeTruthy()
    expect(body.data).toBeUndefined()
    const getTagBody = await testHelper.getDocket(docketId);
    expect(getTagBody.statusCode).toBe(expected: 404)
    expect(getTagBody.success).toBeFalsy()
    expect(getTagBody.data).toBeUndefined()
})
;
```

Abbildung - Request im TestHelper

Der Request wurde vom Testhelper aufgerufen und gibt lediglich den Response-Body zurück. Im Test wird dann der Response-Body nach Richtigkeit überprüft.

docketStore Klassendiagramm

