Clockwise

Entwurf

Intuitive Time Tracking

Betreuer

Maximilian Li

Team

Dominik Pollok
Phil Gengenbach
Alina Petri
José Ayala
Johann Kohl

21. Juni 2024

Intuitive Robots Lab

Inhaltsverzeichnis

1 Einleitung

Das primäre Ziel dieses Projekts ist die Entwicklung einer Webanwendung, die speziell darauf ausgerichtet ist, die Zeiterfassung für wissenschaftliche Hilfskräfte intuitiv und effizient zu gestalten. Diese Entwurfsdokumentation soll eine klare und strukturierte Übersicht über das Projekt bieten. Sie schließt nahtlos an die Anforderungsspezifikation an und gewährleistet, dass alle Komponenten des Projekts gut zusammenarbeiten. Dieses Dokument wird detailliert darlegen, wie das Projekt strukturiert ist, indem es in verschiedene Klassen und Pakete unterteilt wird. Dies ist von entscheidender Bedeutung, um zu verstehen, wie jeder Teil des Projekts funktioniert und wie diese zusammenpassen.
Darüber hinaus bietet dieses Dokument Einblicke in die Architektur der Software. Es erklärt, warum bestimmte Entwurfsentscheidungen getroffen wurden und wie diese Entscheidungen dazu beitragen, die Ziele des Projekts zu erreichen. Der Fokus liegt darauf, die Struktur der Software effizient, wartbar und flexibel genug zu gestalten, um zukünftige Verbesserungen oder Erweiterungen zu ermöglichen.

2 Architektur

2.1 Model View Controller Service (MVCS)

2.1.1 Warum MVCS?

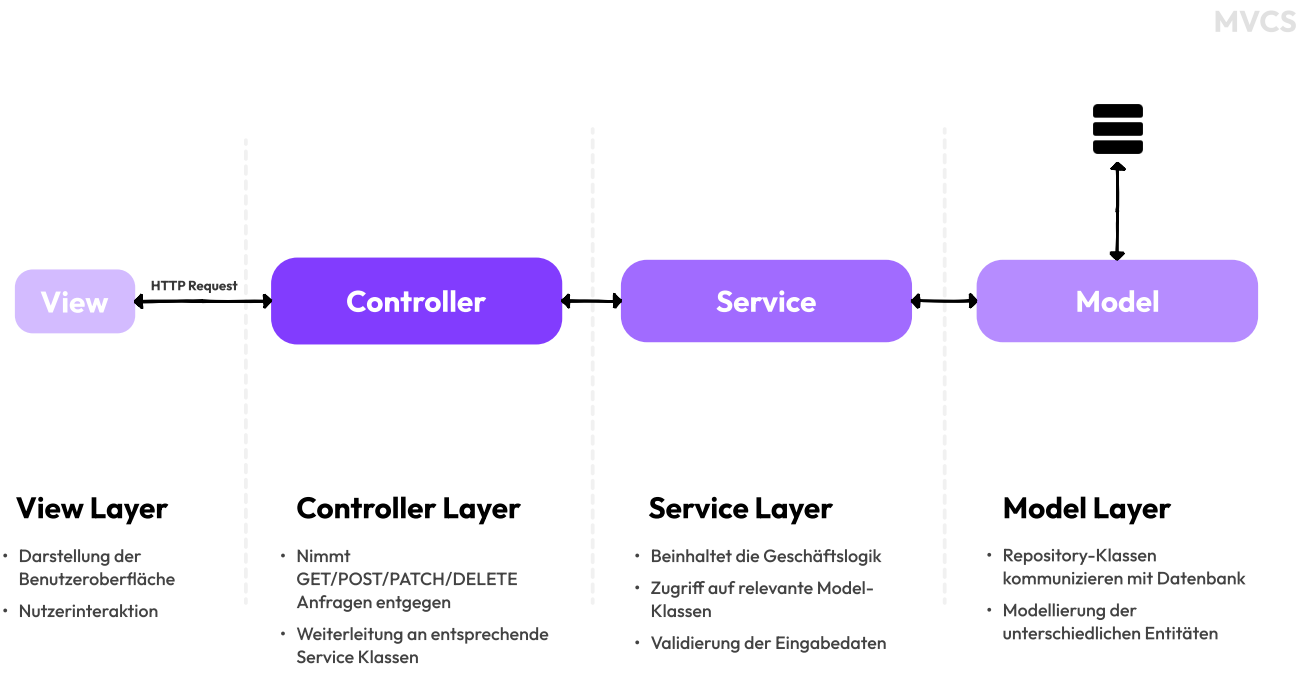
Das Architekturmuster Model-View-Controller-Service (MVCS) unterstützt nachhaltig die Wartbarkeit und Skalierbarkeit des Quellcodes über den erfolgreichen Abschluss des PSE-Projektes hinaus. Der Hauptvorteil der MVCS-Architektur in der Web-App-Entwicklung liegt in der klaren Abgrenzung der verschiedenen logischen Komponenten. Diese Struktur ermöglicht es dem Entwicklungsteam, Datenmodellierung, Benutzeroberfläche und API-Schnittstellen parallel zu entwickeln und zu testen.
Die deutliche Trennung der Komponenten erlaubt es den Teammitgliedern, sich auf ihre spezifischen Stärken zu fokussieren, was wiederum die Fehleranfälligkeit minimiert. Zudem erlaubt die Isolierung der Komponenten eine vereinfachte und effektivere Qualitätssicherung durch isolierte Tests.
Ein zusätzlicher Nutzen in der MVCS-Architektur sind die Service-Klassen, die in der traditionellen MVC-Struktur nicht vorhanden sind. Diese Service-Klassen schaffen eine zusätzliche Schicht zwischen den Modell- und Controller-Klassen und trennen somit die API-Schnittstellen von der Geschäftslogik.

Dies fördert die Wiederverwendung von Code, da verschiedene Controller dieselben Service-Klassen nutzen können, und verhindert somit Code-Duplikation.

2.1.2 Realisierung der MVCS-Architektur

In unserem Projekt haben wir die MVCS-Architektur konsequent umgesetzt. Die View-Komponente wurde mittels React realisiert, was eine unabhängige Entwicklung und Implementierung der Benutzeroberfläche vom Backend ermöglicht. Sie interagiert über die Controller-Klassen mit dem Backend, welche als reine Schnittstellen fungieren und bewusst frei von Geschäftslogik gehalten werden, um die Klarheit und Einfachheit der Schnittstelle zu gewährleisten. Die Geschäftslogik selbst wird durch die Service-Klassen abgebildet, was eine Wiederverwendung von Code durch mehrere Controller ermöglicht. Ein Beispiel hierfür ist der Document-Controller, der auf den File-Service zugreift, um Signaturen abzurufen.
Die Modellierung der Datenstrukturen erfolgt in den Model-Klassen, wie etwa den Timesheet- und User-Klassen. Zudem werden Repository-Klassen eingesetzt, die die persistente Speicherung der Daten in der Datenbank verwalten.
Der Entwurf des Projekts legte größten Wert auf Erweiterbarkeit und Wartbarkeit. Die klare Trennung der Verantwortlichkeiten innerhalb der Klassenstruktur erleichtert das Verständnis des Systems für Außenstehende und gewährleistet, dass das Produkt auch nach Projektabschluss effizient weiterentwickelt und gewartet werden kann.

2.2 Schematische Darstellung der MVCS-Architektur



Die obige Grafik illustriert die Architektur des Model-View-Controller-Service (MVCS) Systems, das in unserem Projekt verwendet wird. Jede der vier Hauptkomponenten der Architektur ist spezifisch für bestimmte Aufgaben zuständig:

1. **View Layer:** Diese Schicht ist für die Darstellung der Benutzeroberfläche und die Interaktion mit dem Nutzer verantwortlich. Sie empfängt Benutzereingaben und sendet HTTP-Anfragen an den Controller.

2. **Controller Layer:** Der Controller agiert als Vermittler zwischen der View und den weiteren Schichten des Systems. Er nimmt HTTP-Anfragen (GET, POST, PATCH, DELETE) entgegen und leitet diese an die entsprechenden Service-Klassen weiter, ohne dabei selbst Geschäftslogik zu implementieren.

3. **Service Layer:** Diese Schicht beinhaltet die Geschäftslogik der Anwendung. Sie verarbeitet die von den Controllern weitergeleiteten An-

fragen, interagiert mit den Model-Klassen und führt Datenvalidierungen durch, um sicherzustellen, dass alle Eingaben korrekt sind.

4. **Model Layer:** In dieser Schicht werden die Datenstrukturen und Domänenklassen zur reinen Modellierung definiert. Zusätzlich sind hier Repository-Klassen angesiedelt, die den Zugriff auf die Datenbank regeln und die konsistente Speicherung sowie Abfrage der Daten gewährleisten.

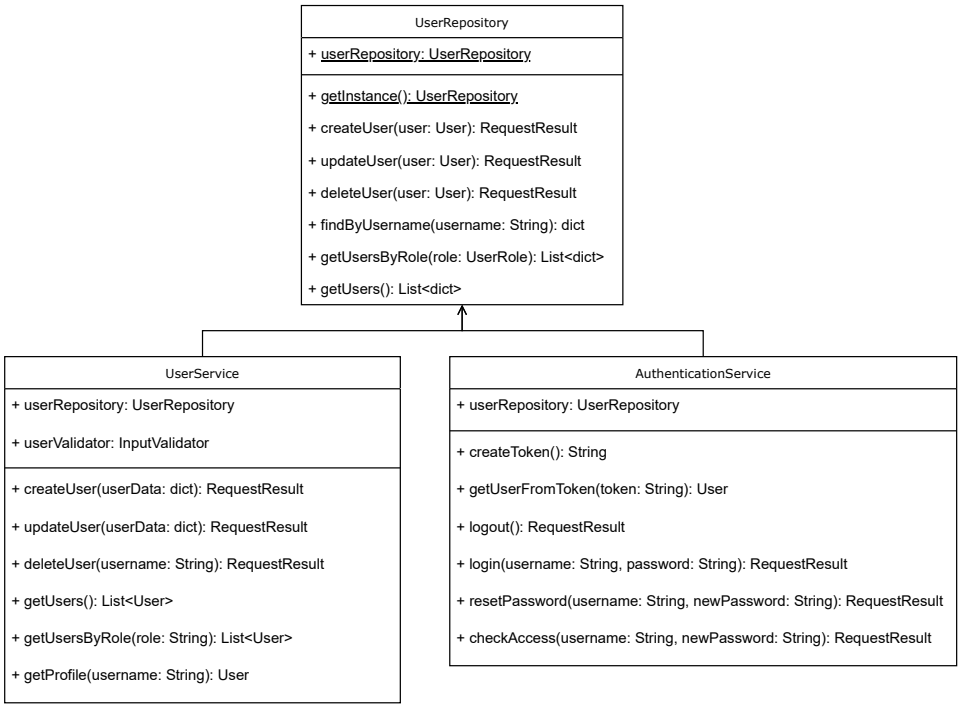
Diese klare Aufteilung der Verantwortlichkeiten in verschiedenen Schichten erleichtert die Wartung, Erweiterung und Skalierung des Systems, indem jede Schicht spezifische Aufgaben ohne Überschneidungen ausführt.

3 UML Diagramme

3.1 Klassendiagramme

Die folgenden UML-Klassendiagramme stellen kleinere Ausschnitte des gesamten UML-Diagramms dar. Diese Ausschnitte wurden aus Gründen der Übersichtlichkeit gewählt. Um die Lesbarkeit und Verständlichkeit zu gewährleisten, wurden einige Teile und Beziehungen weggelassen oder nur unvollständig dargestellt. Die Diagramme sollen dazu dienen, wichtige Aspekte unserer Systemarchitektur zu verdeutlichen, ohne die Komplexität des vollständigen Diagramms zu übernehmen.

3.1.1 User Repository



Im Rahmen unserer Softwarearchitektur spielt das **UserRepository** eine zentrale Rolle in der Verwaltung und Persistierung von Benutzerdaten. Es implementiert zwei wesentliche Entwurfsmuster: das *Singleton* und das *Repository Pattern*, welche entscheidend zur Effizienz und Klarheit unseres Systems beitragen.

Singleton-Muster: Das Singleton-Muster stellt sicher, dass von der `UserRepository`-Klasse eine und nur eine Instanz existiert. Dies ermöglicht eine konsistente Handhabung der Datenzugriffe über unsere Applikation hinweg, da alle Datenanfragen über dasselbe Objekt laufen und damit die Integrität und der Zustand der Benutzerdaten gewahrt bleiben.

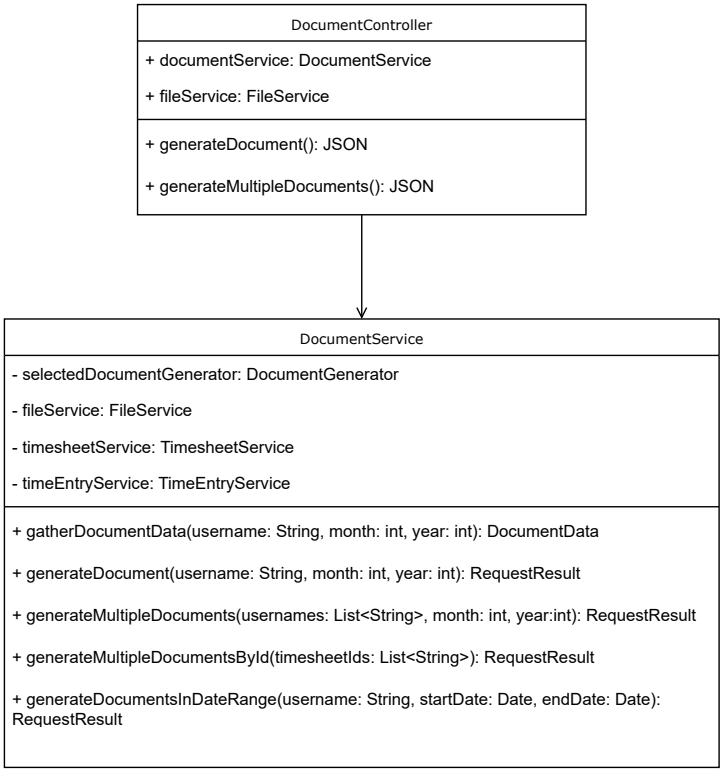
Repository-Muster: Das Repository-Muster wird genutzt, um eine klare Trennung zwischen der Geschäftslogik und den Datenzugriffsschichten zu schaffen. `UserRepository` abstrahiert die Logik für den Zugriff auf die Datenquelle, sodass die Service-Klassen (`UserService` und `AuthenticationService`)

nicht direkt mit der Datenbank kommunizieren müssen. Stattdessen verwenden sie das Repository, um Create, Read, Update und Delete (CRUD) Operationen durchzuführen. Dies verbessert die Wartbarkeit und Erweiterbarkeit des Codes erheblich.
Die `UserService` und `AuthenticationService` Klassen kommunizieren mit dem `UserRepository`, um diverse Benutzer-bezogene Funktionen auszuführen. Beispiele hierfür sind:

- Erstellen neuer Benutzerkonten
- Aktualisieren bestehender Benutzerdaten
- Authentifizierung und Autorisierung von Benutzern
- Verwaltung von Passwort-Rücksetzungen

Durch die Nutzung des `UserRepository` für diese Aufgaben, garantieren wir, dass alle Interaktionen mit den Benutzerdaten konsistent und sicher ablaufen.

3.1.2 Dokumentenverwaltung



Die Architektur unseres Systems zur Dokumentengenerierung umfasst zwei Hauptkomponenten: den **DocumentController** und den **DocumentService**. Diese Komponenten interagieren miteinander, um die Erstellung und Verwaltung von Dokumenten effizient zu handhaben.
Der **DocumentController** dient als primäre Schnittstelle für die Benutzerinteraktion. Er empfängt Anfragen, um einzelne oder mehrere Dokumente zu generieren. Die Hauptfunktionen umfassen:

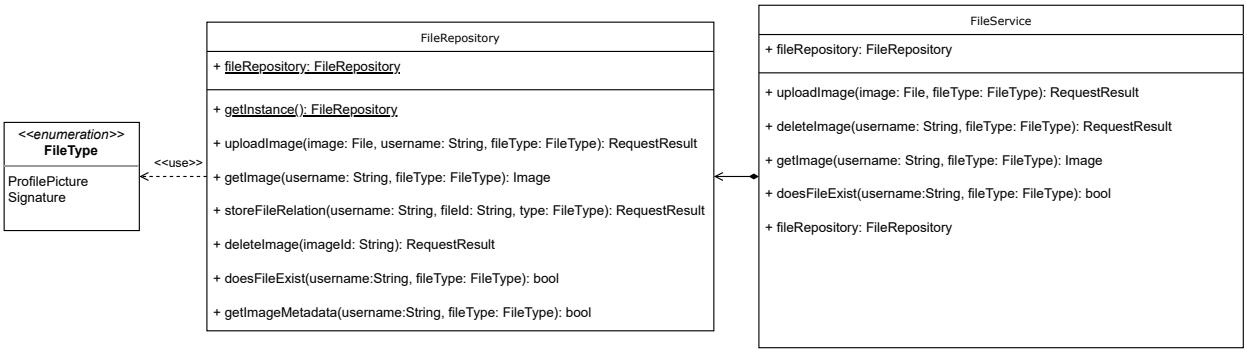
- `generateDocument()`: Generiert ein einzelnes Dokument basierend auf Benutzeranforderungen.
- `generateMultipleDocuments()`: Ermöglicht die Generierung mehrerer Dokumente simultan, was eine effiziente Massenverarbeitung von Dokumentenanforderungen unterstützt.

Der **DocumentService** verarbeitet die vom Controller empfangenen Anforderungen und führt die notwendige Logik zur Dokumentengenerierung

durch. Er integriert mehrere Dienste:

- **DocumentGenerator**: Wählt abhängig von der Anforderung den geeigneten Generator für die Dokumenterstellung aus.
- **FileService**: Verwaltet die Speicherung und Abrufung von Dateien, welche für die Dokumentengenerierung benötigt werden.
- **TimesheetService** und **TimeEntryService**: Diese Services liefern spezifische Daten, die zur Generierung der Arbeitszeitendokumentation benötigt werden.

3.1.3 Dateiverwaltung



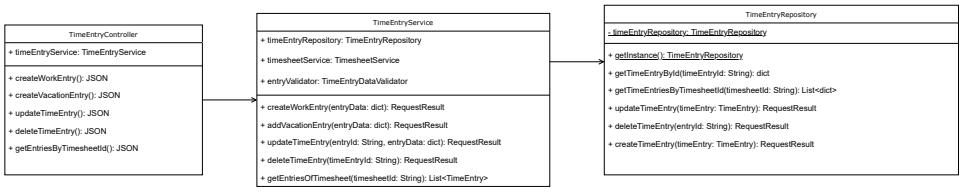
Innerhalb unserer Anwendung spielt die Verwaltung von Dateien wie Profilbildern und Signaturen durch den **FileService** und das **FileRepository** eine zentrale Rolle. Diese Komponenten sind entscheidend für das Speichern und Abrufen von Dateien, die in **MongoDB** mittels **GridFS** gesichert werden.
Das **FileRepository** bietet eine Abstraktionsschicht für den Zugriff auf die Dateispeicherung. Es ermöglicht:

- Hochladen von Bildern (`uploadImage`) und deren Abruf (`getImage`),
- Überprüfung der Existenz von Dateien (`doesFileExist`),
- Löschung von Dateien (`deleteImage`) und
- Speicherung von Beziehungen zwischen Benutzern und Dateien (`storeFileRelation`).

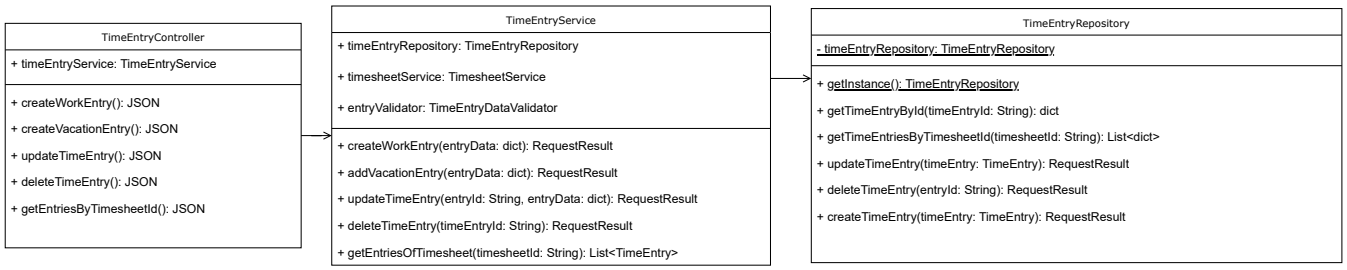
Der **FileService** agiert als Vermittler zwischen den Endbenutzern und dem Repository, wobei er Methoden zur Manipulation und Abfrage von Dateiinformationen bereitstellt. Er nutzt das Repository, um:

- Dateien basierend auf Benutzernamen und Dateityp zu verwalten,
- notwendige Metadaten für hochgeladene Dateien zu sammeln (`getImageMetadata`) und
- die Integrität und Sicherheit der Dateispeicherung zu gewährleisten.

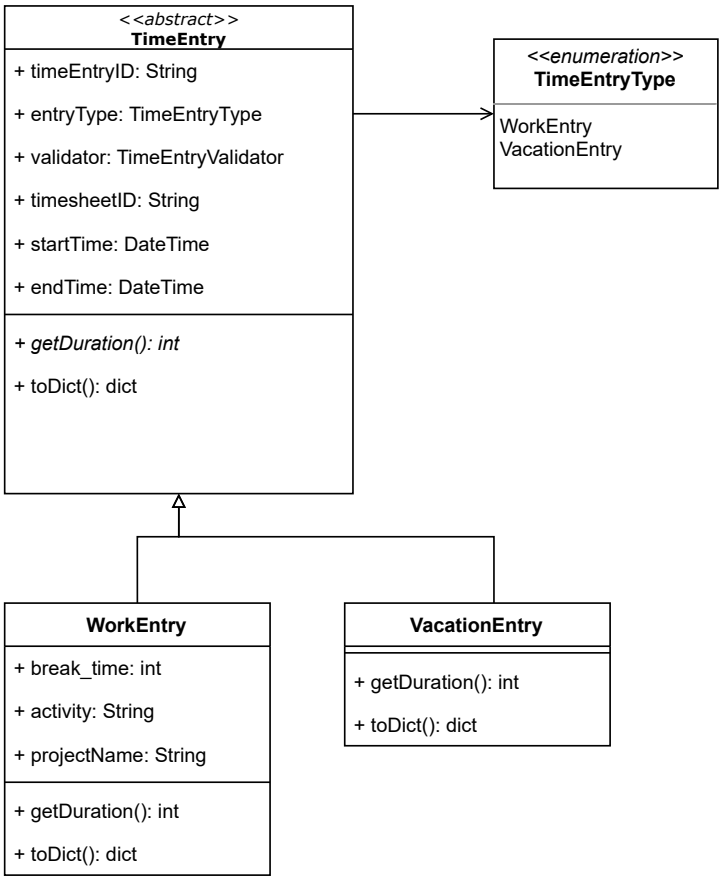
Die Verwendung des Enums **FileType** ermöglicht eine einfache Erweiterung der Anwendung um neue Dateitypen. Dies bietet eine flexible Lösung, um zukünftige Anforderungen ohne größere Änderungen an der Codebasis zu unterstützen.



3.1.4 Time-Entry Controller-Service-Architektur

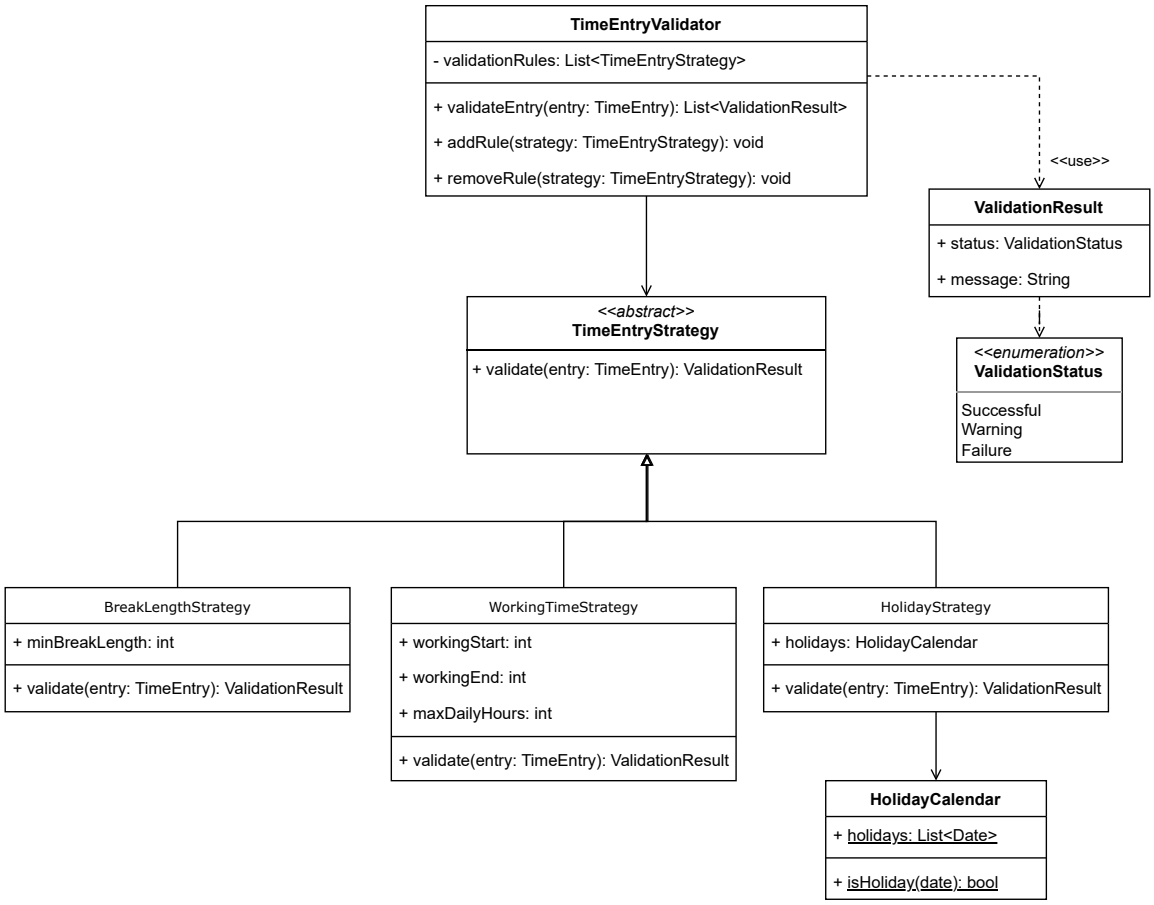


3.1.5 Modellierung der Zeiteinträge



Die Abbildung zeigt die Struktur der Domänenklassen für Zeiteinträge in unserem System. Jeder Zeiteintrag repräsentiert beispielsweise einen Arbeitstag oder einen Urlaubstag und wird durch verschiedene Klassen und Typen abgebildet.
Die abstrakte Klasse **TimeEntry** dient als Basis für alle spezifischen Zeiteintragstypen. Dieses Design ermöglicht es, gemeinsame Funktionalitäten und Attribute zentral zu verwalten, wodurch der Code besser wartbar und erweiterbar wird. Die Spezialisierungen **WorkEntry** und **VacationEntry** erben von `TimeEntry` und fügen spezifische Eigenschaften und Verhaltensweisen hinzu.

3.1.6 Validierung von Zeiteinträgen



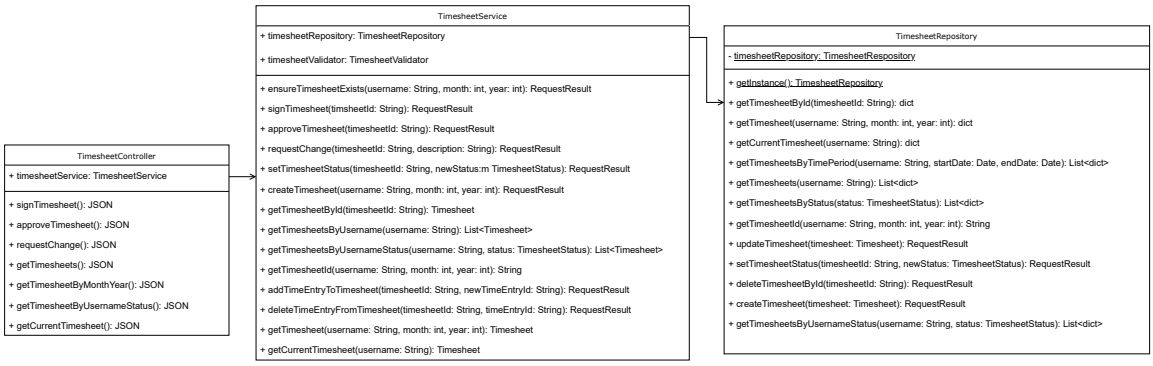
Die Abbildung zeigt den Einsatz des *Strategie-Patterns* zur Validierung von Zeiteinträgen in unserem System. Dieses Muster ermöglicht es, verschiedene Validierungsstrategien für Zeiteinträge zur Laufzeit hinzuzufügen und zu entfernen, wodurch das System flexibel und erweiterbar bleibt.
Der **TimeEntryValidator** ist die zentrale Komponente dieses Musters. Er hält eine Liste von Validierungsstrategien (`validationRules`) und bietet Methoden zur Validierung von Zeiteinträgen (`validateEntry`), sowie zum Hinzufügen und Entfernen von Validierungsregeln (`addRule` und `removeRule`).
Die abstrakte Klasse **TimeEntryStrategy** definiert das Interface für alle spezifischen Validierungsstrategien. Jede Strategie implementiert die Methode `validate`, die einen Zeiteintrag entgegennimmt und ein `ValidationResult`

zurückgibt. Dieses Resultat enthält den Status der Validierung (`ValidationStatus`) und eine Nachricht, die das Ergebnis beschreibt.
Verschiedene konkrete Strategien erben von `TimeEntryStrategy` und implementieren spezifische Validierungslogiken:

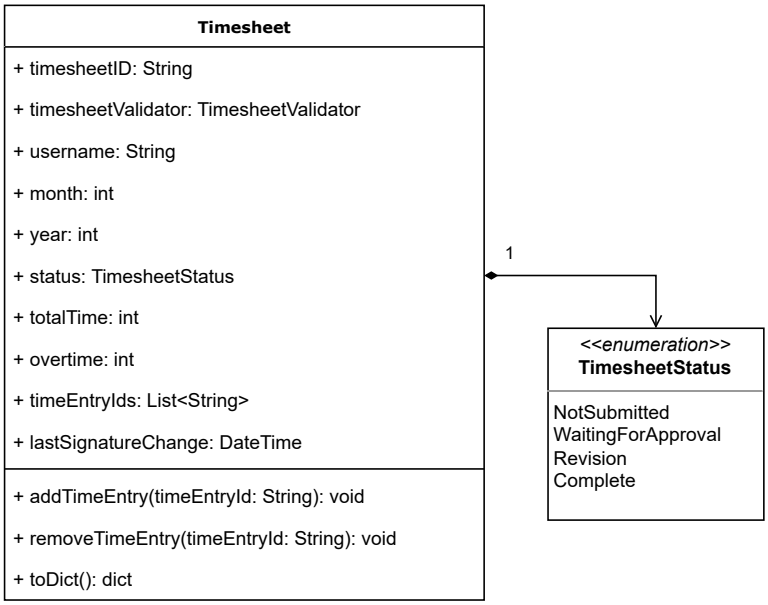
- **BreakLengthStrategy**: Validiert, ob die Pausenzeit eines Zeiteintrags die Mindestanforderungen erfüllt.
- **WorkingTimeStrategy**: Überprüft, ob die Arbeitszeit innerhalb der zulässigen Arbeitsstunden liegt und ob die tägliche Höchstarbeitszeit nicht überschritten wird.
- **HolidayStrategy**: Prüft, ob ein Zeiteintrag auf einen Feiertag fällt, basierend auf dem `HolidayCalendar`.

Der **ValidationResult** gibt den Status der Validierung (`ValidationStatus`), der einer der folgenden Werte sein kann: `Successful`, `Warning`, oder `Failure`. Dies ermöglicht eine differenzierte Rückmeldung über die Gültigkeit des Zeiteintrags.

3.1.7 Timesheet Controller-Service-Architektur



3.1.8 Timesheet Domäne

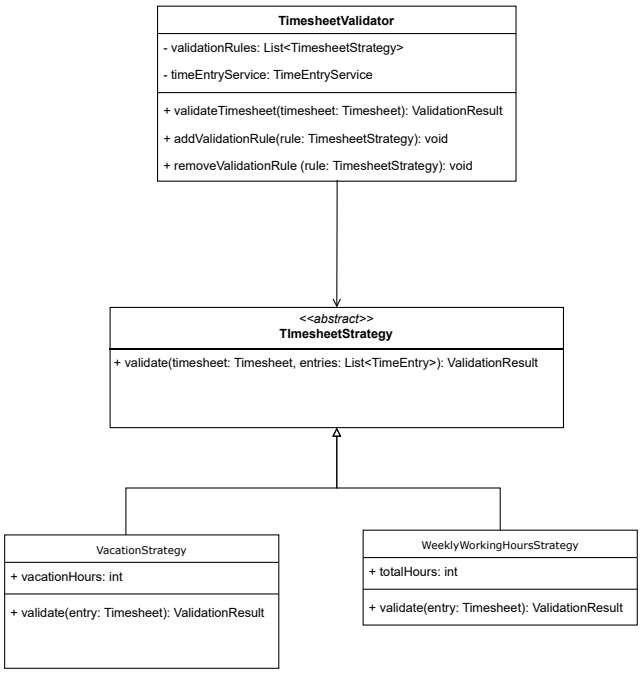


Die Abbildung stellt die Klasse **Timesheet** dar, die einen Arbeitsmonat innerhalb unseres Systems modelliert. Ein **Timesheet** ist verantwortlich für die Verwaltung der Zeiteinträge (Time Entries), die die Arbeitsleistung eines Nutzers über einen bestimmten Monat dokumentieren. Der **TimesheetStatus** spielt eine zentrale Rolle in der Verwaltung und Kontrolle des Fortschritts von Timesheets:

- `NotSubmitted`: Das Timesheet wurde noch nicht zur Überprüfung oder Genehmigung eingereicht.
- `WaitingForApproval`: Das Timesheet wartet auf die Überprüfung und Genehmigung durch einen Vorgesetzten.
- `Revision`: Das Timesheet wurde zurückgegeben und bedarf einer Überarbeitung durch den HiWi.
- `Complete`: Das Timesheet wurde überprüft und genehmigt und ist nun abgeschlossen.

Diese Status ermöglichen es, den Lebenszyklus eines Timesheets präzise zu verwalten und stellen sicher, dass alle notwendigen Schritte zur Validierung und Genehmigung durchgeführt werden.

3.1.9 Timesheet Validierung



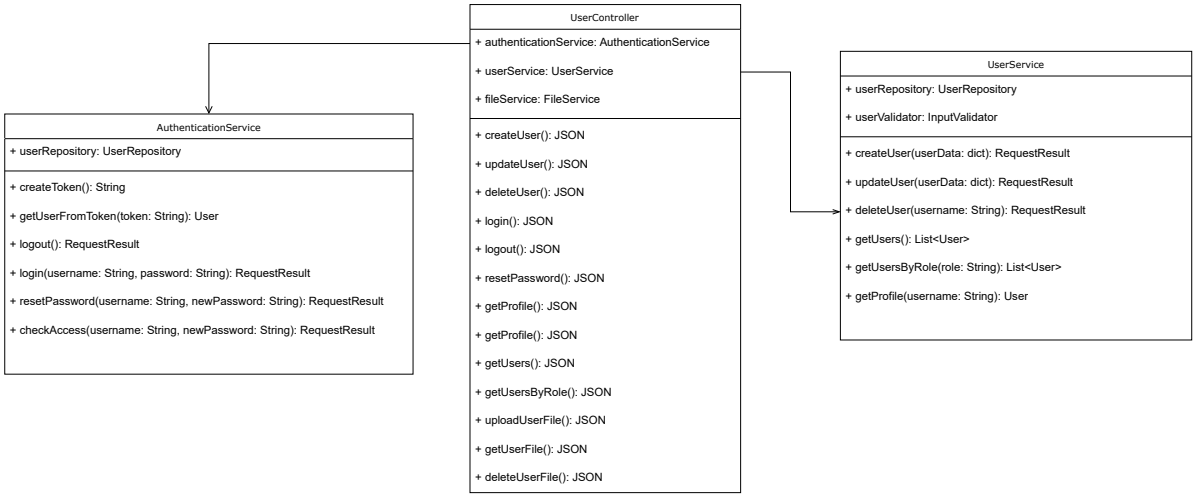
Der **TimesheetValidator** verwendet das Strategie-Pattern, um eine flexible und erweiterbare Architektur zur Validierung von Arbeitszeiten in

Timesheets zu ermöglichen. Dieses Muster ist entscheidend für die Anpassungsfähigkeit des Systems an unterschiedliche Validierungsanforderungen.
Die Architektur des **TimesheetValidator** besteht aus einer zentralen Komponente, die eine Liste von Validierungsstrategien verwaltet. Jede dieser Strategien, repräsentiert durch die abstrakte Klasse **TimesheetStrategy**, definiert eine spezifische Validierungslogik, die auf Timesheets angewendet werden kann.
Das Strategie-Pattern ermöglicht es, unterschiedliche Validierungsregeln dynamisch hinzuzufügen oder zu entfernen. Diese Flexibilität ist besonders vorteilhaft in einem Umfeld, wo Arbeitszeitregelungen häufigen Änderungen unterliegen können oder stark von lokalen Gesetzgebungen abhängen.
Beispiele für spezifische Strategien im Einsatz sind:

- **VacationStrategy**: Validiert Urlaubszeiten gegenüber den festgelegten Richtlinien des Instituts.
- **WeeklyWorkingHoursStrategy**: Überprüft die Einhaltung der wöchentlichen Arbeitsstundenvorgaben.

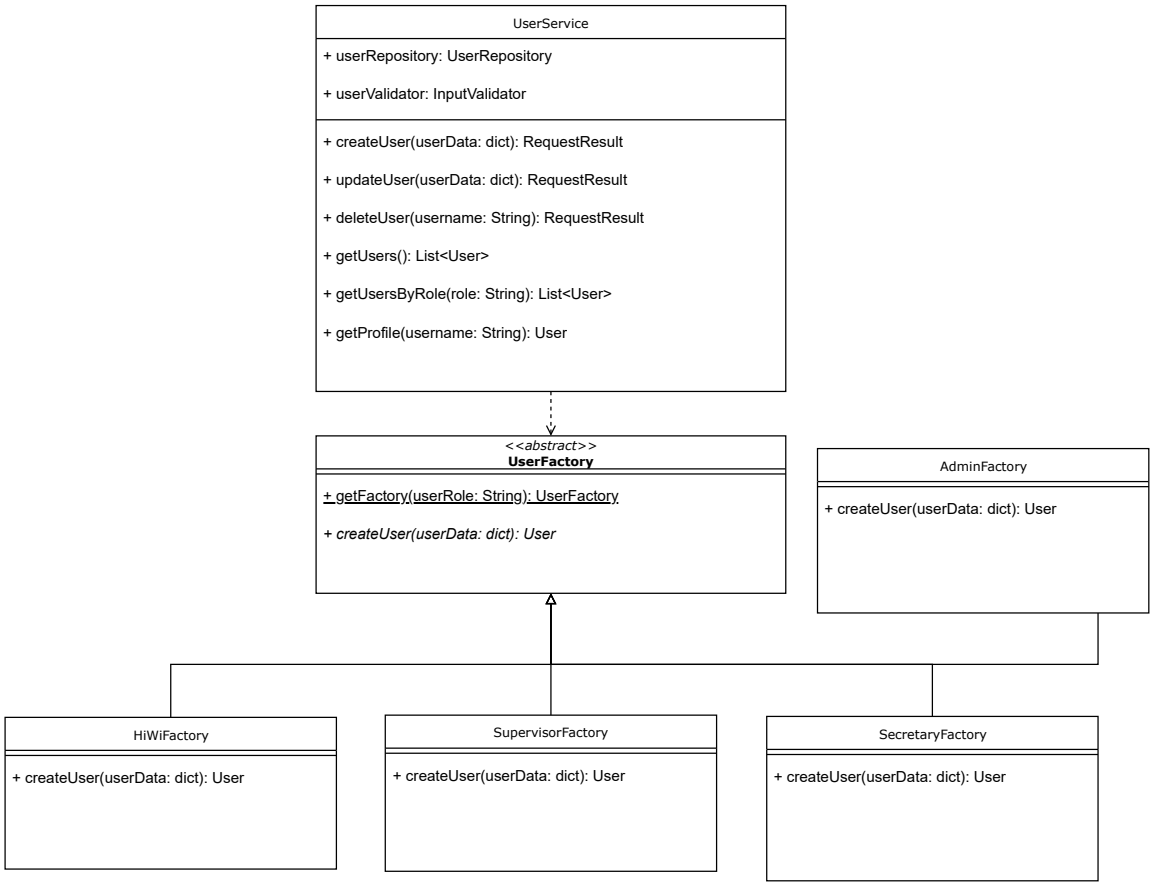
Die Verwendung des Strategie-Patterns ermöglicht es, den Validierungsprozess modular und wartbar zu gestalten. Dies trägt dazu bei, dass das System leicht an neue oder sich ändernde Anforderungen angepasst werden kann, ohne die bestehende Codebasis umfangreich zu modifizieren.

3.1.10 User Controller-Service-Architektur



Das UML-Diagramm verdeutlicht die strukturierte Architektur unserer Benutzerverwaltung, die in drei Hauptkomponenten unterteilt ist: **AuthenticationService**, **UserController**, und **UserService**. Diese Komponenten interagieren miteinander, um verschiedene Aspekte der Benutzerverwaltung wie Authentifizierung, Benutzerdatenmanagement und Dateiverwaltung zu handhaben.
Der **UserController** dient als Schnittstelle zwischen der Benutzeroberfläche und den Diensten, die die Benutzerdaten verwalten. Er leitet Anfragen an die entsprechenden Services weiter und schickt die Ergebnisse am Ende wieder als JSON an das Frontend.
Der **UserService** implementiert die Geschäftslogik zur Verwaltung der Benutzerdaten. Dieser Service interagiert eng mit dem **UserRepository**, um Benutzerdaten zu speichern und abzurufen.

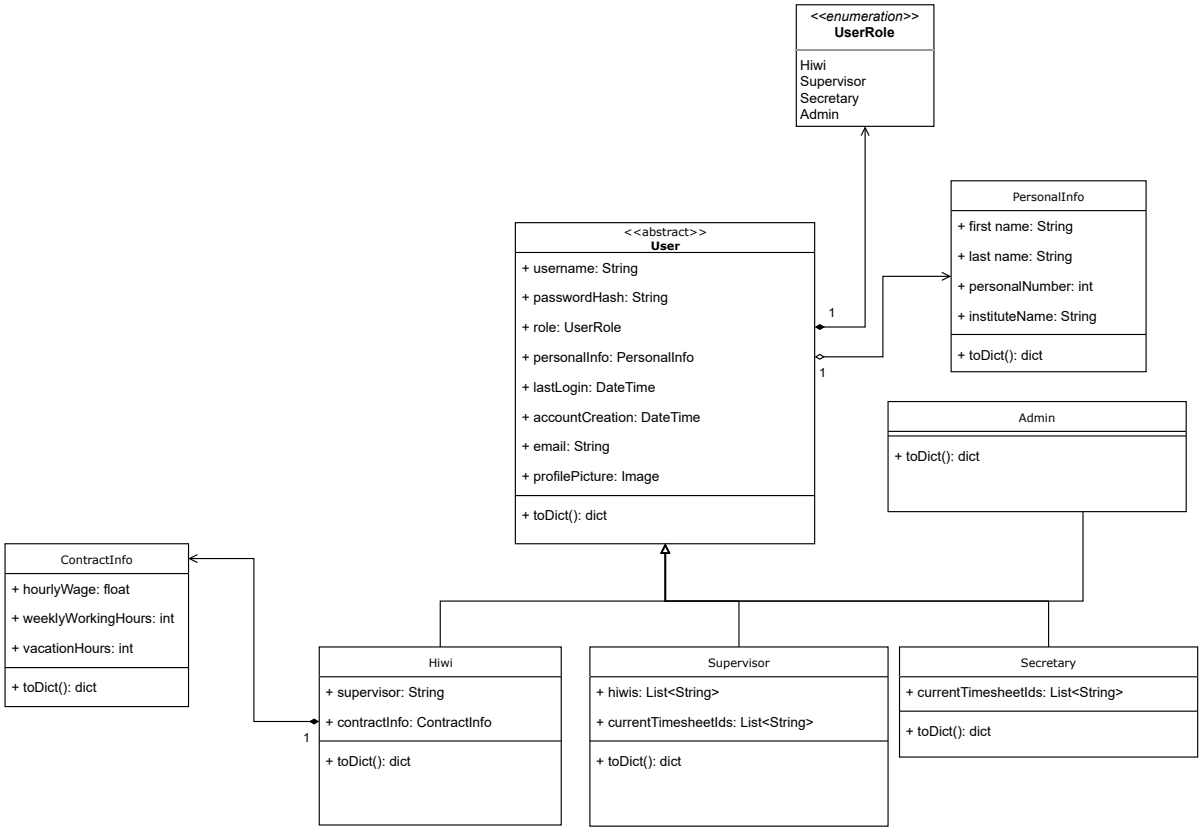
3.1.11 User Factory



Das Diagramm zeigt die Anwendung des **Factory-Method-Musters** in der Architektur der **UserService**-Klasse, welche spezifische **User**-Objekte basierend auf der Benutzerrolle erstellt. Dieses Muster ermöglicht es, die Erstellung von Benutzerobjekten zu vereinfachen und dabei die Flexibilität und Erweiterbarkeit des Systems zu erhöhen.
Die **UserService**-Klasse ist zentral für das Benutzermanagement verantwortlich und nutzt dabei die abstrakte **UserFactory**, um Benutzerobjekte zu instanziieren. Diese Factory abstrahiert die Erstellungsdetails und ermöglicht die dynamische Integration verschiedener Benutzertypen.
Die **UserFactory**-Klasse dient als abstrakte Basis für verschiedene Factories, die spezifische Benutzerobjekte erstellen. Jede dieser Factories implementiert die Methode `createUser()`, die spezifisch für den jeweiligen Be-

nutzertyp angepasst ist. Dies stellt sicher, dass die Erstellung von Benutzern je nach Rolle spezifischen Anforderungen folgt.
Die Entscheidung für das **Factory Method-Muster** ermöglicht eine klare Trennung zwischen der Erstellung von Objekten und ihrer Nutzung. Dies fördert die Prinzipien des `Open/Closed-Prinzips`, indem das System für Erweiterungen offen, aber für Modifikationen geschlossen bleibt. Neue Benutzertypen können hinzugefügt werden, ohne die bestehende Codebasis zu ändern.

3.1.12 User Domänenklassen



Das vorliegende UML-Diagramm zeigt die Struktur und Beziehungen innerhalb unseres User-Domänenmodells, das verschiedene Benutzertypen in einer hierarchischen und modularen Art und Weise abbildet.
Im Zentrum des Modells steht die abstrakte Klasse **User**, die allgemeine Attribute und Methoden für Benutzer im System bereitstellt. Diese Klasse

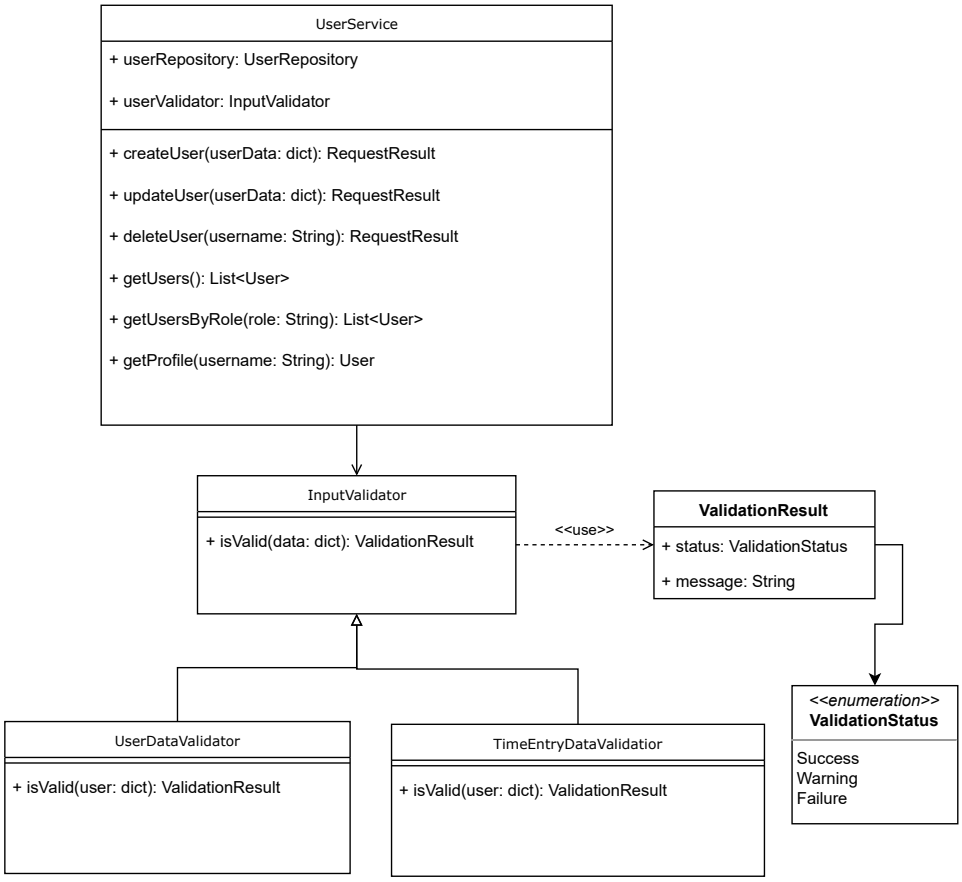
dient als Basisklasse für spezialisierte Benutzertypen, welche durch Vererbung spezifische Eigenschaften und Verhaltensweisen hinzufügen.

- **Hiwi**: Repräsentiert einen studentischen Mitarbeiter, der spezielle Attribute wie Betreuer und vertragliche Informationen enthält.
- **Supervisor**: Beschreibt einen Betreuer, der eine Liste an Hiwis betreut.
- **Secretary**: Verwaltet spezifische administrative Aufgaben und hat Zugriff auf relevante Zeitblätter.
- **Admin**: Stellt einen Administrator dar, der über erweiterte Rechte zur Verwaltung von Systemeinstellungen verfügt.

Die **UserRole** Enumeration definiert die verschiedenen Rollen, die ein *User* annehmen kann Hiwi, Supervisor, Secretary und Admin. Dies erleichtert die Rollenverwaltung und -überprüfung innerhalb des Systems.

- **PersonalInfo**: Hält persönliche Informationen wie Namen und Kontaktinformationen, die allen Benutzertypen gemein sind.
- **ContractInfo**: Enthält Vertragsdetails spezifisch für Mitarbeiter, die Verträge haben, wie Hiwis.

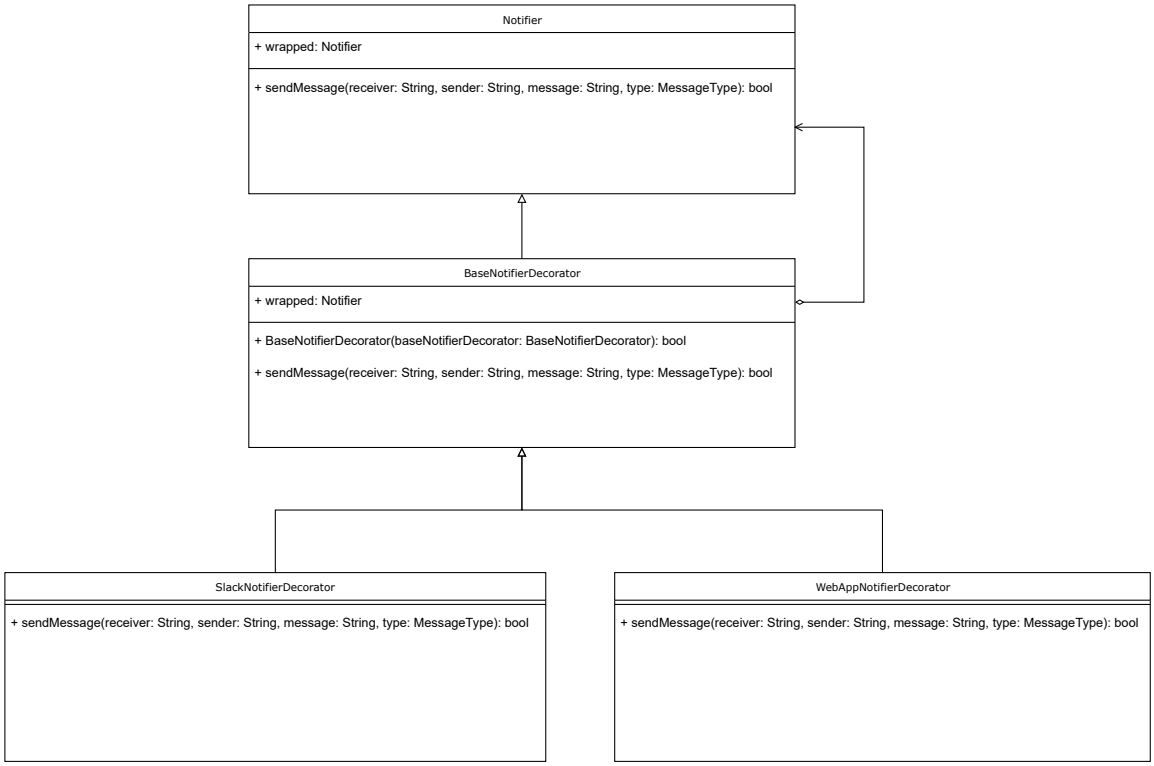
3.1.13 Eingabedaten Validierung (User)



Das UML-Diagramm verdeutlicht die Struktur und das Design der Validierungskomponenten innerhalb der `UserService`-Klasse. Diese Komponenten sind entscheidend für die Überprüfung der Integrität und Gültigkeit der Eingabedaten, die von den Controllern als JSON-Strukturen erhalten werden, bevor diese Daten zur Erstellung oder Aktualisierung von Domänenobjekten verwendet werden.

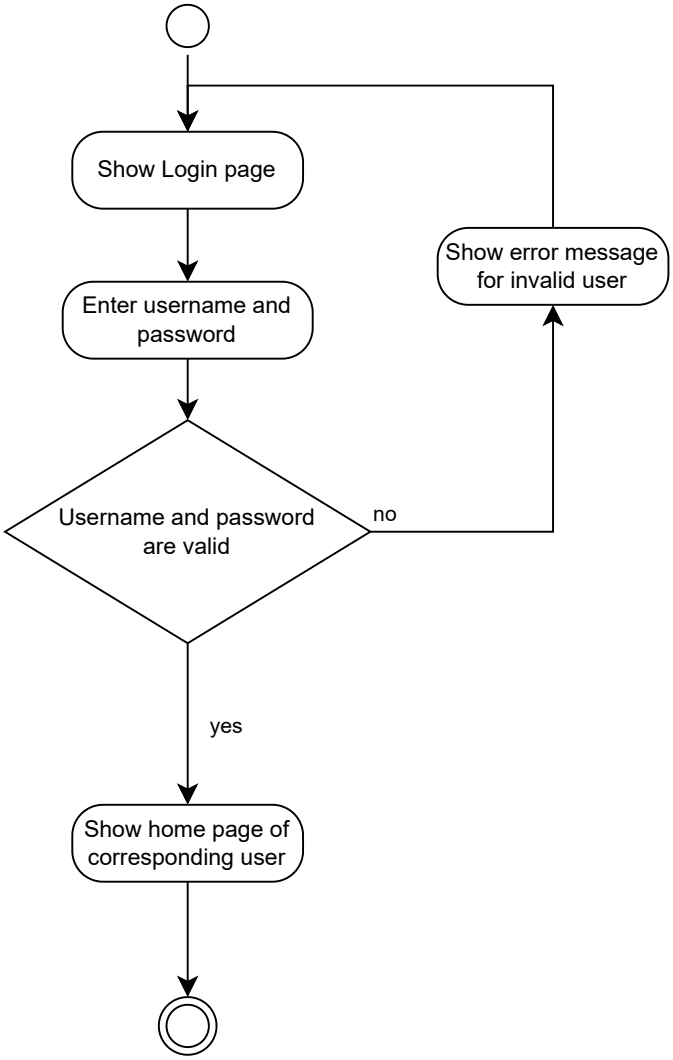
- **InputValidator**: Eine zentrale Validierungskomponente, die als Schnittstelle für spezifischere Validatoren dient und generelle Methoden zur Datenprüfung bereitstellt.

3.1.14 Benachrichtigungen (Optional)

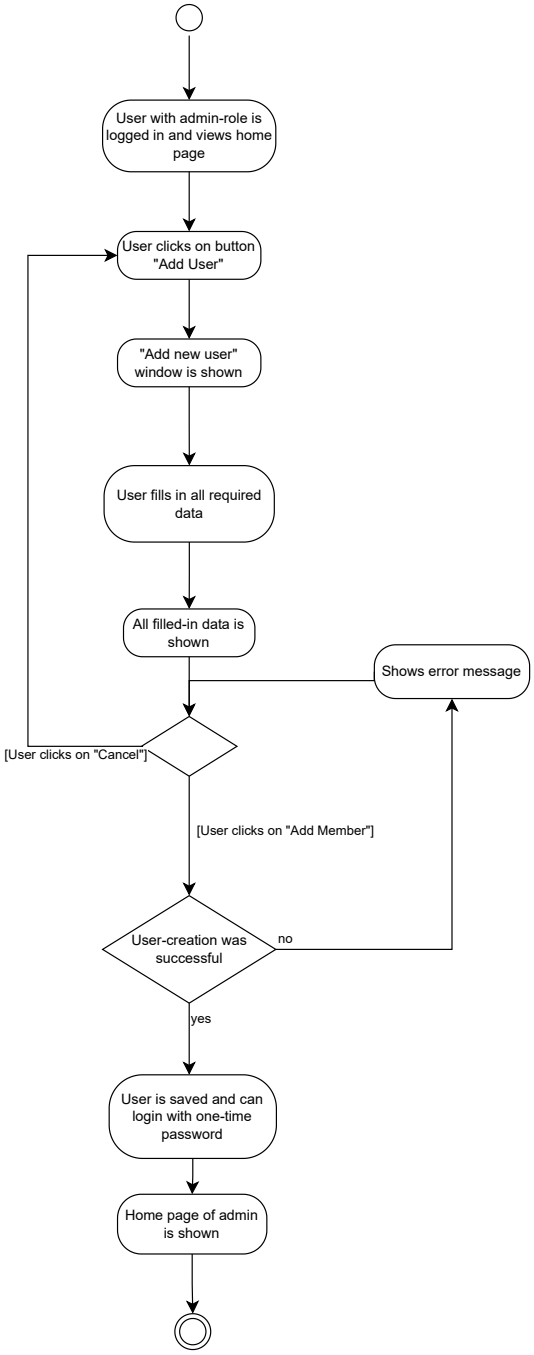


3.2 Aktivitätsdiagramme

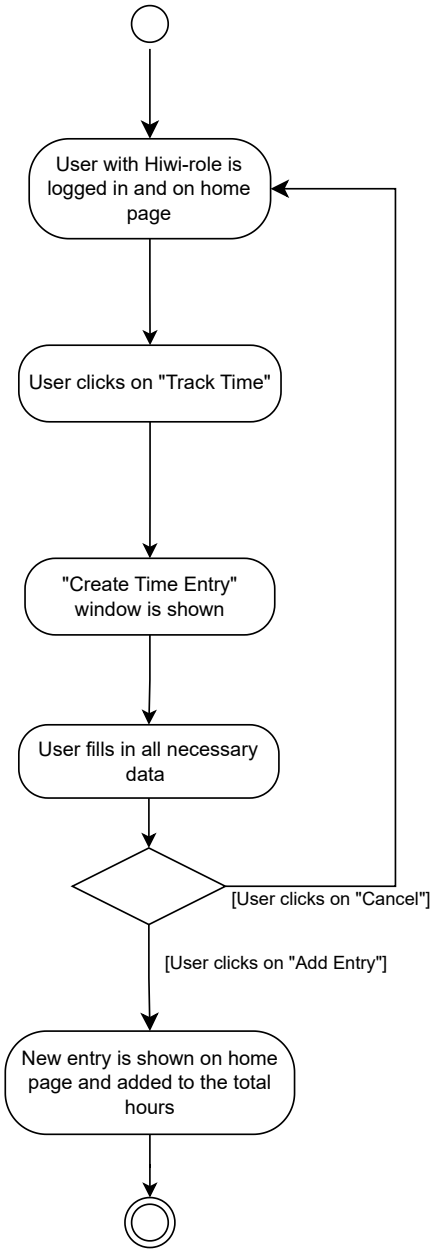
3.2.1 Login



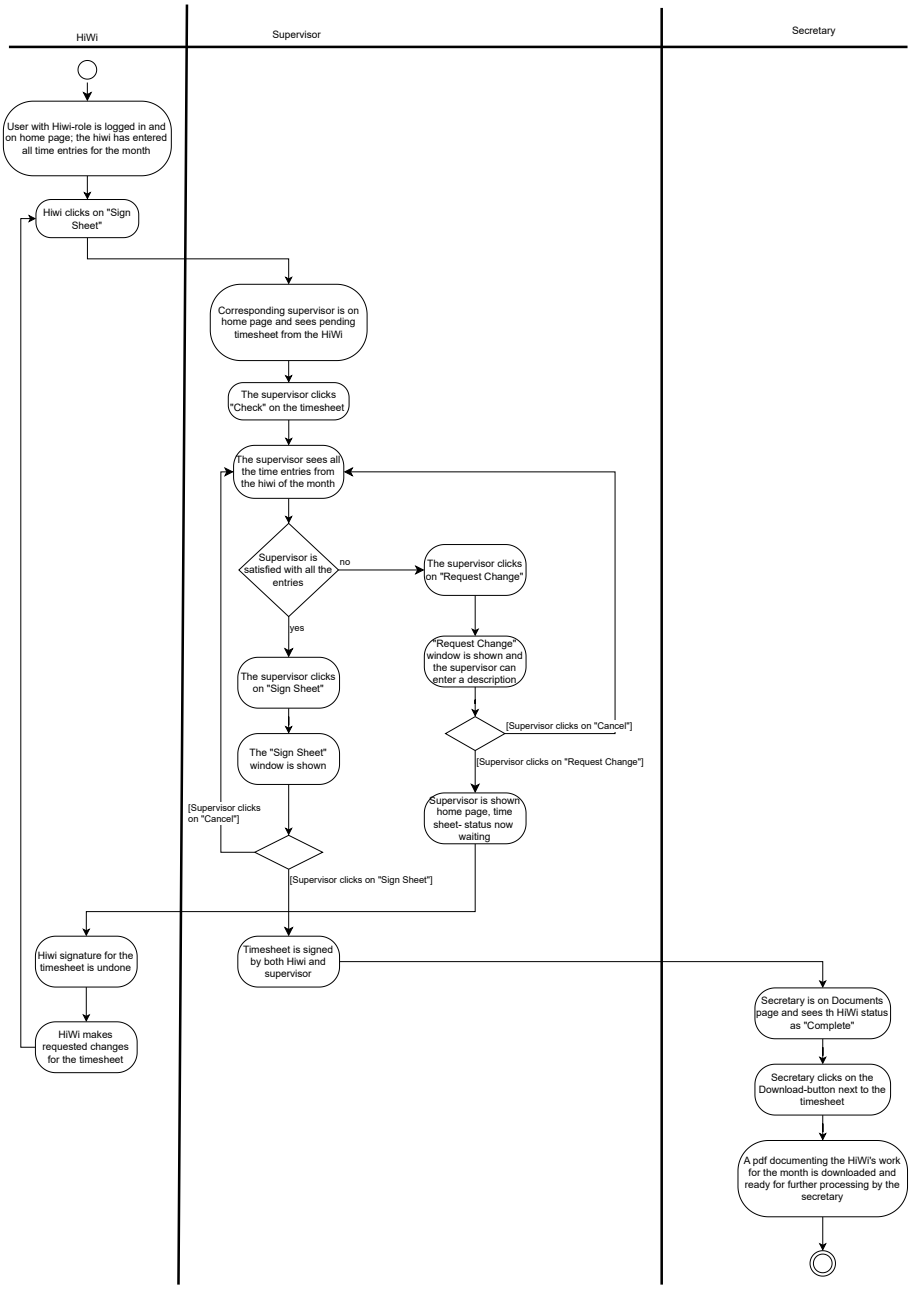
3.2.2 Neuen Nutzer hinzufügen



3.2.3 Zeiteintrag hinzufügen



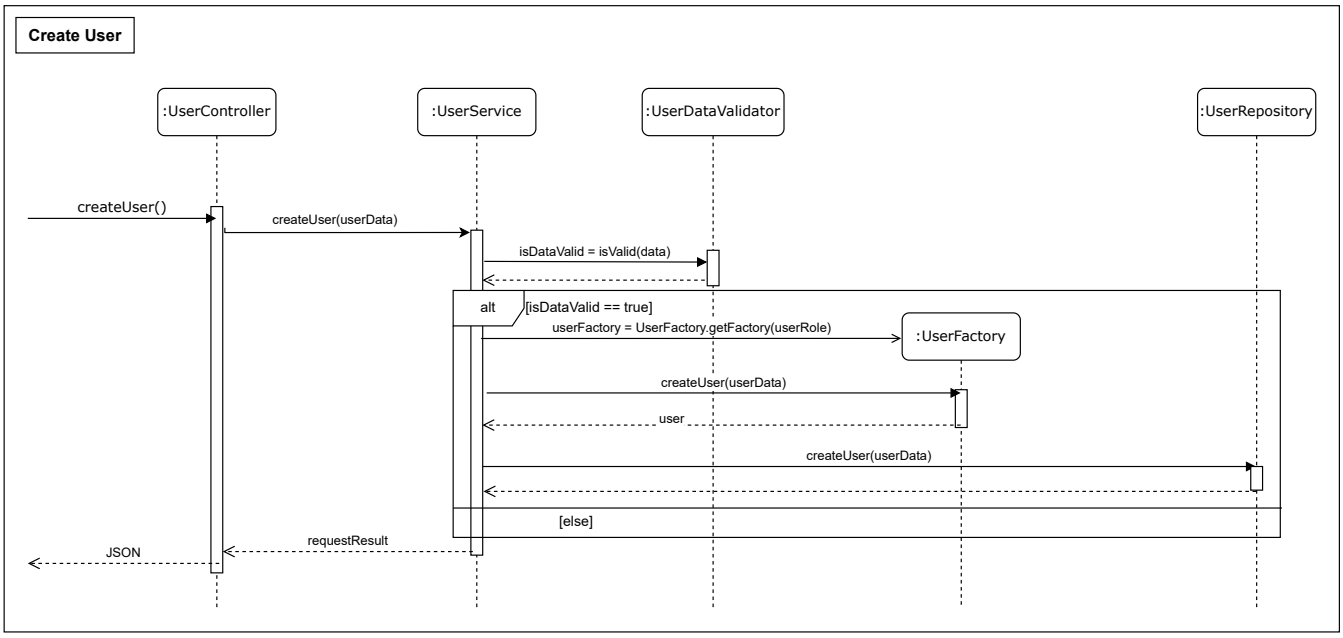
3.2.4 Timesheet signieren



3.3 Sequenzdiagramme

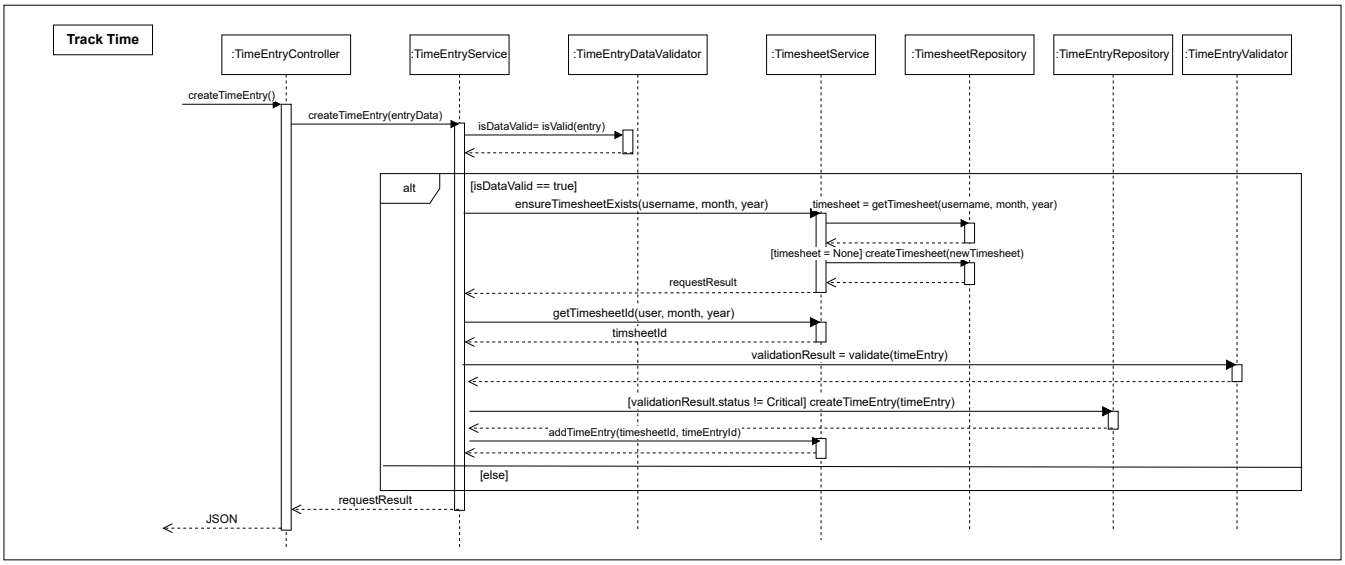
3.3.1 Neuen Nutzer erstellen

Es wird vorausgesetzt, dass der Nutzer über Administratorrechte verfügt und daher die Methode `createUser()` aufrufen kann. Das Ergebnis dieses Aufrufs wird intern verarbeitet und bei Bedarf zwischen verschiedenen Methoden weitergereicht. Der `UserController` initiiert diesen Prozess, indem er seine zugehörige Serviceklasse aufruft, welche die Eignung der eingegebenen Daten für die Erstellung eines neuen Nutzers überprüft. Sollten die Daten unzureichend sein, wird dies durch das `requestResult` an den Controller zurückgemeldet.
Wenn die Daten hingegen als valide erachtet werden, wird eine entsprechende `UserFactory` generiert, deren Auswahl von der Rolle des neuen Nutzers abhängt. Diese Factory ist dann verantwortlich für die Erstellung des Nutzerprofils, das anschließend durch das `UserRepository` in der Datenbank gesichert wird.



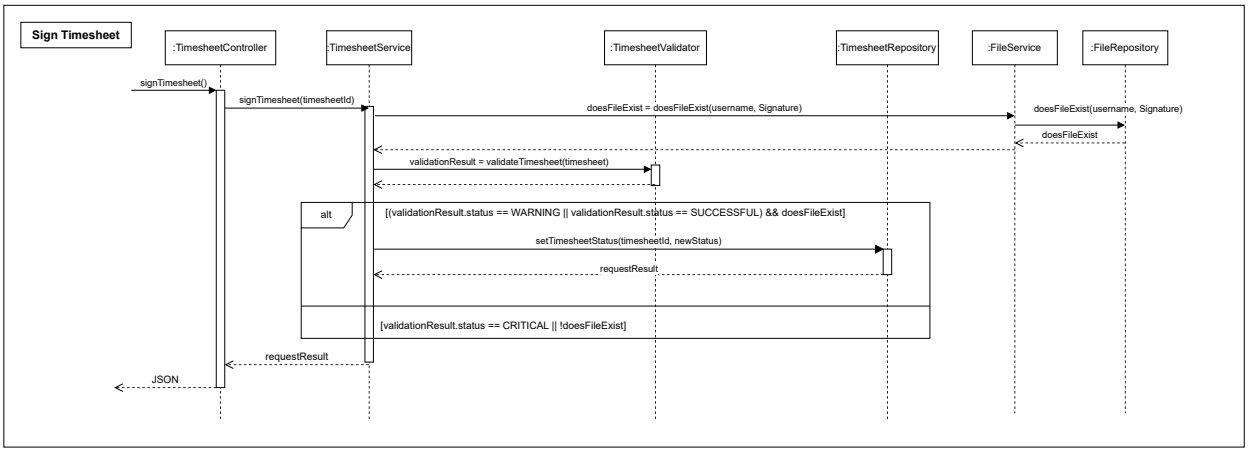
3.3.2 Neuen Zeiteintrag hinzufügen

Es wird angenommen, dass der Nutzer die Rolle eines HiWi hat.
Der `TimeEntryController` leitet den Prozess ein, indem er seine zugehörige Serviceklasse aufruft, um die Validität der eingegebenen Daten zu überprüfen. Falls die Daten ungültig sind, wird dies durch ein `requestResult` an den Controller zurückgemeldet.
Wenn die Daten valide sind, sorgt der `TimesheetService` dafür, dass ein entsprechendes Timesheet für den Nutzer und das Datum des eingetragenen Time Entries vorhanden ist. Dies geschieht durch eine Anfrage an das `TimesheetRepository`. Sollte noch kein Timesheet vorhanden sein, wird eines erstellt und in der Datenbank gespeichert.
Anschließend fragt der TimesheetService die ID des passenden Timesheets ab. Danach erfolgt eine weitere Validierung des neu erstellten TimeEntry-Objekts durch den `TimeEntryValidator`. Diese Validierung ist umfassender als die initiale Überprüfung der Eingabedaten, da sie das Time Entry anhand einer Reihe von Regeln inhaltlich prüft.
Wenn die Ergebnisse der Validierung akzeptabel sind, wird das Time Entry durch das `TimeEntryRepository` in der Datenbank gespeichert und anschließend vom TimesheetService dem zugehörigen Timesheet hinzugefügt.



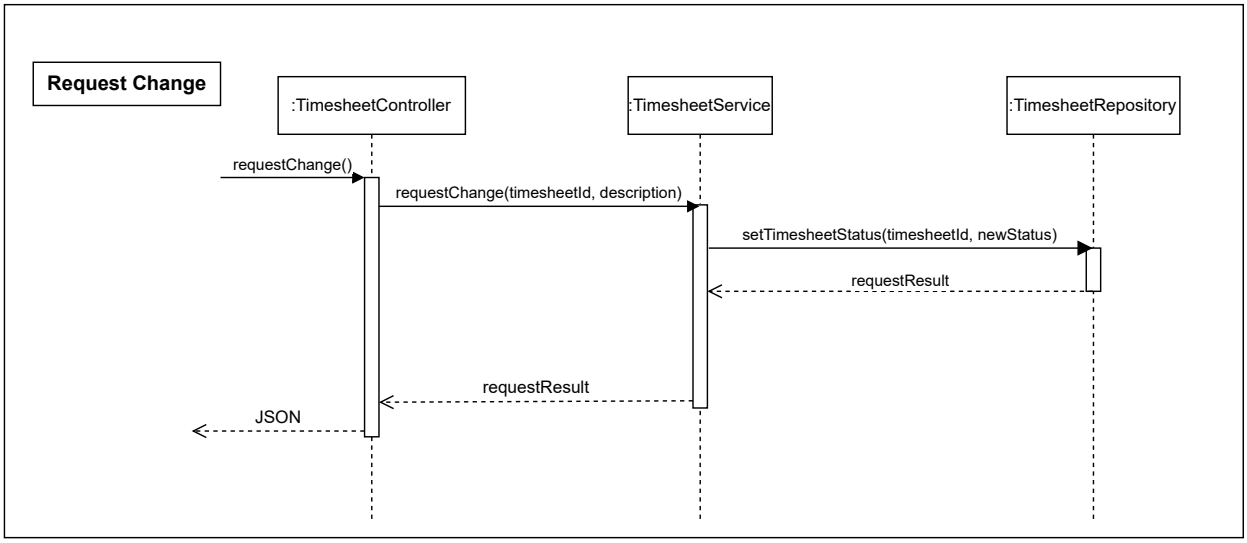
3.3.3 Timesheet signieren

Es wird davon ausgegangen, dass der Benutzer entweder ein wissenschaftlicher Mitarbeiter (HiWi) oder ein Betreuer ist. Im Falle eines Betreuers setzen wir voraus, dass das Timesheet bereits vom HiWi signiert wurde.
Der `TimesheetController` initiiert den Überprüfungsprozess, indem er die zugehörige Serviceklasse aktiviert. Diese führt zuerst eine Kontrolle durch, um sicherzustellen, dass der Benutzer eine Signaturdatei hochgeladen hat. Diese Überprüfung erfolgt mittels einer Anfrage an den `FileService`, der daraufhin das `FileRepository` kontaktiert, um die Verfügbarkeit der Signatur zu verifizieren.
Sollte die Signatur fehlen, wird das Timesheet als nicht signiert betrachtet. Diese Information und der entsprechende Fehler werden über das Objekt `requestResult` an den Controller zurückgemeldet.
Falls eine Signatur vorhanden ist, schreitet der Prozess voran zur Validierung des Timesheets. Diese Validierung prüft das Dokument auf kritische Probleme. Wenn das Timesheet fehlerfrei und korrekt signiert ist, aktualisiert das `TimesheetRepository` den Status des Timesheets in der Datenbank. Abhängig von der Rolle des Benutzers wird der Status auf `WaitingForApproval` für Betreuer oder `Complete` für HiWis gesetzt.



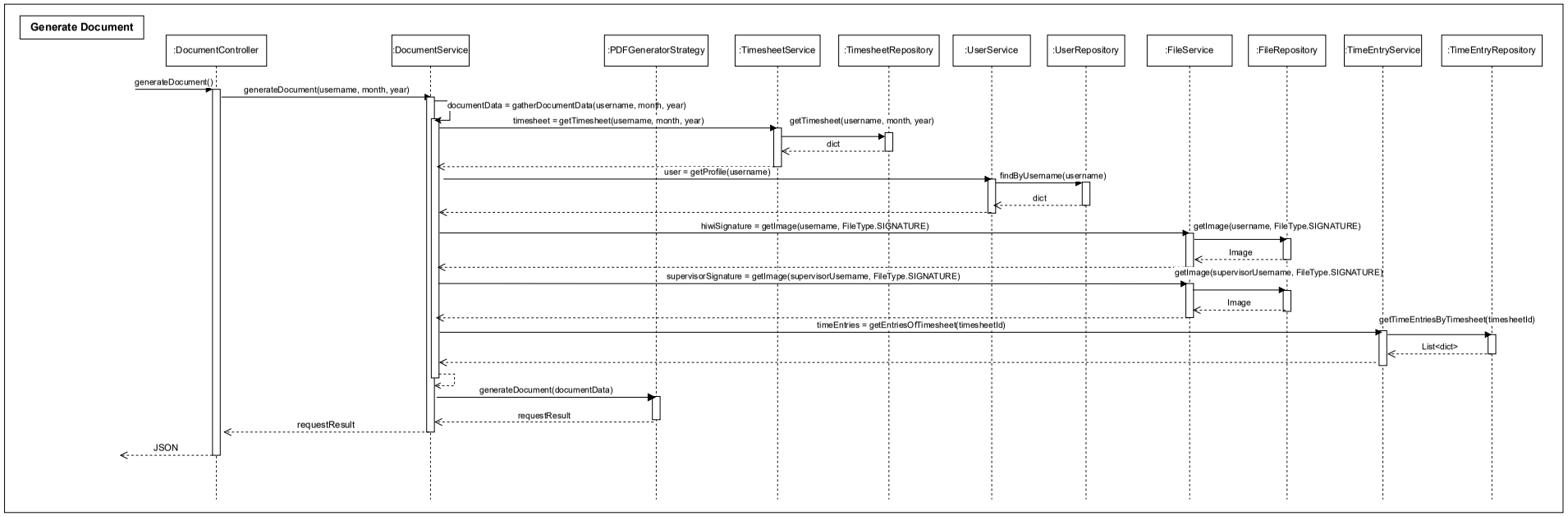
3.3.4 Änderungen an dem Timesheet anfordern

Es wird angenommen, dass der Benutzer die Rolle eines Betreuers innehat. Der `TimesheetController` initiiert den Prozess, indem er die zugehörige Service-Klasse aufruft. Dies geschieht mit der `timesheetId` des zu ändernden Timesheets und einer Beschreibung, die der Betreuer gemäß den Vorgaben des Pflichtenhefts eingegeben hat. Letzteres stellt ein optionales Wunschkriterium dar.
Anschließend setzt das `TimesheetRepository` den Status des Timesheets auf `NotSubmitted`. Diese Änderung ermöglicht es dem HiWi, erneut Änderungen am Timesheet vorzunehmen.
Das Ergebnis dieses Vorgangs wird über ein `requestResult` zurück an den Controller übermittelt, welches den Erfolg oder Misserfolg der Operation dokumentiert.



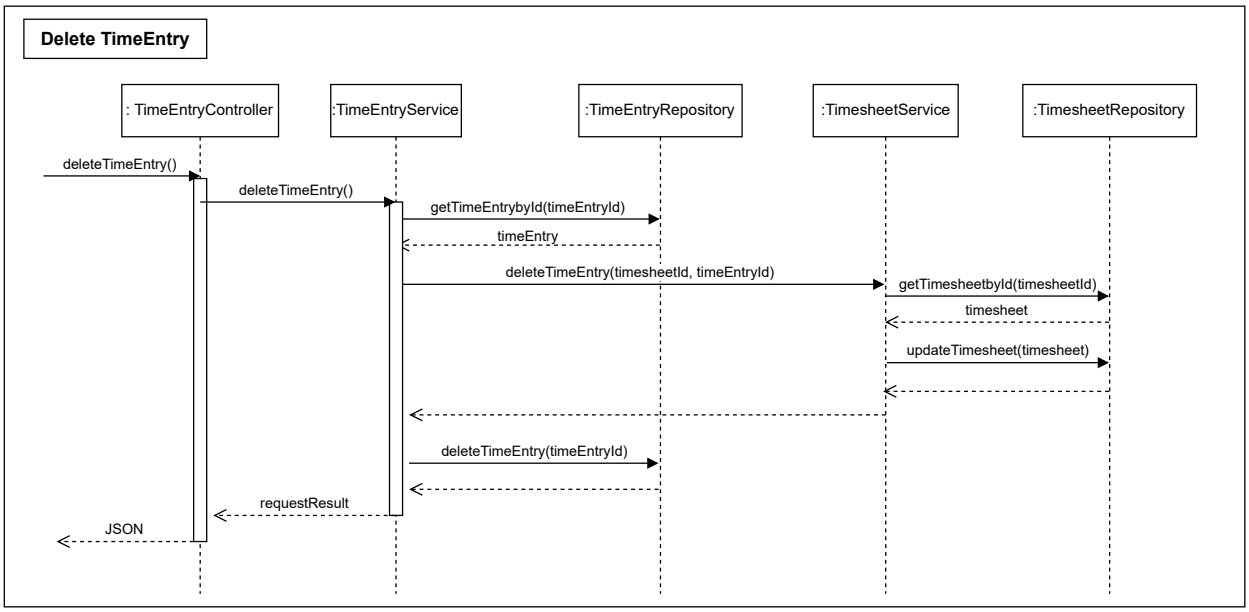
3.3.5 PDF-Dokument generieren

Der `DocumentController` initiiert den Prozess der Dokumentenerstellung, indem er die Methode `generateDocument` aufruft. Dieser Aufruf erfolgt mit spezifischen Parametern: dem Nutzernamen des HiWis, für den das Timesheet erstellt wird, sowie dem Monat und Jahr der Zeiterfassung.
Innerhalb des `DocumentService` erfolgt die eigentliche Verarbeitung. Zunächst ruft dieser Service sich selbst auf, um die benötigten *documentData* zu sammeln. Diese Daten umfassen das Timesheet, die dazugehörigen Time Entries, die Nutzerinformationen und die Signatur-Datei. Jedes dieser Datenstücke wird durch spezialisierte Service-Klassen bezogen, die wiederum mit den entsprechenden Repository-Klassen zusammenarbeiten, um die Informationen direkt aus der Datenbank abzurufen.
Sobald alle erforderlichen Daten gesammelt sind, werden sie in ein `DocumentData`-Objekt zusammengeführt. Dieses Objekt wird dann an die `PDFGeneratorStrategy` übergeben, welche für die Erstellung des endgültigen Dokuments verantwortlich ist.
Das Ergebnis des gesamten Prozesses wird als `RequestResult` zurückgegeben. Dieses Ergebnis zeigt nicht nur den Erfolg oder Misserfolg der Operation an, sondern enthält auch zusätzliche Informationen wie Fehlermeldungen oder den Statuscode der Operation.


Generate Document
:DocumentController
:DocumentService
:PDFGeneratorStrategy
:TimesheetService
:TimesheetRepository
:UserService
:UserRepository
:FileService
:FileRepository
:TimeEntryService
:TimeEntryRepository
generateDocument()
generateDocument(username, month, year)
documentData = gatherDocumentData(username, month, year)
timesheet = getTimesheet(username, month, year)
getTimesheet(username, month, year)
dict
user = getProfile(username)
findByUsername(username)
dict
hiwiSignature = getImage(username, FileType.SIGNATURE)
getImage(username, FileType.SIGNATURE)
Image
supervisorSignature = getImage(supervisorUsername, FileType.SIGNATURE)
getImage(supervisorUsername, FileType.SIGNATURE)
Image
timeEntries = getEntriesOfTimesheet(timesheetId)
getTimeEntriesByTimesheet(timesheetId)
List<dict>
generateDocument(documentData)
requestResult
requestResult
JSON

3.3.6 Zeiteintrag von Timesheet löschen

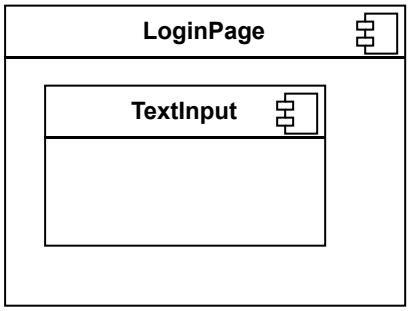
Es wird angenommen, dass der betreffende Nutzer ein HiWi ist. Der Löschvorgang für ein `TimeEntry` wird durch den `TimeEntryController` initiiert und an den `TimeEntryService` weitergeleitet.
Zunächst wird das entsprechende `TimeEntry` anhand seiner ID aus der Datenbank abgerufen. Dies übernimmt das `TimeEntryRepository`. Innerhalb des `TimeEntry`-Objekts befindet sich die ID des zugehörigen `Timesheets`, die notwendig ist, um die Löschmethode `deleteTimeEntry` mit den IDs des `TimeEntry` und des `Timesheets` im `TimesheetService` aufzurufen.
Der `TimesheetService` ruft daraufhin das `Timesheet`-Objekt über das `TimesheetRepository` ab und aktualisiert es, indem das betreffende `TimeEntry` entfernt wird.
Abschließend wird das `TimeEntry` mithilfe des `TimeEntryRepository` aus der Datenbank gelöscht, wodurch der Prozess abgeschlossen wird.



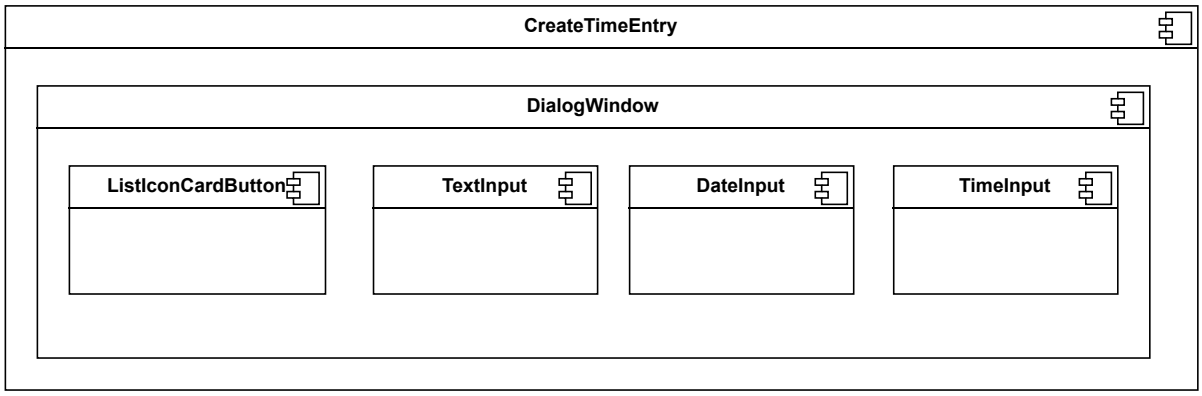
3.4 Komponentendiagramme

Die folgenden Komponentendiagramme sind darauf ausgerichtet, die strukturelle Planung für die Entwicklung des Frontends zu veranschaulichen, speziell unter Verwendung von React-Komponenten. Komponenten, die mehrfach unter demselben Namen auftauchen, sind als wiederverwendbare Einheiten konzipiert. Ein klassisches Beispiel hierfür sind die Komponenten `LayoutWrapper`, `NavigationBar` und `ProfileBar`, die in den meisten Ansichten unserer Webseite zum Einsatz kommen.
Durch diese Wiederverwendung können wir das Frontend in unabhängige, modularisierte Komponenten unterteilen. Diese Modularität ist nicht nur förderlich für die Arbeitsaufteilung im Entwicklungsteam, sondern ermöglicht auch das individuelle Testen jeder einzelnen Komponente, was zur Effizienzsteigerung und Qualitätsverbesserung unseres Entwicklungsprozesses beiträgt.

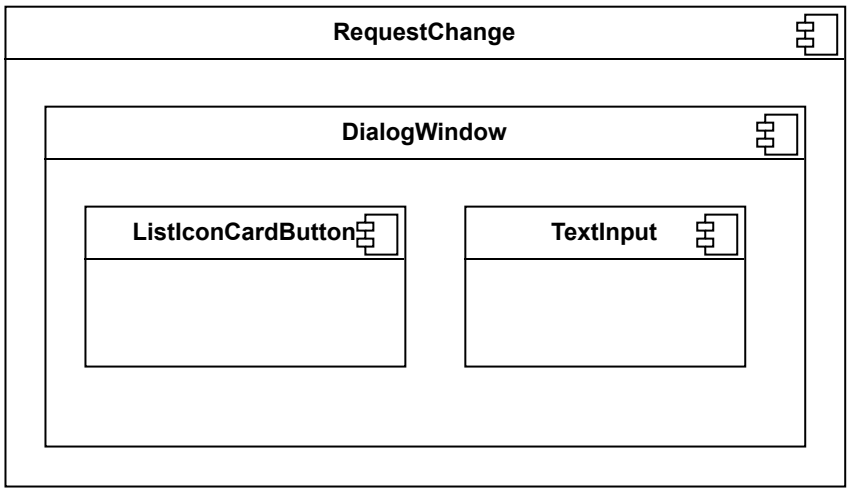
3.4.1 Login



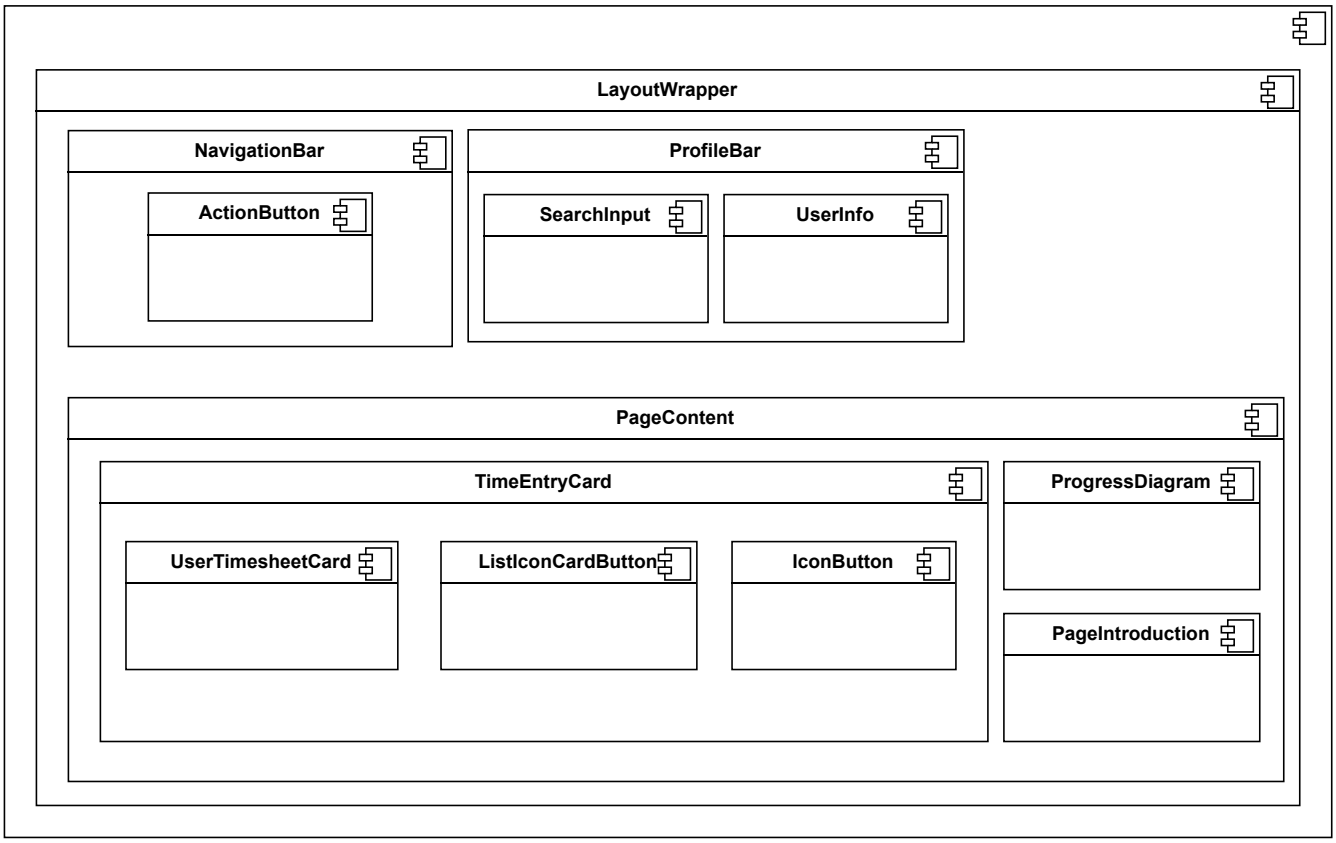
3.4.2 Zeiteintrag erfassen



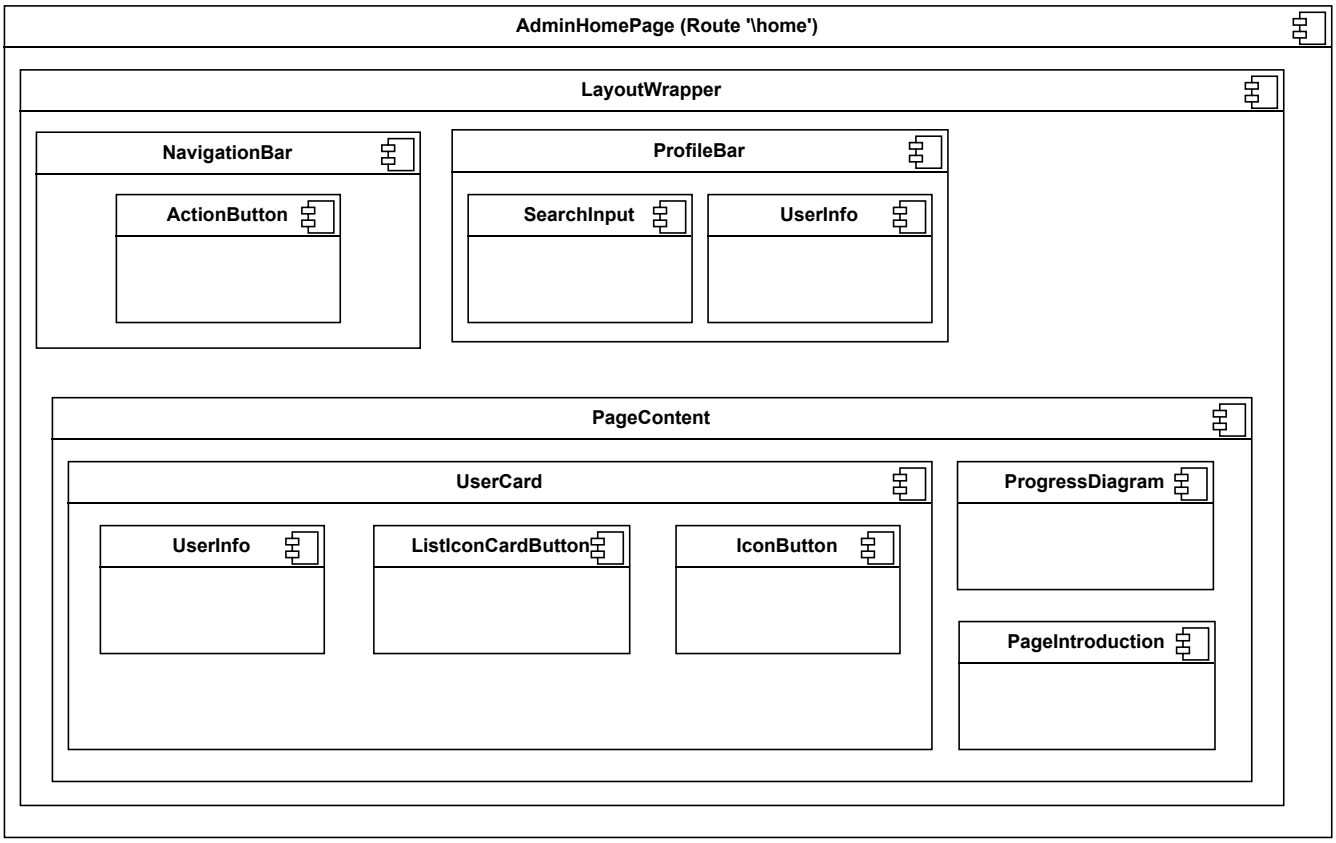
3.4.3 Änderung anfordern



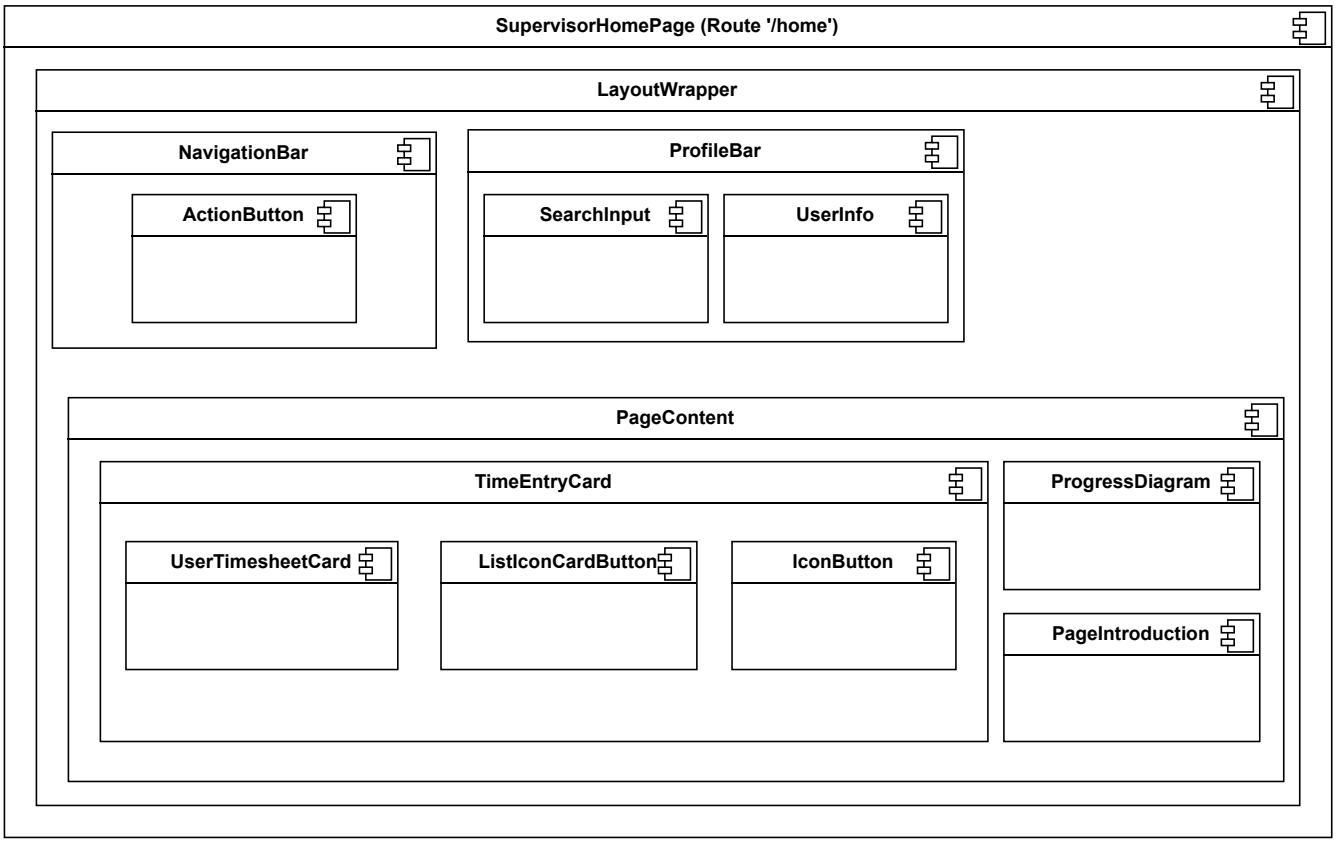
3.4.4 Sekretariat Home Page



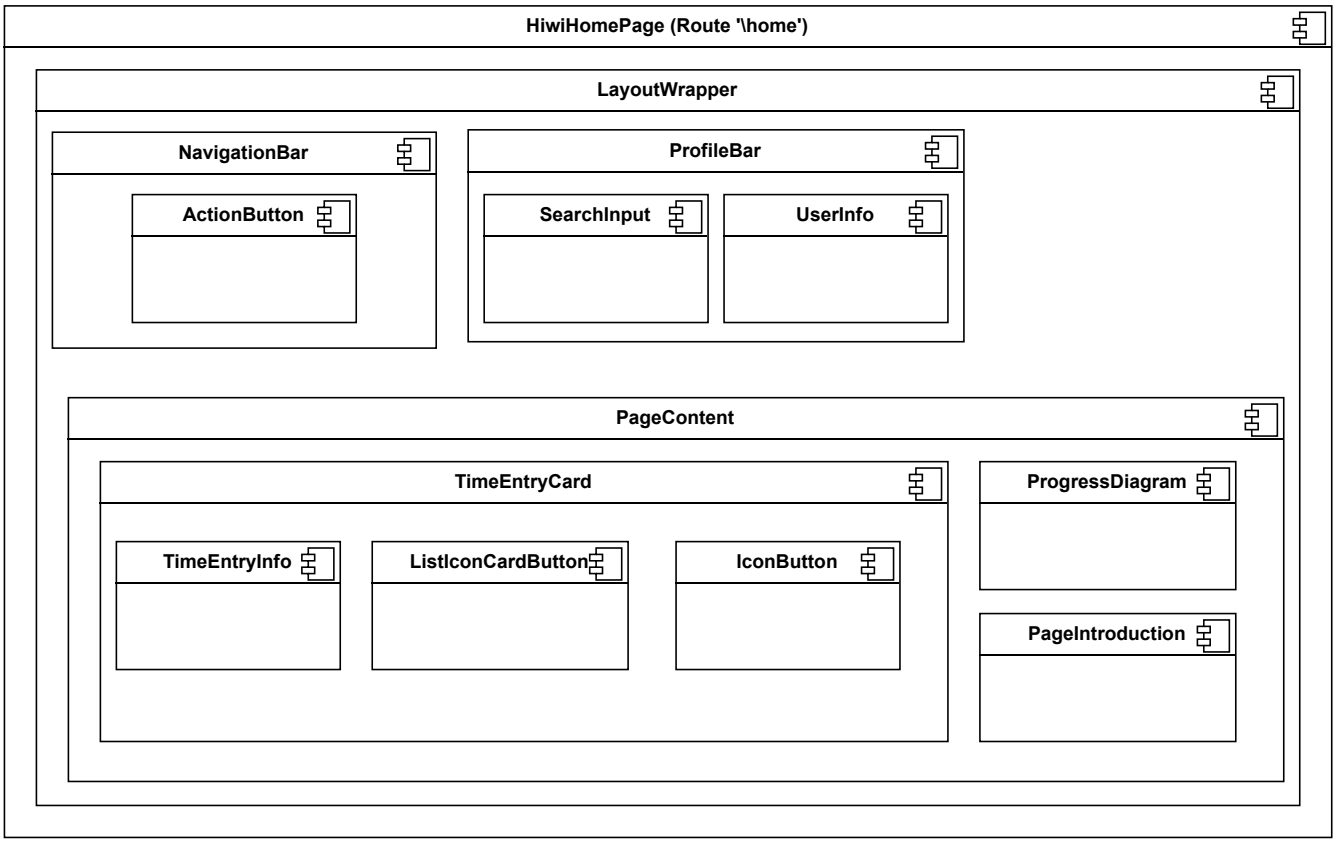
3.4.5 Admin Home Page



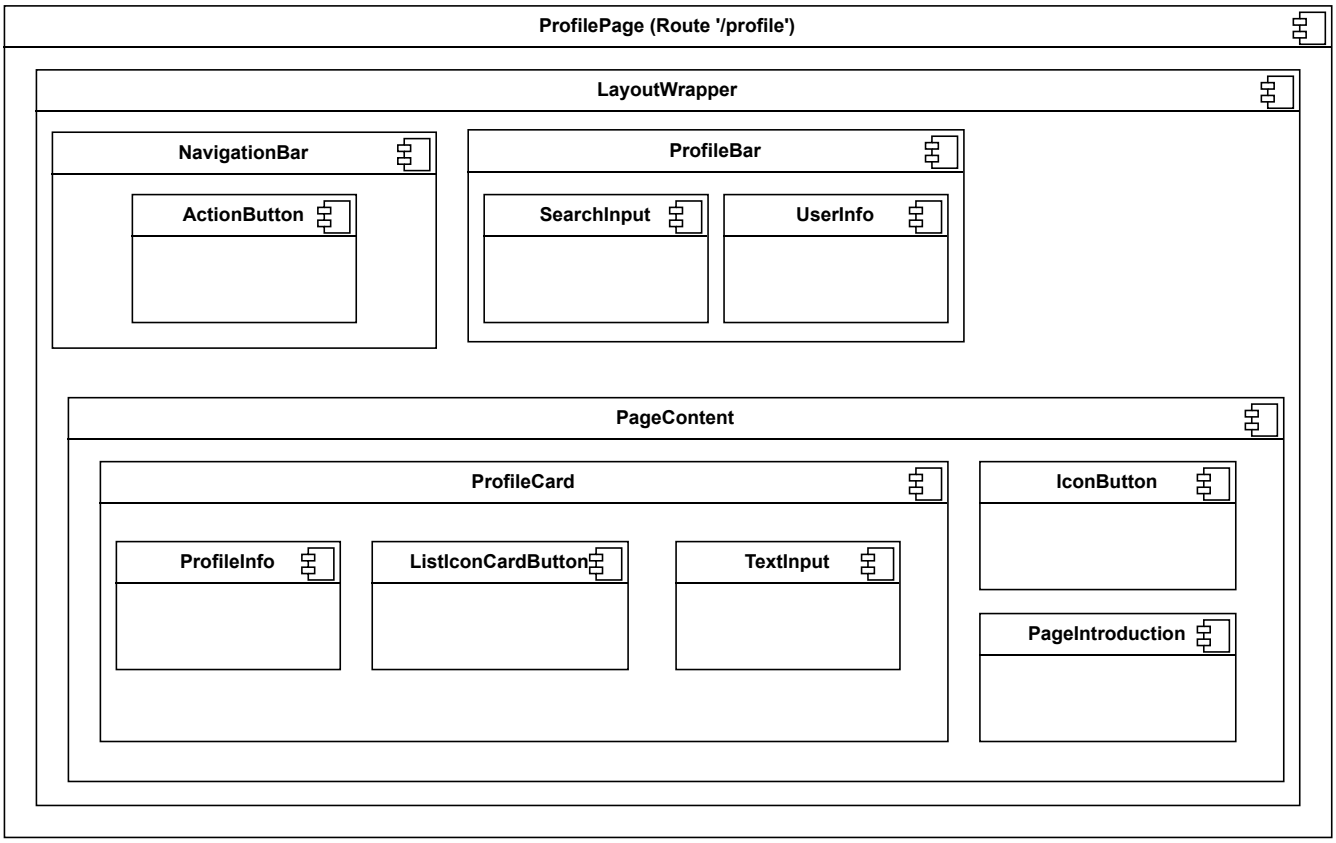
3.4.6 Betreuer Home Page



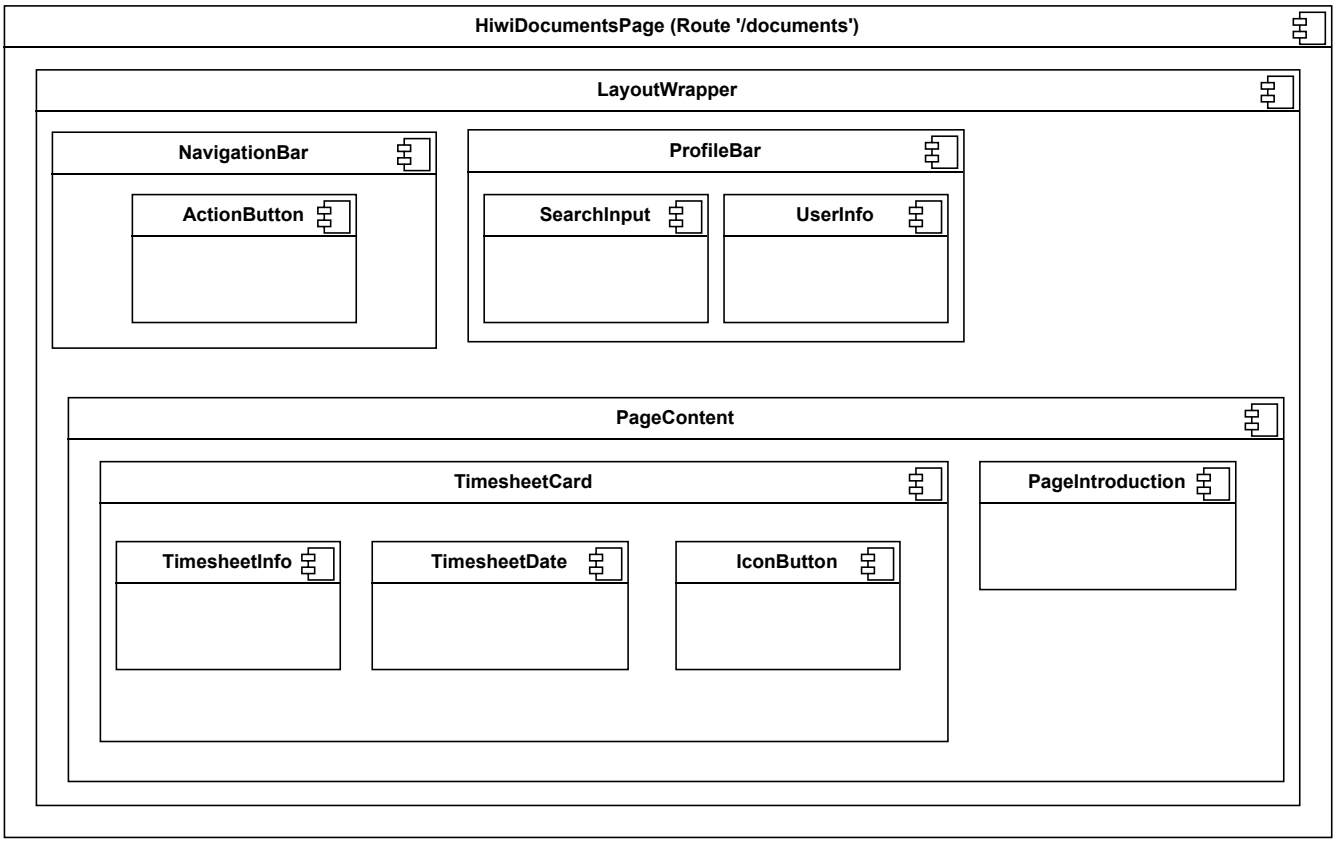
3.4.7 Hiwi Home Page



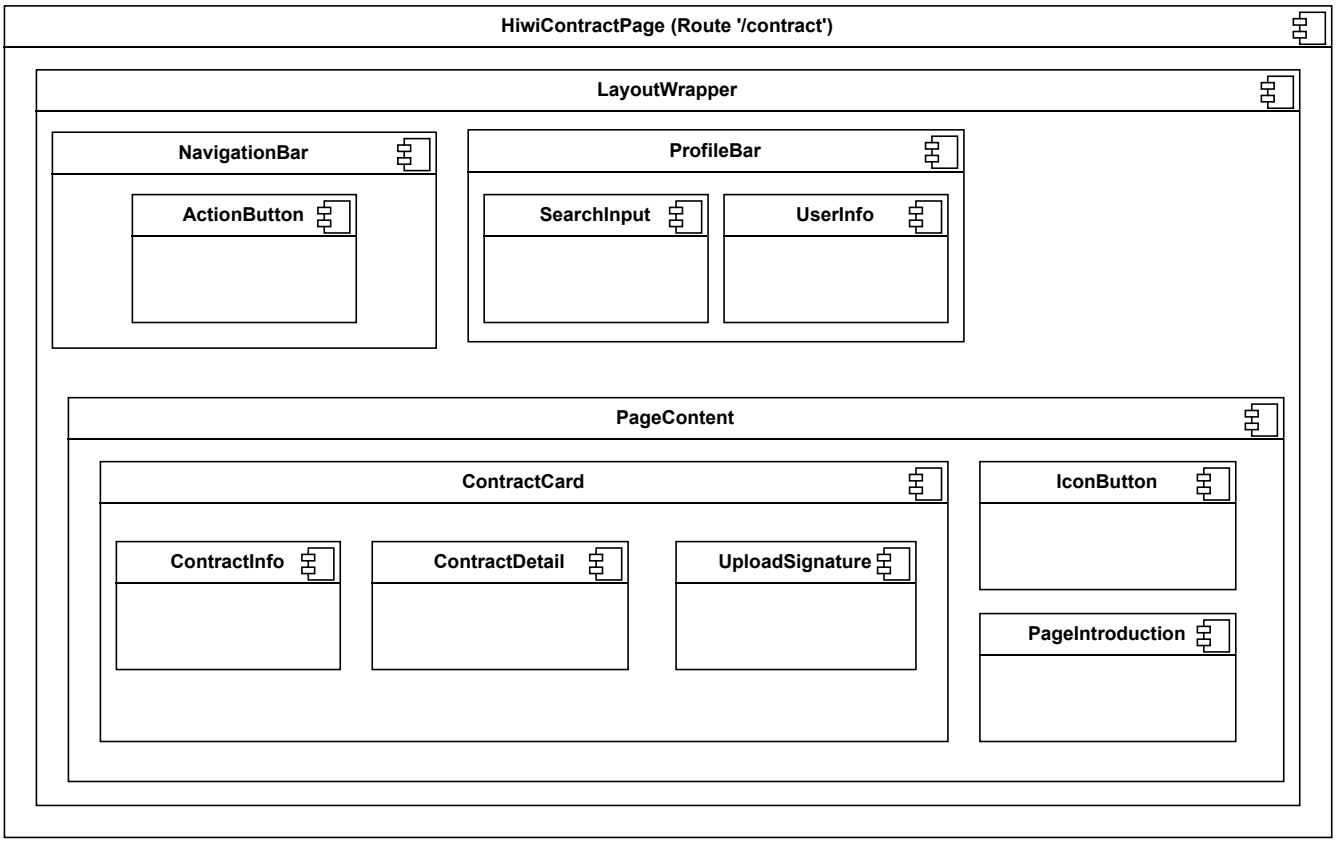
3.4.8 Profile Page



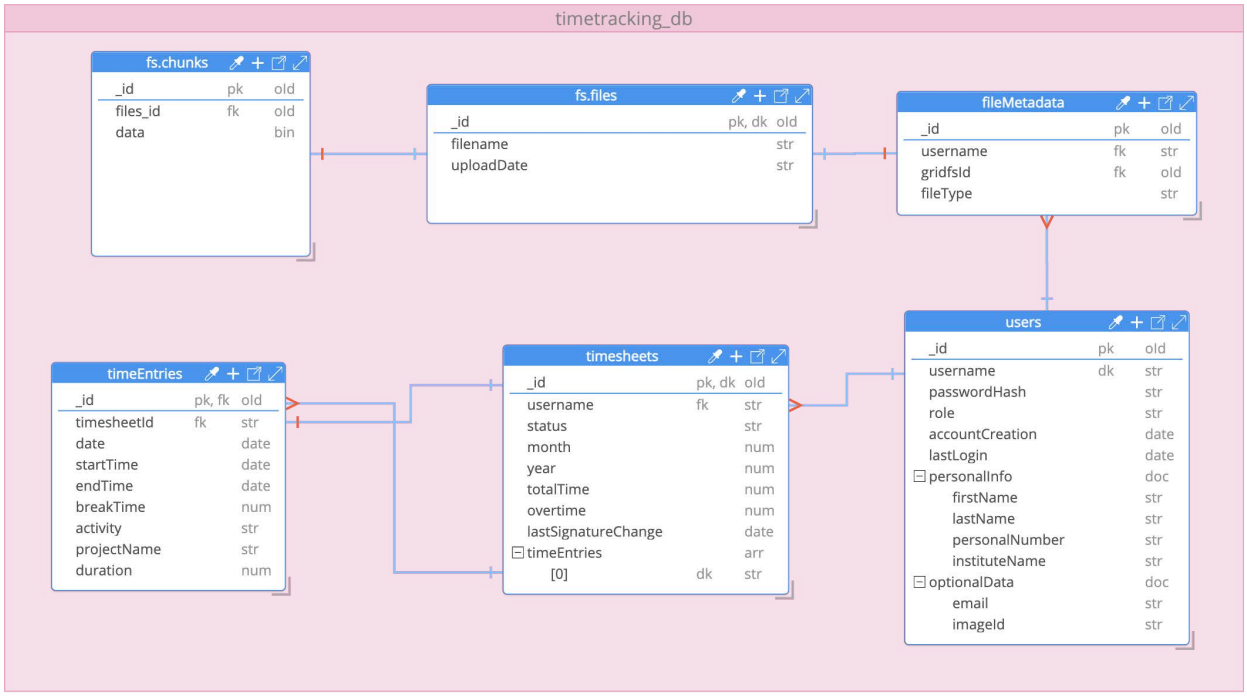
3.4.9 Hiwi Documents Page



3.4.10 Hiwi Contract Page



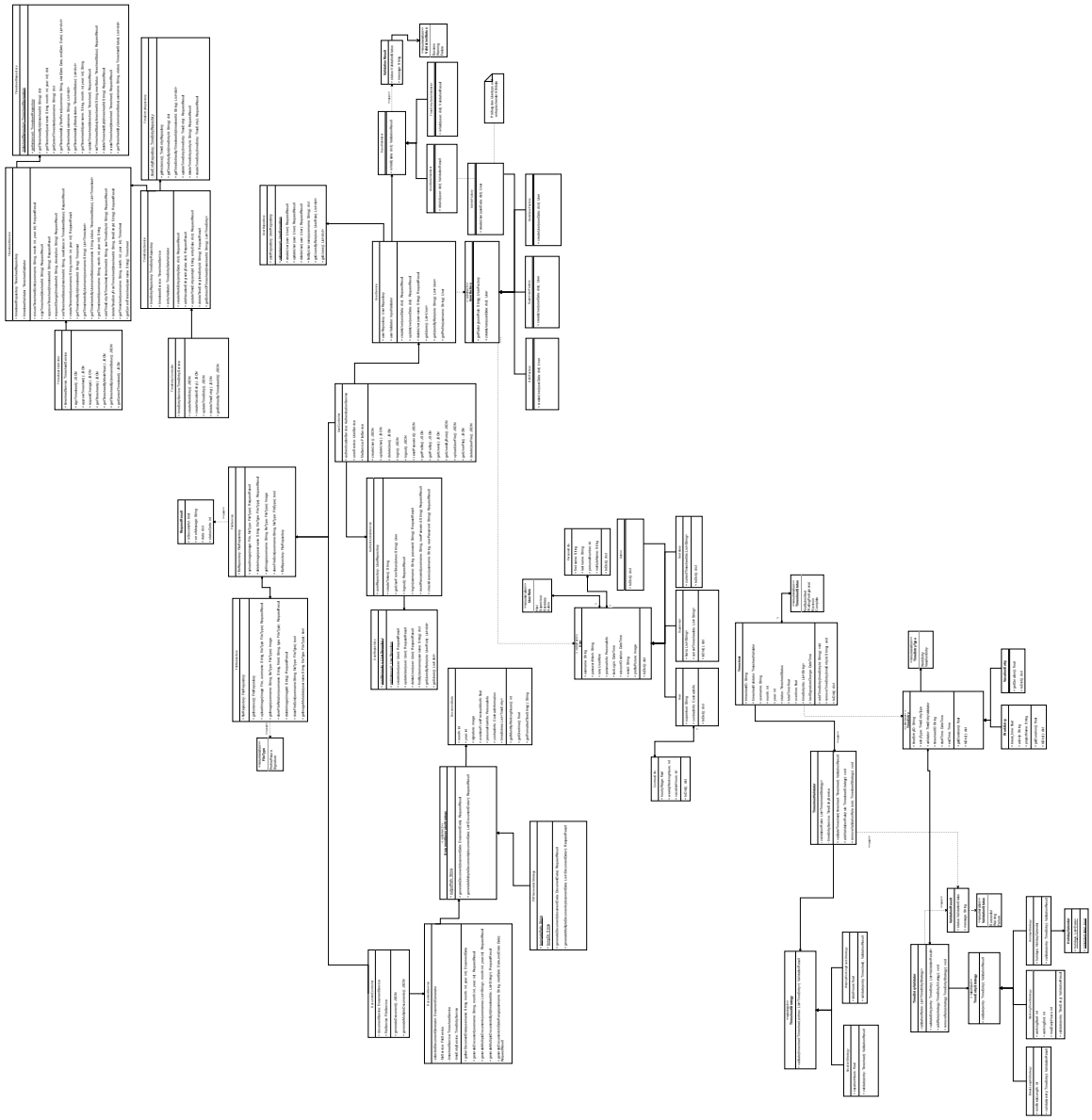
4 Datenbankdiagramm



Das vorliegende Datenbankdiagramm illustriert die Struktur und Beziehungen der Daten innerhalb der MongoDB-Datenbank, wie sie für verschiedene Modellklassen definiert sind. Im Fokus stehen insbesondere die Klassen TimeEntry, Timesheet und User, deren Datenorganisation im unteren Bereich des Diagramms dargestellt wird. Die Beziehungen zwischen den Entitäten sind gemäß der Crow's Foot Notation modelliert, wobei exemplarisch die 1-n Beziehung zwischen üsersünd "timesheets"hervorgehoben wird, welche anzeigt, dass ein User-Objekt mit mehreren Timesheet-Objekten verknüpft sein kann.
In dieser Implementierung wird für jede Modellklasse eine separate MongoDB-Collection angelegt. Besondere Beachtung finden dabei die Collections "fs.chunks", "fs.filesünd "fileMetadata", die speziell für die Dateispeicherung konzipiert sind. Die "fsgekennzeichneten Collections gehören zum GridFS File Service von MongoDB und sind zuständig für die Speicherung binärer Daten ("fs.chunks") sowie der dazugehörigen Metadaten ("fs.files"). Die zusätzlich definierte Collection "fileMetadataërleichtert die Nachverfolgung, welcher Nutzer zu welchen Dateien zugeordnet ist.

5 Anhang

5.1 Klassendiagramm



5.2 Code Dokumentation

CONTENTS:

CHAPTER

ONE

BACKEND

1.1 app module

This is the main python module where we call the run method of the flask app.

app.**check_mongodb_connection**()

Check the connection to the MongoDB database

Returns

A string indicating the connection status

app.**create_time_entry**()

Creates a test time entry in the database

app.**create_timesheet**()

Creates a test timesheet in the database

app.**create_user**()

Creates a test user in the database

Returns

A string indicating that the user was created

app.**home**()

app.**read_time_entries**()

Reads all time entries from the database

Returns

A JSON string containing all time entries

app.**read_timesheets**()

Reads all timesheets from the database

Returns

A JSON string containing all timesheets

app.**timesheet_to_dict**(*timesheet*)

app.**work_entry_to_dict**(*work_entry*)

1.2 auth module

auth.**init_auth_routes**(*app*)

This function initializes the authentication routes for the application.

Parameters
app – The flask application.

1.3 controller package

1.3.1 Subpackages

controller.factory package

Submodules

controller.factory.admin_factory module

class controller.factory.admin_factory.**AdminFactory**

Bases: *UserFactory*

A factory class for creating Admin user objects. Extends the UserFactory class.

_abc_impl = <_abc._abc_data object>

create_user(*user_data: dict*) → *User*

Creates and returns an Admin object based on the provided data.

Parameters
user_data – A dictionary containing user details including username, password hash, personal information and role.

Returns
A User object initialized as an Admin with the provided data.

controller.factory.hiwi_factory module

class controller.factory.hiwi_factory.**HiwiFactory**

Bases: *UserFactory*

A factory class for creating HiWi user objects. Extends the UserFactory class.

_abc_impl = <_abc._abc_data object>

create_user(*user_data: dict*) → *Hiwi*

Creates and returns a Hiwi object based on the provided data.

Parameters
user_data – A dictionary containing user details including username, password hash, personal information, supervisor, and contract information.

Returns
A Hiwi object initialized with the provided data.

controller.factory.secretary_factory module

class controller.factory.secretary_factory.**SecretaryFactory**

Bases: *UserFactory*

A factory class for creating Secretary user objects. Extends the UserFactory class.

_abc_impl = <_abc._abc_data object>

create_user(*user_data: dict*) → *User*

Creates and returns a User object for a Secretary based on the provided data.

Parameters
user_data – A dictionary containing user details including username, password hash, personal information, and role.

Returns
A User object initialized with the provided data.

controller.factory.supervisor_factory module

class controller.factory.supervisor_factory.**SupervisorFactory**

Bases: *UserFactory*

A factory class for creating Supervisor user objects. Extends the UserFactory class.

_abc_impl = <_abc._abc_data object>

create_user(*user_data: dict*) → *Supervisor*

Creates and returns a Supervisor object based on the provided data.

Parameters
user_data – A dictionary containing user details including username, password hash, personal information, and a list of hiwis.

Returns
A Supervisor object initialized with the provided data.

controller.factory.user_factory module

class controller.factory.user_factory.**UserFactory**

Bases: ABC

Abstract base class for creating User objects based on user roles, including the factory method pattern.

_abc_impl = <_abc._abc_data object>

static _role_factory_mapping()

Returns a mapping of user roles to their corresponding factory classes.

This method imports the necessary factory classes and creates a dictionary that maps user role values to their respective factory classes. This mapping allows for dynamic creation of user instances based on their roles.

Returns
A dictionary where the keys are user role values and the values are the corresponding factory classes.

Return type
dict

abstract create_user(*user_data: dict*) → *User*

Abstract method to create a user instance based on the provided user data.

Parameters
user_data – A dictionary containing data needed to create a user instance.

Returns
An instance of a subclass of User.

static create_user_if_factory_exists(*user_data*) → *User*

Create a user if the corresponding factory exists based on the user role specified in user_data.

Parameters
user_data – A dictionary containing data needed to create a user instance.

Returns
An instance of a subclass of UserFactory or None.

static get_factory(*user_role: str*)

Returns the factory instance associated with the given user role.

Parameters
user_role – The role of the user for which the factory is requested.

Returns
An instance of the appropriate factory if found, otherwise None.

Module contents

controller.input_validator package

Submodules

controller.input_validator.input_validator module

class controller.input_validator.input_validator.**InputValidator**

Bases: object

Base class for input validators.

is_valid(*data: dict*) → *ValidationResult*

Validate the input data. Should be overridden by subclasses.

Parameters
data – The data to validate

Returns
if data is valid, return True, otherwise False

controller.input_validator.time_entry_data_validator module

class controller.input_validator.time_entry_data_validator.**TimeEntryDataValidator**

Bases: *InputValidator*

A validator class for time entry data in a project management system. This class checks the validity of different fields specific to time entries.

_get_entry_class(*entry_type_value*)

Maps entry type values to specific TimeEntry subclass.

Parameters
entry_type_value – The entry type value to use.

Returns
Corresponding TimeEntry subclass or None if invalid.

_validate_entry_type(*entry_type_value*)

Validates that the provided entry type value represents a valid TimeEntryType.

Parameters
entry_type_value – The entry type value to validate.

Returns
Boolean indicating whether the entry type value is valid.

is_valid(*time_entry_data*)

Validates the completeness and correctness of time entry data based on the entry type.

Parameters
time_entry_data – A dictionary containing time entry details to validate.

Returns
ValidationResult: An object indicating whether the time entry data is valid, including a message and status code.

controller.input_validator.user_data_validator module

class controller.input_validator.user_data_validator.**UserDataValidator**

Bases: *InputValidator*

A class that validates user data against predefined regular expression patterns to ensure data conforms to the expected formats.

is_valid(*user_data*)

Validates the completeness and correctness of user data using regex.

Parameters
user_data – A dictionary containing user details to validate.

Returns
ValidationResult indicating if the user data is valid.

validate_role(*role: str*)

Validates that the provided role value represents a valid UserRole.

Parameters
role – The role value to validate.

Returns
Boolean, True if the role value is valid, otherwise False.

controller.input_validator.validation_result module

class controller.input_validator.validation_result.**ValidationResult**(*status: ValidationStatus*, *message: str*)

Bases: object

Represents the result of a validation operation, including the status and a message.

Parameters

- **status** – The status of the validation, indicating success or failure.
- **message** – A message providing additional information about the validation result.

Variables

- **status** – Stores the status of the validation after the object is initialized.
- **message** – Stores the message providing details about the validation result.

controller.input_validator.validation_status module

class controller.input_validator.validation_status.**ValidationStatus**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum representing the status of a validation operation.

This Enum defines the possible outcomes of a validation check, helping to standardize the response across different validation operations within the application.

Variables

- **SUCCESS** – Indicates that the validation was successful.
- **WARNING** – Indicates that there is a warning in the validation.
- **FAILURE** – Indicates that the validation failed.

FAILURE = 3

SUCCESS = 1

WARNING = 2

Module contents

1.3.2 Submodules

1.3.3 controller.time_entry_controller module

class controller.time_entry_controller.**TimeEntryController**

Bases: MethodView

Controller for handling requests related to time entries, providing methods for POST and GET requests.

_dispatch_request(*endpoint_mapping*)

Dispatches the request to the appropriate handler based on the request path.

Parameters
endpoint_mapping – Dictionary mapping endpoints to function handlers.

Returns
The response from the handler or an error message if endpoint not found.

create_vacation_entry()

Creates a new vacation time entry with the provided JSON data.

Returns
JSON response containing the status message and status code.

create_work_entry()

Creates a new time entry with the provided JSON data.

Returns
JSON response containing the status message and status code.

delete_time_entry()

Deletes a time entry identified by its ID provided in the JSON data.

Returns
JSON response containing the status message and status code.

get()

Handles GET requests for retrieving time entries by their timesheet ID.

Returns
JSON response with time entries or an error message.

get_entries_by_timesheet_id()

Retrieves all time entries associated with a specific timesheet ID based on the provided JSON data.

Returns
JSON response containing a list of time entries.

methods: t.ClassVar[t.Collection[str] | None] = {'GET', 'POST'}

The methods this view is registered for. Uses the same default (`["GET", "HEAD", "OPTIONS"]`) as `route` and `add_url_rule` by default.

post()

Handles POST requests to manage time entry-related actions such as creating time entries, vacation time, updating entries, and deleting entries based on the specific endpoint.

Returns
JSON response with the result of the operation.

update_time_entry()

Updates an existing time entry using the provided JSON data.

Returns
JSON response containing the status message and status code.

1.3.4 controller.timesheet_controller module

class controller.timesheet_controller.**TimesheetController**

Bases: MethodView

Controller for managing timesheet-related operations such as retrieval, creation, modification, and deletion of timesheet entries.

_dispatch_request(*endpoint_mapping*)

Dispatches the request to the appropriate handler based on the request path.

Parameters
endpoint_mapping – Dictionary mapping endpoints to function handlers.

Returns
The response from the handler or an error message if the endpoint is not found.

approve_timesheet()

Allows a supervisor to approve a timesheet.

Returns
JSON response containing the status message and status code.

ensure_timesheet_exists()

Ensures that a timesheet exists for the given username, month, and year. If the timesheet does not exist, it will be created. :return: The timesheet object

get()

Handles GET requests for retrieving timesheet data based on the endpoint.

Returns
JSON response with the timesheet data or an error message.

get_current_timesheet()

Retrieves the current timesheet for a user.

Returns
JSON response containing the current timesheet data or an error message.

get_timesheets()

Retrieves timesheets for the current user.

Returns
JSON response containing a list of timesheets or an error message.

get_timesheets_by_month_year()

Retrieves timesheets for the current user by month and year.

Returns
JSON response containing the timesheet data or an error message.

get_timesheets_by_username_status()

Retrieves timesheets for a specific user and status.

Returns
JSON response containing a list of timesheets or an error message.

methods: t.ClassVar[t.Collection[str] | None] = {'GET', 'PATCH', 'POST'}

The methods this view is registered for. Uses the same default (`["GET", "HEAD", "OPTIONS"]`) as `route` and `add_url_rule` by default.

patch()

Handles PATCH requests for updating timesheet data based on the endpoint.

Returns
JSON response with the result of the update operation or an error message.

post()

Handles POST requests for creating timesheets

request_change()

Allows a supervisor to request changes to a timesheet.

Returns
JSON response containing the status message and status code.

sign_timesheet()

Allows a HiWi to sign his timesheet.

Returns
JSON response containing the status message and status code.

1.3.5 controller.user_controller module

class controller.user_controller.**UserController**

Bases: MethodView

Controller for managing user-related actions like creating, updating, and deleting users, managing login sessions, and handling user files.

_dispatch_request(*endpoint_mapping*)

Dispatches the request to the appropriate handler based on the request path.

Parameters
endpoint_mapping – Dictionary mapping endpoints to function handlers.

Returns
The response from the handler or an error message if endpoint not found.

create_user()

Creates a new user with the provided JSON data.

Returns
JSON response containing the status message and status code.

delete()

Handles DELETE requests to delete user files.

Returns
JSON response indicating the outcome of the file deletion.

delete_user()

Deletes a user identified by their username provided in JSON data.

Returns
JSON response containing the status message and status code.

delete_user_file()

Deletes a file associated with the given username and specified file type.

Returns
A JSON response with the result of the deletion attempt and an appropriate HTTP status code.

get()

Handles GET requests for retrieving user profiles, user lists, or users by specific roles.

Returns
JSON response with user data or an error message.

get_profile()

Retrieves the profile information for the currently authenticated user.

Returns
JSON response containing the user's profile data.

get_user_file()

Retrieves a file for the specified username and file type

Returns
The file as a download if found, or a JSON response indicating an error with an appropriate HTTP status code if not found or if parameters are missing.

get_users()

Retrieves a list of all users in the system.

Returns
JSON response containing a list of user profiles.

get_users_by_role()

Retrieves a list of users filtered by a specific role provided via query parameters.

Returns
JSON response containing a list of user profiles filtered by role.

login()

Deletes a user identified by their username provided in JSON data.

Returns
JSON response containing the status message and status code.

logout()

Logs out a user by terminating their session or token.

Returns
JSON response with the logout confirmation message.

methods: `t.ClassVar[t.Collection[str] | None] = {'DELETE', 'GET', 'POST'}`

The methods this view is registered for. Uses the same default (`["GET", "HEAD", "OPTIONS"]`) as `route` and `add_url_rule` by default.

post()

Handles POST requests to manage user-related actions based on the specific endpoint.

Returns
JSON response based on the action performed.

reset_password()

Resets the password for a user based on the provided JSON data.

Returns
JSON response with the result of the password reset attempt.

update_user()

Updates an existing user's data with the provided JSON data.

Returns
JSON response containing the status message and status code.

upload_user_file()

Handles the uploading of a file for the currently authenticated user. The file type is determined from form data or a query parameter, and the file is then processed and stored accordingly.

Returns
JSON response with the status message and HTTP status code.

1.3.6 Module contents

1.4 db module

db.**check_db_connection**()

db.**initialize_db**()

Initializes the database connection

1.5 model package

1.5.1 Subpackages

model.file package

Submodules

model.file.FileType module

class model.file.FileType.**FileType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum to represent different types of file classifications within the system.

This enum defines specific categories of files that can be managed within the application, such as profile pictures and signatures. This classification helps in handling file operations more securely and contextually.

PROFILE_PICTURE

Represents an image used as a profile picture.

Type
FileType

SIGNATURE

Represents an image used as a user's signature.

Type
FileType

PROFILE_PICTURE = 'Profile Picture'

SIGNATURE = 'Signature'

static get_type_by_value(*value: str*)

Searches for and returns the FileType enum member that matches the provided string value.

Parameters
value – The string value to match against FileType values.

Returns
The matching FileType enum member if found, otherwise None.

Module contents

model.repository package

Submodules

model.repository.FileRepository module

class model.repository.FileRepository.**FileRepository**

Bases: object

Manages files stored in MongoDB using GridFS. This class handles the uploading, updating, retrieving, and deleting of files along with their metadata, providing a high-level abstraction over the direct GridFS interactions.

_instance = None

delete_image(*gridfs_id: str*) → *RequestResult*

Deletes an image and its associated metadata from GridFS and MongoDB based on the GridFS ID provided.

Parameters
gridfs_id – The GridFS ID of the image to delete.

Returns
A RequestResult indicating success or failure of the deletion, including a message and a status code.

does_file_exist(*username*, *file_type:* FileType)

Checks whether a file exists in the database based on the username and file type specified, useful for validation before file operations.

Parameters

- **username** – The username to check for the file.
- **file_type** – The type of file to check.

Returns
True if the file exists in the database, False otherwise.

get_image(*username: str*, *file_type:* FileType)

Retrieves an image file from GridFS based on the username and file type specified, ensuring files are retrieved securely according to user and type.

Parameters

- **username** – The username associated with the file.
- **file_type** – The type of file to retrieve.

Returns
A file-like object containing the image data if found, otherwise None.

get_image_metadata(*username: str*, *file_type:* FileType)

Retrieves metadata of an image based on the username and file type specified, providing detailed information about the file stored in the database.

Parameters

- **username** – The username associated with the image.
- **file_type** – The type of the image.

Returns
A dictionary containing the metadata of the image if found, otherwise None.

static get_instance()

Provides a singleton instance of FileRepository, ensuring that only one instance of the repository exists throughout the application lifecycle.

Returns
The singleton instance of the FileRepository.

update_image(*file_content*, *gridfs_id: str*, *username: str*, *file_type:* FileType) → *RequestResult*

Updates an existing image in GridFS and its metadata in the MongoDB database.

Parameters

- **file_content** – The new content for the file.
- **gridfs_id** – The GridFS ID of the file to update.
- **username** – The username associated with the file.
- **file_type** – The enum representing the type of the file.

Returns
A RequestResult object indicating success or failure of the file update.

upload_image(*file*, *username: str*, *file_type:* FileType) → *RequestResult*

Uploads an image to MongoDB using GridFS, storing associated metadata.

Parameters

- **(FileStorage)** (*file*) – The file object to be uploaded.
- **(str)** (*username*) – The username associated with the file upload.
- **(FileType)** (*file_type*) – The enum representing the type of the file.

Returns
A RequestResult object indicating success or failure of the file upload, including a message and a status code.

model.repository.time_entry_repository module

class model.repository.time_entry_repository.**TimeEntryRepository**

Bases: object

Repository class for managing TimeEntry objects in a MongoDB database via GridFS. Handles the operations for uploading, updating, retrieving, and deleting time entries along with their associated metadata.

_instance = None

create_time_entry(*time_entry:* TimeEntry)

Creates a new TimeEntry object in the MongoDB database.

Parameters
time_entry – The TimeEntry object to create.

Returns
A RequestResult indicating the success or failure of the creation operation.

delete_time_entry(*entry_id: str*)

Deletes a TimeEntry object from the MongoDB database using its ID.

Parameters
entry_id – The ID of the TimeEntry to delete.

Returns
A RequestResult indicating the success or failure of the delete operation.

static get_instance()

Provides a singleton instance of TimeEntryRepository to ensure only one instance is used throughout the application.

Returns
The singleton instance of TimeEntryRepository.

get_time_entries()

Retrieves all TimeEntry objects from the database

Returns
A list of all TimeEntry objects

get_time_entries_by_date(*date*, *username*)

Retrieves all TimeEntry objects for a specific date and username.

Parameters

- **date** – The date for which to retrieve TimeEntry objects.
- **username** – The username associated with the TimeEntry objects.

Returns
A list of TimeEntry objects for the specified date and username.

get_time_entries_by_timesheet_id(*timesheet_id: str*)

Retrieves all TimeEntry objects associated with a specific timesheet ID from the MongoDB database.

Parameters
timesheet_id – The timesheet ID to query for TimeEntry objects.

Returns
A list of TimeEntry objects linked to the specified timesheet.

get_time_entry_by_id(*time_entry_id*)

Retrieves a TimeEntry object from the MongoDB database using its ID.

Parameters
time_entry_id – The MongoDB ObjectId of the TimeEntry to retrieve.

Returns
The TimeEntry object if found, otherwise None.

update_time_entry(*time_entry:* TimeEntry) → *RequestResult*

Updates an existing TimeEntry object in the MongoDB database.

Parameters
time_entry – The TimeEntry object to update.

Returns
A RequestResult indicating the success or failure of the update operation.

model.repository.timesheet_repository module

class model.repository.timesheet_repository.**TimesheetRepository**

Bases: object

Repository class for managing Timesheet objects within a MongoDB database. Provides functions for creating, retrieving, updating, and deleting timesheets, as well as managing their statuses.

_instance = None

create_timesheet(*timesheet:* Timesheet)

Creates a new Timesheet in the database.

Parameters
timesheet – The Timesheet object to create.

Returns
A RequestResult indicating the success or failure of the creation operation.

delete_timesheet(*timesheet_id*) → *RequestResult*

Deletes a Timesheet from the database based on its ObjectId.

Parameters
timesheet_id – The ObjectId of the Timesheet to delete.

Returns
A RequestResult indicating the success or failure of the deletion.

`get_current_timesheet`(*username: str*)

Retrieves the most recent timesheet for a given username that has not been submitted.

Parameters
`username` – The username for whom to retrieve the current timesheet.

Returns
The current Timesheet object if found; otherwise, None.

`static get_instance`()

Provides a singleton instance of TimesheetRepository, ensuring it operates with a single state across the application.

Returns
The singleton instance of TimesheetRepository.

`get_timesheet`(*username: str*, *month: int*, *year: int*)

Retrieves a Timesheet from the database based on the username, month, and year.

Parameters

- `username` – The username associated with the timesheet.
- `month` – The month for which the timesheet is recorded.
- `year` – The year for which the timesheet is recorded.

Returns
The Timesheet object if found; otherwise, None.

`get_timesheet_by_id`(*timesheet_id*)

Retrieves a Timesheet object from the MongoDB database using its ObjectId.

Parameters
`timesheet_id` – The MongoDB ObjectId of the Timesheet to retrieve.

Returns
The Timesheet object if found; otherwise, None.

`get_timesheet_by_status`(*status: TimesheetStatus*)

Retrieves all Timesheet objects from the database with the given status

Parameters
`status` – The status of the timesheet

Returns
A list of Timesheet objects with the given status

`get_timesheet_by_time_period`(*username: str*, *start_date: date*, *end_date: date*)

Retrieves timesheets for a specified time period for a given username.

Parameters

- `username` – The username for whom to retrieve timesheets.
- `start_date` – The start date of the time period.
- `end_date` – The end date of the time period.

Returns
A list of Timesheet objects within the specified date range.

get_timesheet_id(*username: str*, *month: int*, *year: int*)

Retrieves the timesheet ID for the given username, month, and year

Parameters

- **username** – Username of the Hiwi
- **month** – Month of the timesheet
- **year** – Year of the timesheet

Returns

The timesheet ID if found, otherwise None

get_timesheets()

Retrieves all Timesheet objects from the database

Returns

A list of all Timesheet objects

get_timesheets_by_username(*username: str*)

Retrieves all timesheets for a given username

Parameters

username – Username of the Hiwi

Returns

A list of timesheets for the given username

get_timesheets_by_username_status(*username: str*, *status:* TimesheetStatus)

Retrieves all timesheets for a given username with a specified status

Parameters

- **username** – Username of the Hiwi
- **status** – Status of the timesheet

Returns

A list of timesheets for the given username with the specified status

set_timesheet_status(*timesheet_id*, *status*) → *RequestResult*

Updates the status of a timesheet in the database.

Parameters

- **timesheet_id** – The ID of the timesheet to update.
- **status** – The new status of the timesheet.

Returns

A RequestResult object indicating the success of the operation.

update_timesheet(*timesheet*) → *RequestResult*

Updates a specific Timesheet in the database.

Parameters

timesheet – The Timesheet object to update.

Returns

A RequestResult indicating the success or failure of the update operation.

model.repository.user_repository module

class model.repository.user_repository.**UserRepository**

Bases: object

Repository class for managing user data in the database. It provides functionalities for creating, retrieving, updating, and deleting users, as well as retrieving them by specific criteria such as username or role.

_instance = <model.repository.user_repository.UserRepository object>

create_user(*user:* User)

Creates a new user in the database. Checks if the user already exists to prevent duplicates.

Parameters
user – The User object to be created in the database.

Returns
RequestResult indicating the success or failure of the create operation,

delete_user(*username*) → *RequestResult*

Deletes a user from the database by their username.

Parameters
username – The username of the user to delete.

Returns
RequestResult indicating the success or failure of the deletion process.

find_by_username(*username*)

Retrieves a user's data from the database by their username.

Parameters
username – The username of the user to find.

Returns
A dictionary with the user's data if found, otherwise None.

static get_instance()

Retrieves the singleton instance of the UserRepository class.

Returns
Singleton instance of UserRepository, ensuring that only one instance exists globally.

get_users() → list[dict]

Retrieves all users from the database.

Returns
A list of dictionaries, each containing the data of one user.

get_users_by_role(*role:* UserRole) → list[dict]

Retrieves all users from the database with a specified role.

Parameters
role – The UserRole to filter users by.

Returns
A list of dictionaries, each containing the data of one user with the specified role.

update_user(*user:* User) → *RequestResult*

Updates an existing user in the database based on the provided User object.

Parameters
user – The User object containing updated data for the user.

Returns
RequestResult indicating the success or failure of the update operation.

Module contents

model.time_entry_validator package

Submodules

model.time_entry_validator.break_length_strategy module

class model.time_entry_validator.break_length_strategy.**BreakLengthStrategy**

Bases: *TimeEntryStrategy*

Strategy class for validating the break times of a TimeEntry according to work law regulations. Break lengths are determined based on the total duration of work, with thresholds that dictate the minimum required break length for different work durations.

DEFAULT_MINIMUM_BREAK_LENGTH = 0

MIN_PER_HOUR = 60

WORK_DURATION_THRESHOLDS = [(6, 15), (9, 30), (inf, 45)]

_abc_impl = <_abc._abc_data object>

validate(*entry:* TimeEntry) → *ValidationResult*

Validates the break time specified in a TimeEntry object against legal requirements, based on the total duration of work performed. The method checks if the break time meets or exceeds the required duration specified by work duration thresholds.

Parameters
entry (TimeEntry) – The TimeEntry object containing details of the work duration and break time.

Returns
ValidationResult indicating whether the break time is sufficient as per legal standards.

Return type
ValidationResult

Usage:
If the work duration is 6 hours and above, a break of 30 minutes or more is required, the validation will pass if the break time is equal to or more than the required break time, otherwise it will fail, providing a detailed message.

model.time_entry_validator.holiday_strategy module

class model.time_entry_validator.holiday_strategy.**HolidayStrategy**

Bases: *TimeEntryStrategy*

Strategy for validating TimeEntry objects to ensure they do not coincide with public holidays in Baden-Württemberg, Germany. This strategy utilizes the *holidays* package to check for state-specific public holidays and assesses whether time entries fall on these dates.

_abc_impl = <_abc._abc_data object>

validate(*entry:* TimeEntry) → *ValidationResult*

Validates a given TimeEntry against public holidays in Baden-Württemberg. If the date of the time entry is a public holiday, the validation will fail.

Parameters
entry (TimeEntry) – The TimeEntry object whose date needs to be validated against the public holiday calendar.

Returns
A ValidationResult object that indicates whether the entry date is a public holiday. It returns failure if the date is a public holiday, along with a message stating the name of the holiday. Otherwise, it returns success, indicating that the date is not a public holiday.

Return type
ValidationResult

Example

- If a time entry is made on 01.01.2022 and it's New Year's Day (a public holiday), the validation will return: ValidationResult(ValidationStatus.FAILURE, "Entry date is a public holiday: New Year's Day.")
- If a time entry is made on 01.02.2022, which is not a public holiday, the validation will return: ValidationResult(ValidationStatus.SUCCESS, "Entry date is not a public holiday.")

model.time_entry_validator.time_entry_strategy module

class model.time_entry_validator.time_entry_strategy.**TimeEntryStrategy**

Bases: ABC

Abstract base class for defining validation strategies for TimeEntry objects.

This class provides a framework for implementing various validation strategies as part of a strategy pattern in validating TimeEntry objects. Each subclass should provide a concrete implementation of the validate method that encapsulates the specific validation logic applicable to the TimeEntry being validated.

_abc_impl = <_abc._abc_data object>

abstract validate(*entry:* TimeEntry) → *ValidationResult*

Abstract method to validate a TimeEntry object. Subclasses must implement this method to provide specific validation behavior based on different rules or conditions.

Each validation strategy derived from this class will implement this method to assess a TimeEntry object against specific criteria, determining if the entry meets the required validation standards.

Parameters
entry (TimeEntry) – The TimeEntry object to validate, which contains data such as start and end times, break durations, and potentially other metadata pertinent to the validation logic.

Returns
A ValidationResult object encapsulating the outcome of the validation process. The result includes a status indicating success or failure and, optionally, a message providing details about the validation outcome.

Return type
ValidationResult

Example

- A *BreakLengthStrategy* might return a failure if the breaks are too short for the recorded working hours.
- A *HolidayStrategy* might fail the validation if the work is recorded on a public holiday.

Note: This method is abstract and must be overridden by subclasses; it should not be called directly.

model.time_entry_validator.time_entry_validator module

class model.time_entry_validator.time_entry_validator.**TimeEntryValidator**

Bases: object

Handles the validation of *TimeEntry* objects against a set of dynamically manageable validation rules. This class utilizes a strategy pattern to facilitate the flexible addition and removal of validation strategies, enabling custom validation scenarios for different types of time entries.

validationRules

A list of validation rules that *TimeEntry* objects will be checked against.

Type
list

add_validation_rule(*rule*)

Adds a new validation strategy to the list of rules. Each strategy must be an object that implements a *validate* method capable of processing a *TimeEntry* object.

Parameters
rule (TimeEntryStrategy) – A validation strategy instance, typically derived from the TimeEntryStrategy class, which includes specific validation logic for TimeEntry objects.

remove_validation_rule(*rule*)

Removes a specified validation strategy from the list of rules. This method allows for dynamic adjustments to the validation process, accommodating changes in validation requirements or contexts.

Parameters
rule (TimeEntryStrategy) – The validation strategy instance to remove.

validate_entry(*timeEntry:* TimeEntry)

Validates a *TimeEntry* object against all active validation strategies. This method applies each rule to the *TimeEntry* object and collects the results.

Parameters
timeEntry (TimeEntry) – The *TimeEntry* object to validate, containing all necessary data for the validations.

Returns
A list of *ValidationResult* objects, each representing the outcome of a single validation strategy

applied to the *TimeEntry* object. Each result includes information about whether the validation was successful, along with any relevant messages. :rtype: list[ValidationResult]

model.time_entry_validator.working_time_strategy module

class model.time_entry_validator.working_time_strategy.**WorkingTimeStrategy**

Bases: *TimeEntryStrategy*

Strategy for validating TimeEntry objects to ensure they comply with standard business hours (8 AM to 6 PM) and do not exceed the maximum permitted working hours within a day.

This strategy is particularly useful for organizations that adhere to strict working time regulations and wish to enforce compliance through time entry validations.

BUSINESS_END = datetime.time(18, 0)

BUSINESS_START = datetime.time(8, 0)

FAILURE_WORKING_HOURS = 10

MAX_WORKING_HOURS = 8

_abc_impl = <_abc._abc_data object>

validate(*entry:* TimeEntry) → *ValidationResult*

Validates a single TimeEntry against defined business hours and maximum working hours constraints.

Parameters
entry (TimeEntry) – The TimeEntry object to validate. It should have attributes for start and end times.

Returns
A ValidationResult indicating whether the time entry meets the established business rules. This includes validation against the start and end times for being within business hours, and checking the total working hours against maximum thresholds.

Return type
ValidationResult

Examples

- If a time entry starts at 7:55 AM (which is before 8 AM), this method will return a ValidationResult with a WARNING status indicating the time is outside standard business hours.
- If a time entry represents 9 hours of work, this method will return a ValidationResult with a WARNING status due to exceeding the maximum allowable working hours.
- If a time entry represents 11 hours of work, this method will return a ValidationResult with a FAILURE status due to exceeding the maximum allowable working hours.

Notes

- Time entries within business hours but exceeding 8 hours will trigger a WARNING about long working hours.
- Time entries that exceed 10 hours of work will receive a FAILURE status, as this is beyond the acceptable working time limit.

Module contents

model.time_sheet_validator package

Submodules

model.time_sheet_validator.timesheet_strategy module

class model.time_sheet_validator.timesheet_strategy.**TimesheetStrategy**

Bases: ABC

Abstract base class for defining validation strategies for Timesheet objects. This class serves as a foundation for implementing various validation strategies as part of a strategy pattern, enabling dynamic validation logic tailored to specific requirements of Timesheet objects.

Each subclass of TimesheetStrategy is expected to implement the *validate* method, which encapsulates the validation logic specific to the aspect of the timesheet that needs to be validated, such as adherence to working hours limits, validation of project-specific time allocations, or compliance with labor regulations.

_abc_impl = <_abc._abc_data object>

abstract validate(*timesheet:* Timesheet, *time_entries:* *list*[TimeEntry]) → *ValidationResult*

Abstract method that must be implemented by all subclasses to perform specific validation on a Timesheet object, taking into consideration all related TimeEntry records.

This method evaluates the Timesheet and its associated TimeEntries against predefined validation criteria, which may vary depending on the specific implementation. For example, one strategy might check that the total hours recorded do not exceed legal work limits, while another could verify that time entries fall within approved project scopes.

Parameters

- **timesheet** (Timesheet) – The Timesheet object that is subject to validation. It represents a collection of work records for a specific period.
- **time_entries** (*list*[TimeEntry]) – A list of TimeEntry objects that belong to the timesheet. These are the individual records of work that cumulatively make up the timesheet.

Returns

A ValidationResult object that indicates the outcome of the validation process. This result includes a status (e.g., success, failure, or warning) and a message detailing the reasons for the validation outcome.

Return type

ValidationResult

Examples

- If timesheet hours exceed a regulatory or contractual limit, the validate method might return a ValidationResult with a FAILURE status and an appropriate message.
- If all checks pass, the method would return a ValidationResult with a SUCCESS status.

Note: This method is abstract and should be overridden in each subclass with specific logic appropriate to the particular validation being implemented.

---

model.time_sheet_validator.timesheet_validator module

class model.time_sheet_validator.timesheet_validator.**TimesheetValidator**

Bases: object

Manages the validation of *Timesheet* objects against a set of defined validation rules. This validator uses a strategy pattern to allow for dynamic addition and removal of validation rules.

validationRules

A list of validation rules that *Timesheet* objects will be checked against.

Type
list

addValidationRule(*rule:* TimesheetStrategy)

Adds a new validation rule to the validator.

This method allows for adding custom rules that implement specific validation logic defined in separate classes. Each rule must have a *validate* method that accepts a *Timesheet* object as its parameter.

Parameters
rule (TimesheetStrategy) – The validation strategy to be added, which must implement the TimesheetStrategy interface.

removeValidationRule(*rule:* TimesheetStrategy)

Removes a validation rule from the validator.

This method allows for the dynamic removal of validation rules from the validator, useful for adapting the validation process to different contexts or requirements.

Parameters
rule (TimesheetStrategy) – The validation strategy to be removed, which should currently be part of the validation rules.

validateTimesheet(*timesheet:* Timesheet)

Validates a given *Timesheet* object against all registered validation strategies to assess its compliance with each of the configured rules.

Parameters
timesheet (Timesheet) – The *Timesheet* object to validate.

Returns
A list of *ValidationResult* instances, each representing the outcome from one of the validation strategies applied to the *Timesheet*.

Return type
list[*ValidationResult*]

Note: This method assumes that an external service (*time_entry_service*) is available to fetch time entries related to the timesheet, which are necessary for certain validations. The availability of this service and its proper function must be ensured before validation.

model.time_sheet_validator.weekly_working_hours_strategy module

class model.time_sheet_validator.weekly_working_hours_strategy.**WeeklyHoursStrategy**

Bases: *TimesheetStrategy*

A strategy for validating that the total weekly working hours recorded in a Timesheet do not exceed the maximum allowed hours. This is particularly important for compliance with labor regulations or contractual agreements that limit the amount of work hours per week.

MAX_WEEKLY_HOURS

The maximum number of hours an employee is allowed to work in a week. This is set to 20 hours by default but should be configurable based on organizational policies or labor law requirements.

Type
int

MAX_WEEKLY_HOURS = 20

_abc_impl = <_abc._abc_data object>

validate(*timesheet:* Timesheet, *time_entries:* *list*[TimeEntry]) → *ValidationResult*

Validates the weekly working hours for each week within a Timesheet to ensure that no week's working hours exceed the set maximum. It organizes time entries by week and calculates the total hours worked per week.

Parameters

- **timesheet** (*model.timesheet.Timesheet*) – The Timesheet object associated with the time entries being validated.
- **time_entries** (list[*model.time_entry.TimeEntry*]) – A list of TimeEntry objects that belong to the timesheet, to be aggregated into weekly totals.

Returns
A ValidationResult object that includes a status indicating whether the weekly hours are within the acceptable range, and a message detailing the validation result.

Return type
controller.input_validator.validation_result.ValidationResult

Example

- If a week's total hours are 25, which exceeds the maximum of 20 hours, the validation will return a ValidationResult with a WARNING status and a message detailing the overage.
- If all weeks are within the limit, it returns a ValidationResult with a SUCCESS status.

Module contents

model.user package

Submodules

model.user.admin module

class model.user.admin.**Admin**(*username: str*, *password_hash: str*, *personal_info:* PersonalInfo, *role:* UserRole)

Bases: *User*

to_dict()

Converts the Admin object to a dictionary.

Returns
A dictionary containing all the current attributes of the Admin object.

model.user.contract_information module

class model.user.contract_information.**ContractInfo**(*hourly_wage: float*, *working_hours: int*, *vacation_hours: int*)

Bases: object

static from_dict(*data: dict*)

Creates a ContractInfo instance from a dictionary.

Parameters
data (*dict*) – A dictionary containing the keys hourly_wage, working_hours, and vacation_hours.

Returns
A new instance of ContractInfo.

to_dict()

Converts the ContractInfo object to a dictionary.

Returns
A dictionary representing the contract information.

update_hourly_wage(*new_wage: float*)

Updates the hourly wage.

Parameters
new_wage – The new hourly wage to be set.

update_vacation_hours(*new_vacation_hours: int*)

Updates the remaining vacation hours.

Parameters
new_vacation_hours – The new number of remaining vacation hours.

update_working_hours(*new_hours: int*)

Updates the working hours.

Parameters
new_hours – The new number of working hours per week to be set.

model.user.hiwi module

class model.user.hiwi.**Hiwi**(*username: str*, *password_hash: str*, *personal_info:* PersonalInfo, *supervisor: str*, *contract_info:* ContractInfo)

Bases: *User*

add_timesheet(*timesheet*)

Adds a timesheet entry to the list of timesheets.

Parameters
timesheet – The timesheet to be added.

to_dict()

Converts the Hiwi object to a dictionary.

Returns
A dictionary containing all the current attributes of the Hiwi object.

update_contract_info(*hourly_wage*, *working_hours*, *vacation_hours*)

Updates the contract information for the Hiwi.

Parameters

- **hourly_wage** – The new hourly wage.
- **working_hours** – The new total number of working hours per week.
- **vacation_hours** – The new total number of vacation hours per year.

model.user.personal_information module

class model.user.personal_information.**PersonalInfo**(*first_name: str*, *last_name: str*, *email: str*, *personal_number: str*, *institute_name: str*)

Bases: object

describes the personal information of a user.

classmethod dict_keys()

Returns a list of keys used for the dictionary representation of a PersonalInfo object.

Returns
A list of keys representing the personal information fields.

static from_dict(*data: dict*)

Creates a PersonalInfo instance from a dictionary.

Parameters
data (*dict*) – A dictionary containing keys for first_name, last_name, email, personal_number, and institute_name.

Returns
A new instance of PersonalInfo.

to_dict()

Converts the PersonalInfo object to a dictionary format.

Returns
A dictionary representing the personal information.

model.user.role module

class model.user.role.**UserRole**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

describes the role of a user within the system. There are 4 different user roles: assistant scientist (HiWi), supervisor, secretary and admin.

ADMIN = 'Admin'

HIWI = 'HiWi'

SECRETARY = 'Secretary'

SUPERVISOR = 'Supervisor'

static get_role_by_value(*value: str*)

model.user.secretary module

class model.user.secretary.**Secretary**(*username: str*, *password_hash: str*, *personal_info:* PersonalInfo, *current_timesheet_ids=None*)

Bases: *User*

to_dict()

Converts the Secretary object to a dictionary.

Returns
A dictionary containing all the current attributes of the Secreatry object.

model.user.supervisor module

class model.user.supervisor.**Supervisor**(*username: str*, *password_hash: str*, *personal_info:* PersonalInfo, *hiwis=None*, *currentTimesheetIds=None*)

Bases: *User*

add_hiwi(*hiwi*)

Adds a Hiwi to the Supervisor's list of Hiwis.

Parameters
hiwi – The Hiwi to be added to the list.

remove_hiwi(*hiwi*)

Removes a Hiwi from the Supervisor's list of Hiwis.

Parameters
hiwi – The Hiwi to be removed from the list.

to_dict()

Converts the Supervisor object to a dictionary.

Returns
A dictionary containing all the current attributes of the Supervisor object.

model.user.user module

class model.user.user.**User**(*username: str*, *password_hash: str*, *personal_info:* PersonalInfo, *role:* UserRole)

Bases: object

classmethod dict_keys()

Returns a list of keys used for the dictionary representation of a User object.

Returns
A list of keys representing the user's data fields.

static from_dict(*user_data: dict*)

Creates a User instance from a dictionary.

Parameters
user_data (*dict*) – Dictionary containing user details.

Returns
A new User instance.

is_admin()

Checks if the user's role is 'ADMIN'.

Returns
True if the user is an admin, False otherwise.

to_dict()

Converts the user object to a dictionary format.

Returns
A dictionary representing the user's data.

Module contents

1.5.2 Submodules

1.5.3 model.request_result module

class model.request_result.**RequestResult**(*is_successful*, *message*, *status_code*, *data=None*)

Bases: object

Represents the outcome of a request or operation, encapsulating whether it was successful, along with an associated message, status code, and optional additional data.

This class is typically used to standardize the format of responses across various parts of an application, particularly in APIs or service layers where consistent response formats are beneficial.

is_successful

Indicates whether the request was successful.

Type
bool

message

A descriptive message associated with the result of the request.

Type
str

status_code

The HTTP-like status code that summarizes the outcome of the request.

Type
int

data

Additional data related to the request result.

Type
dict, optional

to_dict()

Converts the RequestResult instance into a dictionary format, which is particularly useful for serialization, especially when sending responses from a web API.

Returns
A dictionary containing keys for success status, message, and optionally additional data.

Return type
dict

Example

- If instantiated with success=True, message="Operation completed", and no data, the dictionary will be: *{'isSuccessful': True, 'message': 'Operation completed'}*
- If there is additional data provided, it will appear in the dictionary as well.

1.5.4 model.time_entry module

class model.time_entry.**TimeEntry**(*timesheet_id: str*, *start_time: datetime*, *end_time: datetime*, *entry_type:* TimeEntryType, *time_entry_id=None*)

Bases: ABC

An abstract base class for time entries that defines common attributes and requires implementation of specific methods that depend on the type of time entry.

This class serves as a template for defining different types of time entries, such as work entries or vacation entries, providing common implementation details and requiring subclasses to implement specific behaviors.

time_entry_id

The unique identifier for the time entry, which may be None if not set.

Type
str

timesheet_id

The unique identifier of the timesheet this entry belongs to.

Type
str

start_time

The start time of the time entry.

Type
datetime

end_time

The end time of the time entry.

Type
datetime

entry_type

The type of time entry, which dictates additional behaviors.

Type
TimeEntryType

_abc_impl = <_abc._abc_data object>

classmethod from_dict(*data: dict*)

Factory method that creates an instance of a TimeEntry subclass from a dictionary.

This method dynamically determines the subclass of TimeEntry to instantiate based on the *entryType* key in the input dictionary.

Parameters
data (*dict*) – A dictionary containing all necessary data to instantiate a specific TimeEntry.

Returns
An instance of a subclass of TimeEntry based on the type provided.

Return type
TimeEntry

Raises
ValueError – If the entry type is unsupported or missing.

abstract get_duration()

Abstract method that calculates the duration of the time entry, which must be implemented by subclasses.

Returns
The duration of the time entry, typically in hours or minutes.

Return type
float

set_id(*time_entry_id*)

Sets or updates the identifier for this time entry.

Parameters
time_entry_id (*str*) – A new identifier to assign to this time entry.

abstract to_dict()

Abstract method that must be overridden in subclasses to convert the specific TimeEntry object to a dictionary format, suitable for serialization or storage.

Returns
A dictionary representation of the time entry including keys for timesheet ID, start and end times, and entry type.

Return type
dict

1.5.5 model.time_entry_type module

class model.time_entry_type.**TimeEntryType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum representing different types of time entries within a timesheet system. This enum facilitates distinguishing between different categories of time entries, such as work-related entries and vacation or leave entries.

WORK_ENTRY

Represents a time entry for regular work.

Type
str

VACATION_ENTRY

Represents a time entry for vacation or leave.

Type
str

VACATION_ENTRY = 'Vacation Entry'

WORK_ENTRY = 'Work Entry'

static get_type_by_value(*value*)

Retrieves a *TimeEntryType* enum member based on its string value. This method is useful for converting string data (e.g., from a database or user input) into the corresponding enum member.

Parameters
value (*str*) – The string representation of the time entry type to look up.

Returns
The corresponding *TimeEntryType* enum member if found, or *None* if no match is found.

Return type
Optional[*TimeEntryType*]

Examples

- *TimeEntryType.get_type_by_value("Work Entry")* will return *TimeEntryType.WORK_ENTRY*.
- *TimeEntryType.get_type_by_value("Nonexistent Type")* will return *None*.

1.5.6 model.timesheet module

class model.timesheet.**Timesheet**(*username: str*, *month: int*, *year: int*, *timesheet_id=None*, *status=TimesheetStatus.NOT_SUBMITTED*, *total_time=0.0*, *overtime=0.0*, *last_signature_change=datetime.datetime(2024, 6, 17, 19, 18, 46, 754968)*, *time_entry_ids=[]*)

Bases: object

Represents a timesheet for an employee (Hiwi), encapsulating all relevant data for a specific month and year, including time entries, total hours, and overtime.

timesheet_id

Unique identifier for the timesheet, typically generated by the database.

Type
ObjectId

username

Username of the Hiwi associated with the timesheet.

Type
str

month

Month for which the timesheet is applicable.

Type
int

year

Year for which the timesheet is applicable.

Type
int

status

Current status of the timesheet, e.g., submitted, not submitted, approved.

Type
TimesheetStatus

total_time

Total hours recorded in the timesheet.

Type
float

overtime

Total overtime hours recorded in the timesheet.

Type
float

last_signature_change

Timestamp of the last signature or approval change.

Type
datetime

time_entry_ids

List of ObjectIds corresponding to the time entries included in the timesheet.

Type
list[ObjectId]

add_time_entry(*time_entry_id*)

Adds a new time entry ObjectId to the timesheet.

Parameters
time_entry_id (*ObjectId*) – The ObjectId of the time entry to add.

static from_dict(*timesheet_dict: dict*)

Factory method that creates a Timesheet object from a dictionary containing its data, typically retrieved from a database.

Parameters
timesheet_dict (*dict*) – A dictionary containing all necessary data keys to instantiate a Timesheet.

Returns
A fully instantiated Timesheet object.

Return type
Timesheet

remove_time_entry(*time_entry_id*)

Removes a time entry ObjectId from the timesheet.

Parameters
time_entry_id (*ObjectId*) – The ObjectId of the time entry to remove.

set_id(*timesheet_id*)

Sets or updates the unique identifier of the timesheet.

Parameters
timesheet_id (*ObjectId*) – The new identifier to assign to this timesheet.

to_dict()

Converts the Timesheet object to a dictionary suitable for serialization, typically for database storage or API responses.

Returns
A dictionary representation of the Timesheet object.

Return type
dict

1.5.7 model.timesheet_status module

class model.timesheet_status.**TimesheetStatus**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum representing the various statuses a timesheet can have within a system, defining the progress and state of a timesheet from creation to completion.

This enum facilitates state management of timesheets, allowing systems to track and control the workflow of timesheet processing, approval, and archival.

NOT_SUBMITTED

Status indicating the timesheet has not yet been submitted for approval.

Type
str

WAITING_FOR_APPROVAL

Status indicating the timesheet has been submitted and is awaiting approval.

Type
str

REVISION

Status indicating the timesheet requires revisions before it can be approved.

Type
str

COMPLETE

Status indicating the timesheet has been approved and is now considered complete.

Type
str

COMPLETE = 'Complete'

NOT_SUBMITTED = 'Not Submitted'

REVISION = 'Revision'

WAITING_FOR_APPROVAL = 'Waiting for Approval'

static get_status_by_value(*value: str*)

Retrieves a *TimesheetStatus* enum member based on its string value. This method is useful for converting string data (e.g., from user input or database values) into the corresponding enum member.

Parameters
value (*str*) – The string representation of the timesheet status to look up.

Returns
The corresponding *TimesheetStatus* enum member if found, or *None* if no match is found.

Return type
Optional[*TimesheetStatus*]

Example

- *TimesheetStatus.get_status_by_value("Not Submitted")* will return *TimesheetStatus.NOT_SUBMITTED*.
- *TimesheetStatus.get_status_by_value("Nonexistent Status")* will return *None*.

1.5.8 model.vacation_entry module

class model.vacation_entry.**VacationEntry**(*timesheet_id: str*, *start_time: datetime*, *end_time: datetime*)

Bases: *TimeEntry*

Represents a vacation or leave entry in a timesheet, inheriting from TimeEntry. This class specifically handles entries that are marked as vacation, applying appropriate validation rules that may differ from those for other types of time entries.

The class also integrates validation strategies specific to vacation entries, such as checks against working during holidays or exceeding allowable vacation time.

_abc_impl = <_abc._abc_data object>

classmethod dict_keys()

Provides a list of keys that are used in the dictionary representation of a VacationEntry object. This method can be used to understand what information is serialized for storage or transmission.

Returns
A list of string keys that represent the attributes of a VacationEntry object.

Return type
list[str]

`get_duration()`

Calculates and returns the total duration of the vacation entry, expressed in hours.

Returns
The total number of hours between the start and end times of the vacation entry.

Return type
float

Example

- If the start_time is at 10 AM and the end_time is at 2 PM on the same day, the duration would be 4.0 hours.

`to_dict()`

Converts the VacationEntry object to a dictionary format, including all base class attributes and any that are specific to the VacationEntry class.

Returns
A dictionary representation of the VacationEntry, suitable for serialization.

Return type
dict

1.5.9 model.work_entry module

`class model.work_entry.WorkEntry(timesheet_id: str, start_time: datetime, end_time: datetime, break_time: float, activity: str, project_name: str, time_entry_id=None)`

Bases: *TimeEntry*

Represents a work entry in the timesheet, capturing detailed information about work periods, activities performed, and breaks taken during the work period.

`break_time`

The duration of the break time in minutes.

Type
float

`activity`

A description of the activity performed during the work entry.

Type
str

`project_name`

The name of the project associated with the work entry.

Type
str

_abc_impl = <_abc._abc_data object>

classmethod dict_keys()

Provides a list of keys that are used in the dictionary representation of a WorkEntry object.

Returns
A list of string keys that represent the attributes of a WorkEntry object.

Return type
list[str]

classmethod from_dict(*data: dict*)

Creates a WorkEntry instance from a dictionary containing data typically retrieved from a database.

Parameters
data (*dict*) – A dictionary containing all necessary data keys to instantiate a WorkEntry.

Returns
A fully instantiated WorkEntry object.

Return type
WorkEntry

get_activity_project_str()

Returns the activity and project name as a string. :return: The activity and project name as a string.

get_duration()

Calculates the duration of the work entry. :return: The duration of the work entry in "hh:mm" format.

to_dict()

Converts the WorkEntry object to a dictionary format, including base class attributes and those specific to WorkEntry.

Returns
A dictionary representation of the WorkEntry, suitable for serialization.

Return type
dict

1.5.10 Module contents

1.6 service package

1.6.1 Subpackages

service.document package

Submodules

service.document.document_generator_strategy module

class service.document.document_generator_strategy.**DocumentGeneratorStrategy**

Bases: ABC

The DocumentGeneratorStrategy interface defines the methods that must be implemented by concrete document generator strategies.

_abc_impl = <_abc._abc_data object>

abstract generate_document(*data: DocumentData*)

Generates a document based on the given data.

Parameters
data – The data to use for generating the document.

Returns
The generated document.

generate_multiple_documents(*data: list[DocumentData]*)

Generates multiple documents based on the given list of data.

Parameters
data – The list of data to use for generating the documents.

Returns
The list of generated documents.

service.document.document_service module

class service.document.document_service.**DocumentService**

Bases: object

The DocumentService class is responsible for generating documents.

_create_zip_from_directory(*output_dir: str*) → BytesIO

Creates a zip file from all PDF files in the given directory.

Parameters
output_dir – The directory containing the PDF files.

Returns
The zip file as a BytesIO stream.

_increment_month(*date*)

Increments the month of the given date. If the month is December,
the year is incremented and the month is set to January.

Parameters
date – The date to increment.

Returns
The date with the month incremented.

gather_document_data(*month: int*, *year: int*, *username: str*) → DocumentData | None

Gathers the data required for generating a document.

Parameters

- **month** – The month for which to gather the data.
- **year** – The year for which to gather the data.
- **username** – The username of the user for which to gather the data.

Returns
The document data.

generate_document(*month: int*, *year: int*, *username: str*)

Generates a document for the given month and year.

Parameters

- **month** – The month for which to generate the document.
- **year** – The year for which to generate the document.
- **username** – The username of the user for which to generate the document.

Returns
The generated document.

generate_document_in_date_range(*start_date: datetime*, *end_date: datetime*, *username: str*)

Generates a document for the given date range.

Parameters

- **start_date** – The start date of the date range.
- **end_date** – The end date of the date range.
- **username** – The username of the user for which to generate the document.

Returns
The generated document.

generate_multiple_documents(*usernames: list[str]*, *month: int*, *year: int*)

Generates a zip file containing PDF documents for a specified list of users, month, and year. Each document is gathered and generated based on user-specific data for the given time period.

Parameters

- **usernames** – The usernames of the users for which to generate the documents.
- **month** – The month for which to generate the documents.
- **year** – The year for which to generate the documents.

Returns
The generated documents.

generate_multiple_documents_by_id(*timesheet_ids: list[str]*)

Generates a zip file containing PDF documents for a specified list of timesheet IDs.

Parameters
timesheet_ids – The timesheet IDs for which to generate the documents.

Returns
The generated documents.

service.document.pdf_generator_strategy module

class service.document.pdf_generator_strategy.**PDFGeneratorStrategy**

Bases: *DocumentGeneratorStrategy*

The PDFGeneratorStrategy class is responsible for generating PDF documents.

SIGNATURE_HEIGHT = 30

SIGNATURE_WIDTH = 300

SIGNATURE_X_POS = 18

SIGNATURE_Y_POS = 637

SUPERVISOR_SIGNATURE_X_POS = 257

SUPERVISOR_SIGNATURE_Y_POS = 637

TEMPLATE_PATH = 'resources/timesheet_template.pdf'

TEMP_DIR = 'resources/temp/'

_abc_impl = <_abc._abc_data object>

_cleanup_temp_files(*files*)

_format_time_entry_data(*time_entry*, *i*)

Formats the time entry data for the PDF document.

Parameters

- **time_entry** – The time entry data to format.
- **i** – The index of the time entry.

Returns

The formatted time entry data.

_get_output_path(*document_data*, *suffix*)

Returns the output path for the generated PDF document.

Parameters

- **document_data** – The document data.
- **suffix** – The suffix to append to the file name.

Returns

The output path for the generated PDF document.

_place_signature(*pdf_path*, *output_path*, *signature_path*, *type_hiwi: bool*)

Places the signature on the PDF document.

Parameters

- **pdf_path** – The path to the PDF document.
- **output_path** – The path to the output PDF document.
- **signature_path** – The path to the signature image.
- **type_hiwi** – The type of the signature (Hiwi or supervisor).

_prepare_data_dict(*document_data*)

Prepares the data dictionary for the PDF document.

Parameters

document_data – The document data.

Returns

The data dictionary for the PDF document.

_temporary_store_signature(*signature_stream*, *path*)

Stores the signature image in a temporary file.

Parameters

- **signature_stream** – The signature image stream.
- **path** – The path to store the signature image.

generate_document(*document_data: DocumentData*)

Generates a PDF document based on the given data.

Parameters
document_data – The data to use for generating the PDF document.

Returns
The generated PDF document.

generate_multiple_documents(*documents: list[DocumentData]*)

Generates multiple PDF documents based on the given list of data.

Parameters
documents – The list of data to use for generating the PDF documents.

Returns
The list of generated PDF documents.

Module contents

1.6.2 Submodules

1.6.3 service.auth_service module

class service.auth_service.**AuthenticationService**

Bases: object

Provides authentication services including login, logout, token generation, and password reset. This service interfaces with the UserRepository to manage user data and security operations.

create_token(*username: str*, *role:* UserRole) → str

Generates a JWT token for a given user with additional claims.

Parameters

- **username** (str) – The username of the user.
- **role** (UserRole) – The role of the user.

Returns
A JWT access token as a string.

Return type
str

get_user_from_token()

Retrieves a user based on the JWT identity from the current request context.

Returns
A User object if the token is valid, otherwise None.

Return type
User or None

`login`(*username*, *password*)

Authenticates a user and returns a JWT token if the credentials are valid.

Parameters

- **username** (*str*) – The username of the user.
- **password** (*str*) – The password of the user.

Returns
A RequestResult including a token if authentication is successful, otherwise an error message.

Return type
RequestResult

`logout`()

Logs out the user by unsetting the JWT cookies.

Returns
A RequestResult indicating the success of the logout operation.

Return type
RequestResult

`reset_password`(*username*, *new_password*)

Resets the password for a given username.

Parameters

- **username** (*str*) – The username for which to reset the password.
- **new_password** (*str*) – The new password.

Returns
A RequestResult indicating the success or failure of the reset process.

Return type
RequestResult

service.auth_service.**check_access**(*roles: [<enum 'UserRole'>] = []*)

Decorator function to check if the user has the required role to access the endpoint.

Parameters
roles (*list[UserRole]*) – A list of UserRole objects that are allowed to access the endpoint.

Returns
The decorator function.

Return type
function

1.6.4 service.file_service module

class service.file_service.**FileService**

Bases: object

Provides service-layer functionality to handle file-related operations, such as uploading, updating, retrieving, and deleting files. This service works with the FileRepository to interact with the model-data layer, ensuring files are managed according to defined business rules.

MAX_FILE_SIZE

The maximum allowed file size for uploads (20 MB).

Type
int

ALLOWED_EXTENSIONS

A set of allowed file extensions for uploads.

Type
set

ALLOWED_EXTENSIONS = {'gif', 'jpeg', 'jpg', 'png'}

MAX_FILE_SIZE = 20971520

_allowed_file(*filename: str*) → bool

Checks if the file's extension is among the allowed types.

Parameters
filename (*str*) – The name of the file to check.

Returns
True if the file's extension is allowed, False otherwise.

Return type
bool

delete_image(*username: str*, *file_type:* FileType) → *RequestResult*

Deletes an image associated with a specific username and file type.

Parameters

- **username** (*str*) – The username linked to the image.
- **file_type** (FileType) – The type of the file to delete.

Returns
Indicates the success or failure of the delete operation.

Return type
RequestResult

does_file_exist(*username: str*, *file_type:* FileType) → bool

Verifies if a particular file exists for a given username and specified file type.

Parameters

- **username** (*str*) – The username to check.
- **file_type** (FileType) – The type of file to verify.

Returns
True if the file exists, otherwise False.

Return type
bool

get_image(*username: str*, *file_type:* FileType)

Retrieves an image based on the specified username and file type.

Parameters

- **username** (*str*) – The username associated with the image.
- **file_type** (FileType) – The type of file being requested.

Returns
The image object if found, otherwise None.

Return type
file-like object or None

upload_image(*file*, *username: str*, *file_type:* FileType) → *RequestResult*

Handles the uploading or updating of an image file after validating its size and type.

Parameters

- **file** – The file object to be uploaded, typically a Flask *request.files* object.
- **username** (*str*) – The username associated with the file.
- **file_type** (FileType) – The type of file, determined by an enumeration.

Returns
Indicates the success or failure of the upload operation.

Return type
RequestResult

1.6.5 service.time_entry_service module

class service.time_entry_service.**TimeEntryService**

Bases: object

Manages time entry operations, interfacing with the TimeEntryRepository for data storage and retrieval, the TimesheetService for handling related timesheet operations, and a validator for time entry data validation.

_add_time_entry(*entry_data: dict*, *entry_type:* TimeEntryType)

General method to handle addition of work or vacation time entries.

Parameters

- **entry_data** (*dict*) – Time entry data.
- **entry_type** (TimeEntryType) – The type of the entry (e.g., WorkEntry or VacationEntry).

Returns
A RequestResult object containing the outcome.

Return type
RequestResult

add_vacation_entry(*entry_data: dict*) → *RequestResult*

Adds a new vacation time entry based on the provided entry data.

Parameters
entry_data (*dict*) – A dictionary containing vacation time entry attributes.

Returns
A RequestResult object containing the result of the add operation.

Return type
RequestResult

create_work_entry(*entry_data: dict*) → *RequestResult*

Creates a new work time entry in the system based on the provided entry data.

Parameters
entry_data (*dict*) – A dictionary containing time entry attributes necessary for creating a new time entry.

Returns
A RequestResult object containing the result of the create operation.

Return type
RequestResult

delete_time_entry(*entry_id: str*) → *RequestResult*

Deletes a time entry from the system identified by its ID.

Parameters
entry_id (*str*) – The ID of the time entry to be deleted.

Returns
A RequestResult object containing the result of the delete operation.

Return type
RequestResult

get_entries_of_timesheet(*timesheet_id: str*) → list[*TimeEntry*]

Retrieves a list of time entries associated with a specific timesheet ID and converts them into TimeEntry objects, considering the specific type of each entry.

Parameters
timesheet_id (*str*) – The ID of the timesheet for which to retrieve entries.

Returns
A list of TimeEntry model instances representing all time entries for the specified timesheet.

Return type
list[*TimeEntry*]

update_time_entry(*entry_id: str*, *update_data: dict*) → *RequestResult*

Updates an existing time entry in the system with the provided update data after validating the data.

Parameters

- **entry_id** (*str*) – The ID of the time entry to update.
- **update_data** (*dict*) – A dictionary with time entry attributes that should be updated.

Returns
RequestResult object containing the result of the update operation.

Return type
RequestResult

1.6.6 service.timesheet_service module

class service.timesheet_service.**TimesheetService**

Bases: object

Provides service-layer functionality to handle timesheet-related operations, such as creating, updating, retrieving, and managing timesheets and their statuses. This service works with the TimesheetRepository to interact with the model-data layer.

add_time_entry_to_timesheet(*timesheet_id: str*, *time_entry_id: str*)

Adds a time entry to a timesheet_data.

Parameters

- **timesheet_id** – The ID of the timesheet_data
- **time_entry_id** – The ID of the time entry

Returns
The result of the add operation

approve_timesheet(*timesheet_id: str*)

Method used by the supervisor to sign a timesheet. This sets the status to approved.

Parameters
timesheet_id (*str*) – The ID of the timesheet to approve.

Returns
The result of the approval operation.

Return type
RequestResult

create_timesheet(*username: str*, *month: int*, *year: int*)

Creates a new timesheet.

Parameters

- **username** – The username of the Hiwi
- **month** – The month of the timesheet
- **year** – The year of the timesheet

Returns
The result of the create operation

delete_time_entry_from_timesheet(*timesheet_id: str*, *time_entry_id: str*)

Removes a time entry from a timesheet.

Parameters

- **timesheet_id** – The ID of the timesheet
- **time_entry_id** – The ID of the time entry

Returns
The result of the remove operation

ensure_timesheet_exists(*username: str*, *month: int*, *year: int*)

Ensures that a timesheet exists for the given username, month, and year. If the timesheet does not exist, it will be created.

Parameters

- **username** (*str*) – The username of the Hiwi.
- **month** (*int*) – The month of the timesheet.
- **year** (*int*) – The year of the timesheet.

Returns
A RequestResult object containing the result of the ensure operation.

Return type
RequestResult

get_current_timesheet(*username: str*)

Retrieves the current timesheet for a given username.

Parameters
username – The username of the Hiwi

Returns
The timesheet object

get_timesheet(*username: str*, *month: int*, *year: int*)

Retrieves a timesheet by username, month, and year.

Parameters

- **username** – The username of the Hiwi
- **month** – The month of the timesheet
- **year** – The year of the timesheet

Returns
The timesheet object

get_timesheet_by_id(*timesheet_id: str*)

Retrieves a timesheet by its ID.

Parameters
timesheet_id – The ID of the timesheet

Returns
The timesheet object

get_timesheet_id(*username: str*, *month: int*, *year: int*)

Retrieves the ID of a timesheet.

Parameters

- **username** – The username of the Hiwi
- **month** – The month of the timesheet
- **year** – The year of the timesheet

Returns
The ID of the timesheet

`get_timesheets_by_username`(*username: str*)

Retrieves all timesheets for a given username.

Parameters
username – The username of the Hiwi

Returns
A list of timesheet objects

`get_timesheets_by_username_status`(*username: str*, *status:* TimesheetStatus)

Retrieves all timesheets for a given username and status.

Parameters

- **username** – The username of the Hiwi
- **status** – The status of the timesheets

Returns
A list of timesheet objects

`request_change`(*timesheet_id: str*)

Method used by the supervisor to request changes to a timesheet. This sets the status to change requested.

Parameters
timesheet_id (*str*) – The ID of the timesheet to request changes for.

Returns
The result of the request change operation.

Return type
RequestResult

`set_timesheet_status`(*timesheet_id: str*, *status:* TimesheetStatus)

Sets the status of a timesheet.

Parameters

- **timesheet_id** – The ID of the timesheet to update
- **status** – The new status of the timesheet

Returns
The result of the status update operation

`sign_timesheet`(*timesheet_id: str*)

Method used by the Hiwi to sign his timesheet.

Parameters
timesheet_id (*str*) – The ID of the timesheet to sign.

Returns
The result of the sign operation.

Return type
RequestResult

1.6.7 service.user_service module

class service.user_service.**UserService**

Bases: object

Provides service-layer functionality to handle user-related operations, such as creating, updating, retrieving, and deleting users. This service works with the UserRepository to interact with the model-data layer, ensuring users are managed according to defined business rules.

_recursive_update(*original: dict*, *updates: dict*, *exclude_keys=None*) → dict

Recursively update a dictionary with another dictionary, excluding specified keys.

Parameters

- **original** – The original dictionary to update.
- **updates** – The dictionary containing updates to apply.
- **exclude_keys** – A set or list of keys to exclude from the updates.

Returns
The updated dictionary.

create_user(*user_data*) → *RequestResult*

Creates a new user in the system based on the provided user data.

Parameters
user_data – A dictionary containing user attributes necessary for creating a new user.

Returns
A RequestResult object containing the result of the create operation.

delete_user(*username: str*)

Deletes a user from the system identified by their username.

Parameters
username – The username of the user to be deleted.

Returns
A RequestResult object containing the result of the delete operation.

get_profile(*username: str*) → *User*

Retrieves the profile of a specific user identified by their username.

Parameters
username (str) – The username of the user whose profile is being requested.

Returns
A User model instance representing the user's profile.

Return type
User

get_users() → list[*User*]

Retrieves a list of all users in the system.

Returns
A list of User model instances representing all users in the system.

Return type
list[*User*]

get_users_by_role(*role: str*) → list[*User*]

Retrieves a list of users in the system filtered by a specific role.

Parameters
role (*str*) – The role to filter users by.

Returns
A list of User model instances that match the specified role.

Return type
list[*User*]

update_user(*user_data: dict*)

Updates an existing user in the system with the provided user data.

Parameters
user_data – A dictionary with user attributes that should be updated.

Returns
RequestResult object containing the result of the update operation.

1.6.8 Module contents

1.7 utils package

1.7.1 Submodules

1.7.2 utils.object_utils module

class utils.object_utils.**ObjectUtils**

Bases: object

Utility class providing methods for handling and converting BSON ObjectId instances within data structures.

static convert_objectids_to_strings(*data*)

Recursively converts all fields within a dictionary or list that are instances of bson.ObjectId to strings.

This method is particularly useful for preparing data for JSON serialization, where ObjectId instances need to be converted to string format.

Parameters
data (*dict, list, or ObjectId*) – The data structure to convert. It can be a dictionary, list, or a single ObjectId instance.

Returns
The data structure with all ObjectId fields converted to strings.

Return type
dict, list, or str

1.7.3 utils.security_utils module

class utils.security_utils.**SecurityUtils**

Bases: object

Provides static methods for common security operations such as password hashing and password verification using bcrypt.

static check_password(*password: str*, *hashed_password: str*) → bool

Checks if a plain text password matches a hashed password.

This method verifies whether the provided plain text password, when hashed, matches the given hashed password. This is useful for authentication purposes.

Parameters

- **password** (*str*) – The plain text password to check.
- **hashed_password** (*str*) – The hashed password to compare against.

Returns
True if the password matches the hashed password, False otherwise.

Return type
bool

static hash_password(*password: str*) → str

Hashes a password using bcrypt.

This method takes a plain text password and returns its hashed version using bcrypt, which is a password hashing function designed to be computationally expensive to resist brute-force attacks.

Parameters
password (*str*) – The plain text password to hash.

Returns
The hashed password.

Return type
str

PYTHON MODULE INDEX

U

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

N

O

P

R

S

T

U

V

W

Y