Designbeschreibung

Erstellt von:	**Überprüft von:**
Miguel Friesen, Stephan Braun, Tom-Liam Müller, Maurice Nowotni, Noah Schwenk	Noah Schwenk, Miguel Friesen

Version	Effektiv ab	Beschreibung / Änderungen
1.0	06.04.2026	Initialer Entwurf
1.1	07.04.2026	Etablierung der Controller-Grenzen ([Authorize]). Bereinigung der REST-API-Spezifikation (Kapitel 4.5)
1.2	08.04.2026	Anpassung der statischen UML-Klassendiagramme
1.3	08.04.2026	Überarbeitung aktueller Stand
1.4	10.04.2026	Integration des Lebenszyklus (Zustandsautomat).
1.5	12.04.2026	Erstellung algorithmischen Kontrollflusses
2.0	12.04.2026	Finale Überarbeitung
3.0	16.04.2026	Überarbeitung auf Grund der Anmerkungen aus dem Review
4.0	27.04.2026	Aktualisierung des Klassendiagramms
5.0	03.05.2026	Aktualisierung nach User Stories Woche 2
6.0	10.05.2026	Aktualisierung nach User Stories Woche 3
7.0	17.05.2026	Aktualisierung nach User Stories Woche 4

Inhalt

1. Allgemeines 2
2. Produktübersicht 3
2.1 Konfiguration des individuellen Arbeitsprofils und Mandantenfähigkeit 4
2.2 Granulare Aufgabenparametrisierung und Flow-Schutz 4
2.3 Algorithmische Arbeitsplangenerierung und Kognitives Routing 4
2.4 Dynamische Resilienz: Aufgabenfortschritt und Rescheduling 5
2.5 Transparente Planungslogik und Clash Management 5
3. Grundsätzliche Struktur- und Entwurfsentscheidungen für die Anwendung (Makro-Architektur) 5
3.1 Präsentationsschicht (Frontend: React.js) 7
3.2 Business-Logik & REST-API (Backend: C# .NET Core) 8
3.3 Persistenzschicht (Datenbank: Cloud Firestore) 9
4. Struktur- und Entwurfsentscheidungen der einzelnen Komponenten 9
4.1. Statische Systemarchitektur (3-Schichten-Modell) 10
4.2. Entwurfsmuster (Design Patterns) und OOD-Spezifika 11
4.3. Datenmodellierung und Persistenzstruktur (Cloud Firestore) 14
Struktur des Datenbank-Schemas 14
Root-Kollektionen 14
Die 4 Sub-Kollektionen von USERS 14
Referenz-Logik und Verknüpfungen 14
4.4 Dynamisches Verhaltensmodell 14
4.4.1. Algorithmischer Kontrollfluss (Aktivitätsdiagramm) 14
4.4.2. Intersystem-Datenaustausch (Sequenzdiagramm) 14
4.4.3. Objekt-Lebenszyklus der Aufgabe (Zustandsautomat) 14
4.4.4. Mandantenfähigkeit und Einladungsprozess 14
4.5. API-Spezifikation (REST-Schnittstellen) 14
Globale HTTP-Statuscodes 14
User & Identity (/api/v1/user) 14
Arbeitsprofil & Settings (/api/v1/user/work-profile) 14
Aufgaben (Tasks) (/api/v1/user/calendar) 14
Organisations Management (/api/v1/organizations) 14
4.6. Architektonische Nachverfolgbarkeit (Traceability-Matrix) 14

Version 7.0
17.05.2026

1. Allgemeines

Das vorliegende Dokument beschreibt den Designentwurf und die Systemarchitektur für das geplante Online Aufgaben-Planungssystem unseres Teams „TaskFlow Engineering". Es definiert die Makro-Architektur, die verwendeten Technologien sowie die zugrundeliegenden statischen und dynamischen Modelle für die anstehende Implementierungsphase. Ziel dieses Dokuments ist es, eine verbindliche technische Blaupause zu schaffen, anhand derer eine programmiererfahrene Person die Struktur- und Entwurfsprinzipien des Systems nachvollziehen kann.

Einsatzumfeld und fundamentale Systemlogik:

Die Applikation ist architektonisch primär auf das individuelle Selbstmanagement von Aufgaben fokussiert. Die wichtigste technische Rahmenbedingung (Constraint) für den zugrundeliegenden Planungsalgorithmus ist die Limitierung der maximalen täglichen Kapazität für hochkonzentrierte Wissensarbeit (definiert auf 2 bis 4 Stunden pro Tag).

Das System setzt dieses Kapazitätslimit als harte algorithmische Restriktion um. Die Business-Logik (Planungs-Engine) berücksichtigt bei der Zuweisung von Aufgaben (Scheduling) zwingend den deklarierten mentalen Energiebedarf sowie die minimale Splitting-Dauer (Chunking). Dies dient der Minimierung von ineffizientem Task-Switching (Fragmentierung der Arbeitszeit) und verhindert algorithmisch eine Überplanung des Nutzers an einem Arbeitstag.

Architektonische Ausschlusskriterien (Anti-Patterns):

Aus dem funktionalen Fokus auf das individuelle Kapazitätsmanagement ergeben sich klare Systemgrenzen. Folgende Konzepte sind architektonisch explizit ausgeschlossen und werden im Datenmodell sowie in der API-Spezifikation nicht abgebildet:

- **Team-Planung und aggregierte Ressourcenplanung:** Das System verteilt Aufgaben nicht top-down in Teams, sondern fokussiert sich rein auf die Organisation der Eigenarbeit.
- **Vorgesetzten-Dashboards (Supervisor-Views):** Organisierende und Administratoren erhalten architektonisch keinerlei Einsicht in die persönlichen Aufgaben, Arbeitspläne oder Kalender der Nutzenden.
- **Zeiterfassung (Time-Tracking):** Die Architektur ist als reines Planungstool konzipiert. Module zur aktiven Leistungskontrolle oder minutengenauen Arbeitszeiterfassung sind technisch und fachlich ausgeschlossen.

2. Produktübersicht

Das vorliegende intelligente Aufgaben-Planungssystem ist eine exklusiv für Desktop-Umgebungen konzipierte Webanwendung zur automatisierten Termin- und Aufgabenplanung. Die Kernkomponente bildet ein deterministischer Planungsalgorithmus, der das Problem der Ressourcenplanung unter strikten kognitiven und zeitlichen Nebenbedingungen (Constraints) löst . Die Architektur verzichtet bewusst auf aggregierte Team-Planungs-Features und fokussiert sich auf die isolierte Optimierung des individuellen Arbeitsplans. Die maximale kognitive Tageskapazität für tiefe Wissensarbeit (konfigurierbar auf 2 bis 4 Stunden) fungiert dabei als harte algorithmische Systemgrenze. Die Funktionalität gliedert sich in folgende technische Kernbereiche:

2.1 Konfiguration des individuellen Arbeitsprofils und Mandantenfähigkeit

Das informationstechnische Fundament der Planungs-Engine bildet das individuelle Arbeitsprofil. Der Nutzende definiert hier initial seine zeitlichen und kognitiven Rahmenbedingungen. Dazu gehören die Definition verfügbarer Arbeitszeiten, die harte Limitierung der maximalen Dauer zusammenhängender Arbeitsphasen (das 2-4-Stunden-Limit) sowie zwingend einzuhaltende Pausenzeiten. Zudem ist das System mandantenfähig konzipiert: Es differenziert strikt zwischen reinen Mitarbeiter-Accounts, die nur für eine spezifische Organisation planen, und organisationsübergreifenden Freelancer-Accounts. Freelancer können feste, überschneidungsfreie Zeitslots exklusiv an bestimmte, verknüpfte Organisationen binden oder sogenannte "Flex-Blöcke" definieren, in denen der Algorithmus organisationsübergreifend optimieren darf.

2.2 Granulare Aufgabenparametrisierung und Flow-Schutz

Die Eingabe einer Aufgabe (Task) verlangt dem Nutzenden eine präzise Parametrisierung ab, die weit über Standard-Metadaten hinausgeht. Zwingende Parameter sind neben Titel, Beschreibung, geschätzter Bearbeitungsdauer, Priorität und einer harter Deadline vor allem der kognitive Energiebedarf (Schwierigkeitsgrad) sowie logische Aufgaben-Abhängigkeiten. Ein zentrales architektonisches Merkmal ist die Implementierung des sogenannten "Flow-Schutzes": Das System erzwingt die Definition einer minimalen und maximalen Splitting-Dauer (Chunk-Größe) pro Aufgabe. Dies definiert die absolute zeitliche Unter- und Obergrenze, in die die Engine eine Aufgabe zerteilen darf. Dadurch wird verhindert, dass der Algorithmus verbleibende Restzeiten im Kalender mit kognitiv ineffizienten Kleinstblöcken auffüllt, was den mentalen Rüstaufwand minimiert.

Zudem können Termine als "statische Blocker" (z.B. ein Arzttermin) deklariert werden, wodurch sie unverrückbare Ereignisse im Kalender darstellen, welche reine Zeit, aber keine kognitive Energie verbrauchen.

2.3 Algorithmische Arbeitsplangenerierung und Kognitives Routing

Auf Basis der erfassten Aufgaben und des Arbeitsprofils generiert das System vollautomatisch einen überschneidungsfreien, nachhaltigen Arbeitsplan. Die Engine wertet hierzu einen strikten Entscheidungsbaum aus: Bevor ein Task-Chunk in einen Zeitslot platziert wird, verifiziert der Algorithmus zwingend über einen Guard, ob das kognitive Tageslimit erreicht ist. Ist dies der Fall, wird die Einplanung für den aktuellen Tag hart abgebrochen und auf den nächsten Arbeitstag verschoben. Zur aktiven Vermeidung von mentaler Erschöpfung durch falsches Energiemanagement prüft die Engine zudem nach jeder Einplanung, ob die vorherige Aufgabe einen hohen kognitiven Energiebedarf besaß. Ist dies der Fall, erzwingt die Logik die Einplanung einer leichten Aufgabe zur kognitiven Entlastung. Diese algorithmische Weichenstellung garantiert einen ausgewogenen Arbeitsrhythmus.

2.4 Dynamische Resilienz: Aufgabenfortschritt und Rescheduling

Das System operiert mit einem planbasierten Soll-Fortschritt und verzichtet bewusst auf eine kleinteilige Zeiterfassung ("Stempeln"). Pläne sind im TaskFlow Engineering niemals starr. Reicht die geplante Zeit für eine Aufgabe in der Realität nicht aus, kann der Nutzende die geschätzte Bearbeitungsdauer nachträglich manuell verlängern. Markiert der Nutzende eine Aufgabe als erledigt oder greift er manuell verschiebend in den Kalender ein, wandelt das System die betroffenen Aufgaben von dynamisch geplant zu statisch fixiert um und erzwingt ein sofortiges, systemweites Rescheduling (Neuberechnung) der verbleibenden kognitiven Kapazitäten für alle restlichen, noch offenen Aufgaben.

2.5 Transparente Planungslogik und Clash Management

Das System überwacht die Auslastung der kognitiven Ressourcen strikt. Reichen die verfügbare Netto-Arbeitszeit oder die kognitive Kapazität mathematisch nicht aus, um alle Aufgaben vor ihrer Hard-Deadline abzuschließen, verweigert das System die Generierung eines fehlerhaften, überlastenden Plans. Es kommt zu einem sogenannten "Clash". In diesem Fall bricht der Algorithmus ab und das System eskaliert den Konflikt durch das Clash Management. Dem Nutzenden wird durch eine transparente Fehlermeldung dargelegt, dass das Limit überschritten wurde. Er wird gezwungen, aktiv Deadlines zu verschieben, Prioritäten zu ändern oder Arbeitszeiten auszuweiten, bevor das System einen neuen, validen Arbeitsplan freigibt.

3. Grundsätzliche Struktur- und Entwurfsentscheidungen für die Anwendung (Makro-Architektur)

Das intelligente Aufgaben-Planungssystem ist als verteilte, webbasierte Client-Server-Architektur konzipiert. Die zentrale Leitlinie des Entwurfs ist die strikte logische und physische Trennung zwischen der Präsentationsschicht (Frontend), der rechenintensiven Business-Logik (Backend-Planungs-Engine) und der Persistenzschicht (Datenbank).

Diese „Separation of Concerns“ ist kein Selbstzweck, sondern eine Reaktion auf die fachlichen Anforderungen: Da der Planungsalgorithmus eine hohe Rechenlast erzeugt, muss dieser von der Benutzeroberfläche entkoppelt sein, um die Bedienbarkeit (Responsivität) des Frontends jederzeit zu gewährleisten.

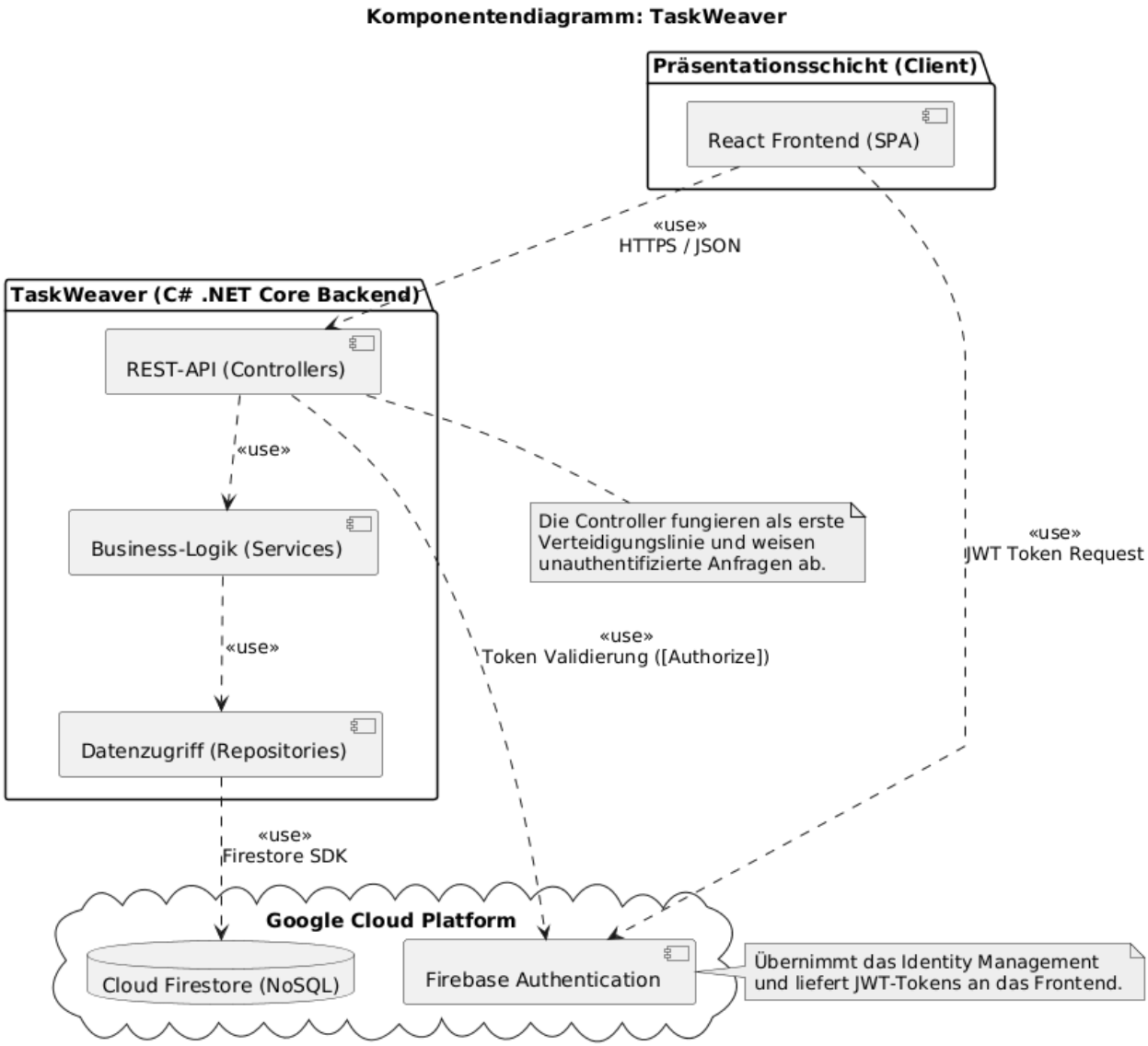


Abbildung 1: Komponentendiagramm: Makro-Architektur

Das Komponentendiagramm ist essenziell, um die physische und logische Kapselung (Separation of Concerns) des verteilten Systems formal zu spezifizieren. Es belegt visuell die Etablierung von Vertrauensgrenzen, wie die Delegierung der Authentifizierung und die strikte serverseitige Token-Validierung, und rechtfertigt die statuslose Architektur für eine zukünftige, isolierte Erweiterbarkeit der Planungs-Engine.

3.1 Präsentationsschicht (Frontend: React.js)

Die Benutzungsschnittstelle wird als Single Page Application (SPA) auf Basis von React.js realisiert. Die fundamentale Architekturentscheidung für das Frontend ist das "Dumb-Client"-Prinzip: Das Frontend enthält **keinerlei eigene Planungslogik**. Es fungiert ausschließlich als Visualisierungs- und Interaktionsschicht. Aufgaben des Frontends sind:

- Die reaktive Darstellung des Kalenders und der dynamischen/statischen Aufgaben.
- Die Visualisierung des Soll-Fortschritts der Aufgaben.
- Die transparente Darstellung des Clash Managements: Werden Aufgaben wegen kognitiver Überlastung durch das Backend abgewiesen, rendert das Frontend die entsprechende Fehlermeldung für den Nutzenden.

3.2 Business-Logik & REST-API (Backend: C# .NET Core)

Das Herzstück des Systems ist das in C# .NET Core implementierte Backend. Es kapselt die gesamte algorithmische Intelligenz und stellt dem Frontend eine statuslose REST-API (Representational State Transfer) über HTTP/HTTPS zur Verfügung.

- **API-Design:** Die Kommunikation zwischen Client und Server erfolgt ausschließlich über streng typisierte DTOs (Data Transfer Objects) im JSON-Format. Ein zentraler Endpunkt der Architektur ist beispielsweise der POST-Request POST /api/v1/user/calendar/task, über welchen neue Aufgaben an die Planungs-Engine übermittelt werden.
- **Sicherheitsarchitektur:** Die API-Endpunkte sind restriktiv abgesichert. Die Authentifizierung wird systemübergreifend an ein externes Identity Management (Firebase Auth) delegiert. Jeder API-Call erfordert ein valides Bearer-Token, wodurch sichergestellt wird, dass Nutzende ausschließlich ihre eigenen kognitiven Profile und Pläne manipulieren können.
- **Architekturmuster:** Innerhalb des C#-Backends wird das Controller-Service-Repository-Pattern angewendet. Die Controller (z.B. der PlanningController) nehmen die HTTP-Requests entgegen und delegieren die Business-Logik an die Services (PlanningService). Die algorithmische Auswertung des kognitiven Energiebedarfs ist über das Strategy Pattern (IPlanningStrategy) abstrahiert. Dies ermöglicht es, die Berechnungslogik zukünftig durch erweiterte Algorithmen oder KI-Modelle auszutauschen, ohne die Kernarchitektur anzupassen.

- **Authentifizierung und Autorisierung:** Die Absicherung der REST-API erfolgt über Firebase Authentication. Das Backend validiert die eingehenden JSON Web Tokens (JWT) über die ASP.NET Core Middleware-Pipeline und extrahiert die Nutzeridentität (FirebaseUid) ausschließlich serverseitig aus den verifizierten Token-Claims. Um *Insecure Direct Object References (IDOR)* dabei vollständig auszuschließen, nutzt das System einen zweistufigen Autorisierungsansatz. Bei individuellen Nutzerdaten (wie Tasks oder Blockern) greift die strikte architektonische Isolation des NoSQL-Schemas, da Abfragen zwingend auf den Pfad der authentifizierten Nutzer-ID beschränkt sind. Bei Zugriffen auf organisationsweite Daten (Company-Management) validiert die Service-Schicht hingegen zusätzlich explizit in der Datenbank, ob der authentifizierte Nutzer in der angefragten Organisation referenziert ist und die für den Endpunkt erforderliche Rolle besitzt.

3.3 Persistenzschicht (Datenbank: Cloud Firestore)

Die Datenhaltung erfolgt asynchron in einer dokumentenorientierten NoSQL-Datenbank (Cloud Firestore).

- **Schema-Flexibilität:** Die Entscheidung gegen eine klassische relationale SQL-Datenbank (wie PostgreSQL) begründet sich in der hohen Flexibilität, die für die granulare Parametrisierung von Aufgaben (inklusive dynamischer Chunk-Größen und kognitiver Energiebedarfe) benötigt wird.
- **Asynchrone I/O-Operationen:** Da das System bei manuellen Planänderungen durch den Nutzenden (z.B. "Manuelles Verschieben im Kalender") ein sofortiges, systemweites Rescheduling erzwingt, müssen Datenbankoperationen extrem performant sein. Der Datenzugriff ist im Backend über das Repository-Pattern abstrahiert und erfolgt zwingend asynchron (z.B. SavePlanAsync), um den Haupt-Thread der Planungs-Engine nicht zu blockieren.

4. Struktur- und Entwurfsentscheidungen der einzelnen Komponenten

Dieses Kapitel spezifiziert die innere Struktur der Anwendungskomponenten. Um eine hohe Code-Qualität, Wartbarkeit und eine leichte Einarbeitung neuer Entwickler zu gewährleisten, folgen die Entwurfsentscheidungen etablierten Architekturkonzepten. Um den Anforderungen des Objektorientierten Designs (OOD) gerecht zu werden, wurde das konzeptionelle Analysemodell (OOA) in ein detailliertes Implementierungsmodell überführt. Das nachfolgende statische Modell beschreibt das System aus einer strukturellen, zeitunabhängigen Perspektive.

4.1. Statische Systemarchitektur (3-Schichten-Modell)

Das C# .NET-Backend bildet das Herzstück der Anwendung und ist konsequent nach der 3-Schichten-Architektur strukturiert. Diese strikte Trennung der Zuständigkeiten (Separation of Concerns) teilt das Backend in folgende statische Ebenen auf, wie das nachfolgende Paketdiagramm belegt:

- **Controller-Schicht (Präsentation für APIs):** Die Controller bilden die statischen Eintrittspunkte der Applikation. Ihre Zuständigkeit umfasst das HTTP-Routing, die JSON-Serialisierung sowie – als wichtigste architektonische Grenze – die zwingende Zugriffsvalidierung. Durch das [Authorize]-Attribut fungieren die Controller als erste Verteidigungslinie und weisen unauthentifizierte Anfragen strikt ab (HTTP 401), bevor die dahinterliegende Business-Logik (Service-Schicht) überhaupt erreicht wird. Sie enthalten explizit keinerlei Geschäftslogik, sondern delegieren eingehende Anfragen an die zuständigen Services.
- **Service-Schicht (Geschäftslogik):** Hier ist die zentrale Domänenlogik verortet, insbesondere die Engine zur algorithmischen Arbeitsplangenerierung (/F40/).
- **Repository-Schicht (Datenzugriff):** Diese Schicht kapselt die gesamte Kommunikation mit der Cloud-Datenbank. Sollte in späteren Projektphasen ein Wechsel der Datenbanktechnologie erforderlich sein, muss lediglich diese isolierte Schicht ausgetauscht werden, während Controller und Services unberührt bleiben.

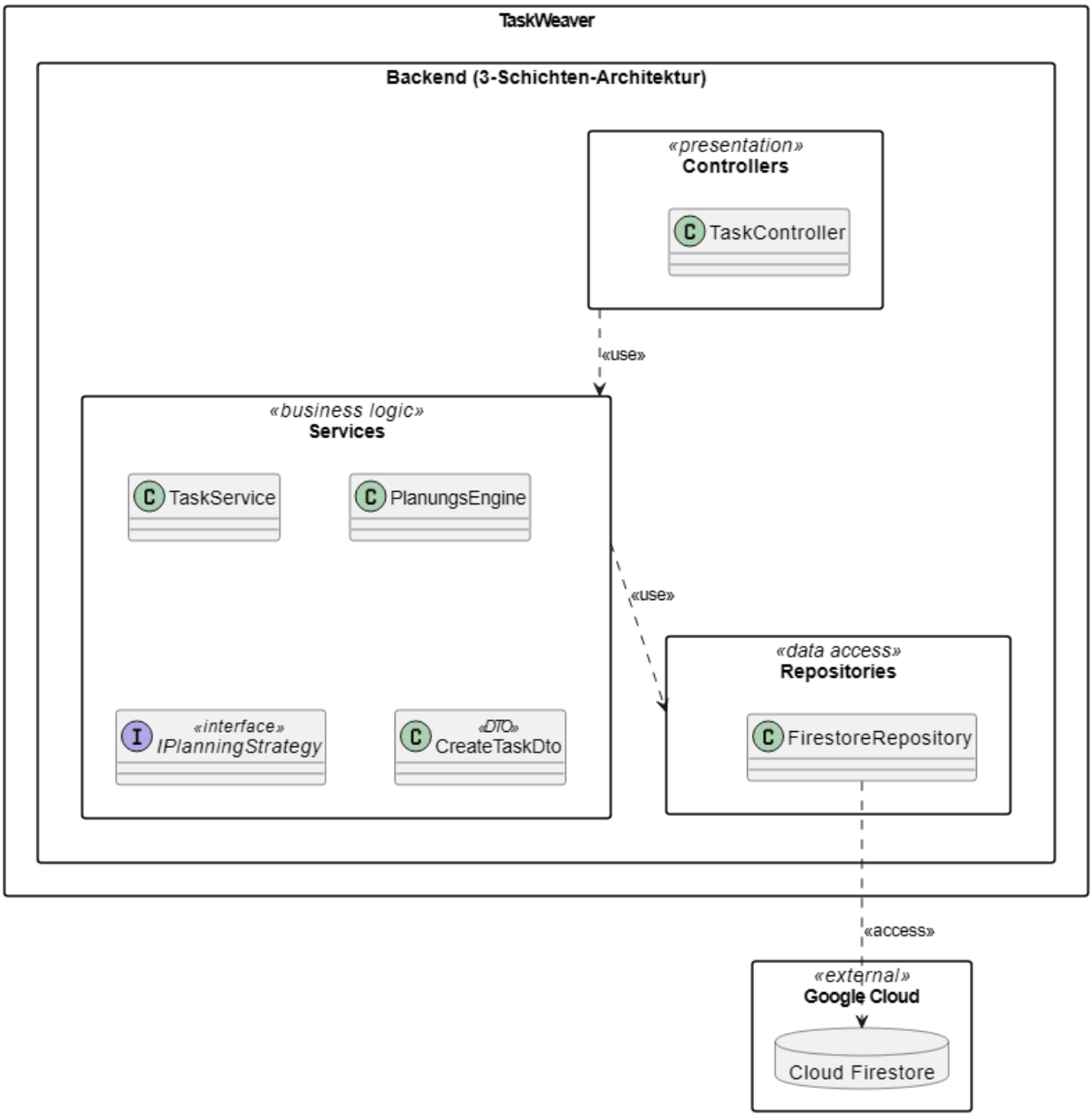


Abbildung 2: Packet-Diagramm statische Systemarchitektur

Dieses Diagramm visualisiert die "Strict Layering Architecture" des Backends, bei der Zugriffe ausschließlich von oben nach unten (Controller -> Service -> Repository) erfolgen. Dies rechtfertigt die Entwurfsentscheidung zur Austauschbarkeit: Die Business-Logik ist durch Interfaces vollständig vom spezifischen Datenzugriff entkoppelt und ermöglicht somit isoliertes Testing (Mocking).

4.2. Entwurfsmuster (Design Patterns) und OOD-Spezifika

Um die Softwarearchitektur robust und erweiterbar zu gestalten, implementieren die statischen Klassen im Detailentwurf folgende Entwurfsmuster und Prinzipien:

- **Strategy Pattern:** Um die Architekturanforderung der Austauschbarkeit zu erfüllen, ist die Logik zur Arbeitsplangenerierung nicht hart im Backend verdrahtet (hardcoded). Stattdessen ruft der Service die Berechnungslogik über ein definiertes Interface auf. Dies ermöglicht es, die deterministische Planungsstrategie des aktuellen Entwicklungsstandes später nahtlos gegen komplexere KI-Module oder alternative Algorithmen auszutauschen, ohne die Kernarchitektur der Service-Schicht anpassen zu müssen.
- **Singleton Pattern:** Zentrale Dienstleistungskomponenten und Ressourcenverwalter, die einen globalen Status halten oder deren mehrfache Instanziierung unnötigen Overhead erzeugen würde (z. B. Konfigurations-Services oder Datenbank-Verbindungspools), werden als Singletons implementiert. Dies stellt sicher, dass systemweit nur eine einzige Instanz existiert, was die Datenkonsistenz wahrt und den Speicherverbrauch des Backends optimiert.
- **Dependency Injection (DI):** Abhängigkeiten zwischen den Schichten (z. B. von einem Service zu einem Repository) werden über Konstruktoren von außen injiziert. Dies entkoppelt die Klassen voneinander und bereitet die Architektur optimal auf automatisierte Komponententests vor, da echte Datenbankzugriffe durch Mock-Objekte ersetzt werden können.
- **Data Transfer Objects (DTOs):** Für den Datenaustausch zwischen dem React-Client und der Backend-API werden dedizierte DTO-Klassen verwendet. Sie stellen sicher, dass interne Datenbankmodelle nicht nach außen durchsickern und minimieren die Payload-Größe im Netzwerk.
- **Objektorientiertes Design (OOD):** Im Gegensatz zur Analysephase (OOA) weisen die modellierten Klassen im Entwurf präzise C#-Datentypen, definierte Sichtbarkeiten (Modifier wie public, private) sowie detaillierte Methoden-Signaturen auf. Beziehungen zwischen den Objekten wurden spezifisch durch starke (Komposition) und schwache (Aggregation) Teil-Ganzes-Beziehungen sowie exakte Multiplizitäten aufgelöst.

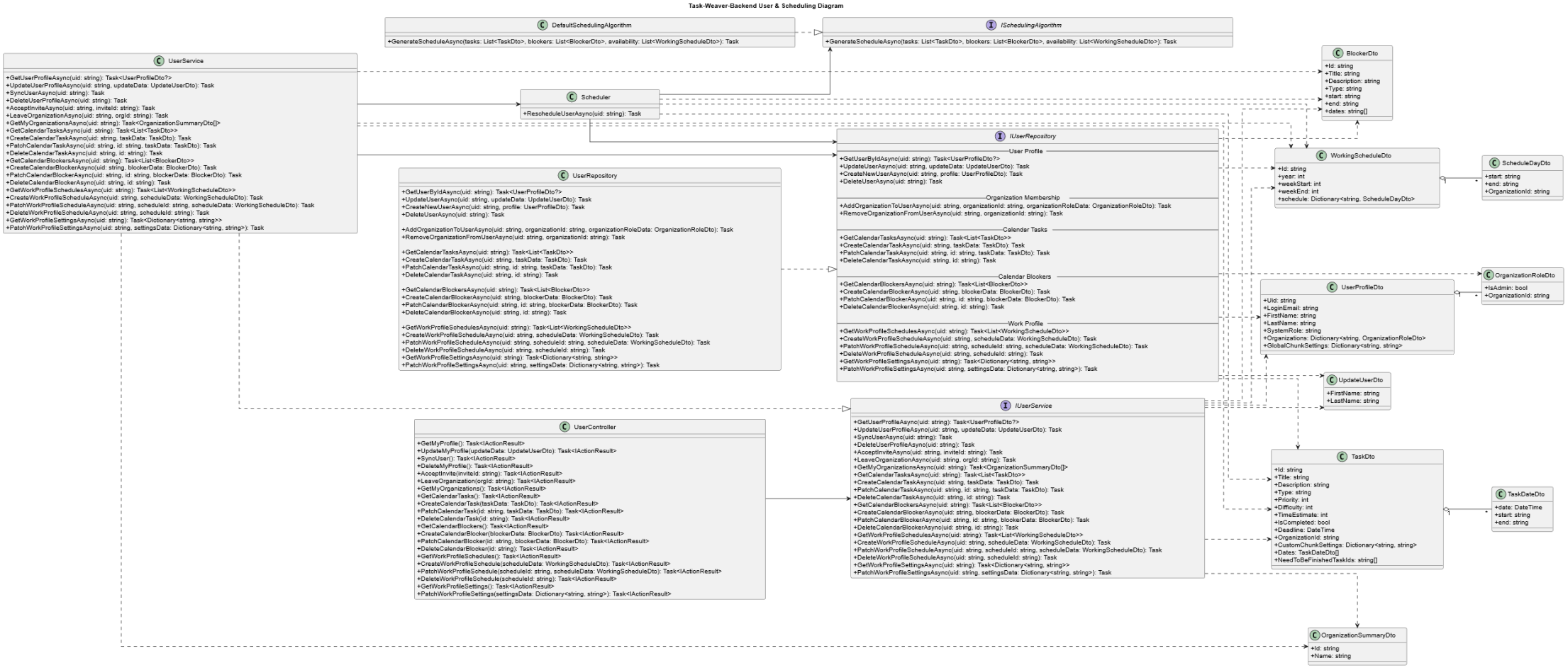
Fokus des statischen Detailentwurfs (OOD):

Gemäß den Best Practices der Modellierung fokussiert sich der nachfolgende Detailentwurf gezielt auf den wesentlichen Komplexitätsschwerpunkt des Systems: Den Lebenszyklus einer Aufgabe (Task), vom API-Eingang über die algorithmische Planungs-Engine bis zur Persistenz. Um die Lesbarkeit des Modells zu gewährleisten und Überspezifikationen zu vermeiden, wurden periphere Domänen (wie das User- und Organisations-Management über Firebase Auth) in diesem Diagramm bewusst ausgeblendet. Da diese Randdomänen exakt demselben architektonischen Controller-Service-Repository-Muster folgen, bietet das dargestellte Modell einen repräsentativen und vollständigen Beweis für die architektonische Struktur des gesamten Backends.

TASKFLOW ENGINEERING

Designbeschreibung

TaskFlow Engineering




Task-Weaver-Backend DTO Diagram
WorkingScheduleDto
+Id: string
+year: int
+weekStart: int
+weekEnd: int
+schedule: Dictionary<string, ScheduleDayDto>
ScheduleDayDto
+start: string
+end: string
+OrganizationId: string
1
*
TaskDto
+Id: string
+Title: string
+Description: string
+Type: string
+Priority: int
+EnergyLevel: int
+TimeEstimateMinutes: int
+IsCompleted: bool
+Deadline: DateTime
+OrganizationId: string
+CustomChunkSettings: Dictionary<string, string>
+Dates: TaskDateDto[]
+NeedToBeFinishedTaskIds: string[]
TaskDateDto
+date: DateTime
+start: string
+end: string
1
*
OrganizationSummaryDto
+Id: string
+Name: string
1
0..*
OrganizationRoleDto
+IsAdmin: bool
+OrganizationId: string
UserProfileDto
+Uid: string
+LoginEmail: string
+FirstName: string
+LastName: string
+SystemRole: string
+Organizations: Dictionary<string, OrganizationRoleDto>
+GlobalChunkSettings: Dictionary<string, string>
1
*
OrganizationDto
+Id: string
+Name: string
+DomainWhiteList: string[]
+EmailWhiteList: string[]
+Employees: string[]
+InviteLinkValidityDays: int
UpdateUserDto
+FirstName: string
+LastName: string
BlockerDto
+Id: string
+Title: string
+Description: string
+Type: string
+start: string
+end: string
+dates: string[]
MemberDto
+Id: string
+FirstName: string
+LastName: string
+LoginEmail: string
InviteDto
+Id: string
+Email: string
+ExpiresAt: DateTime

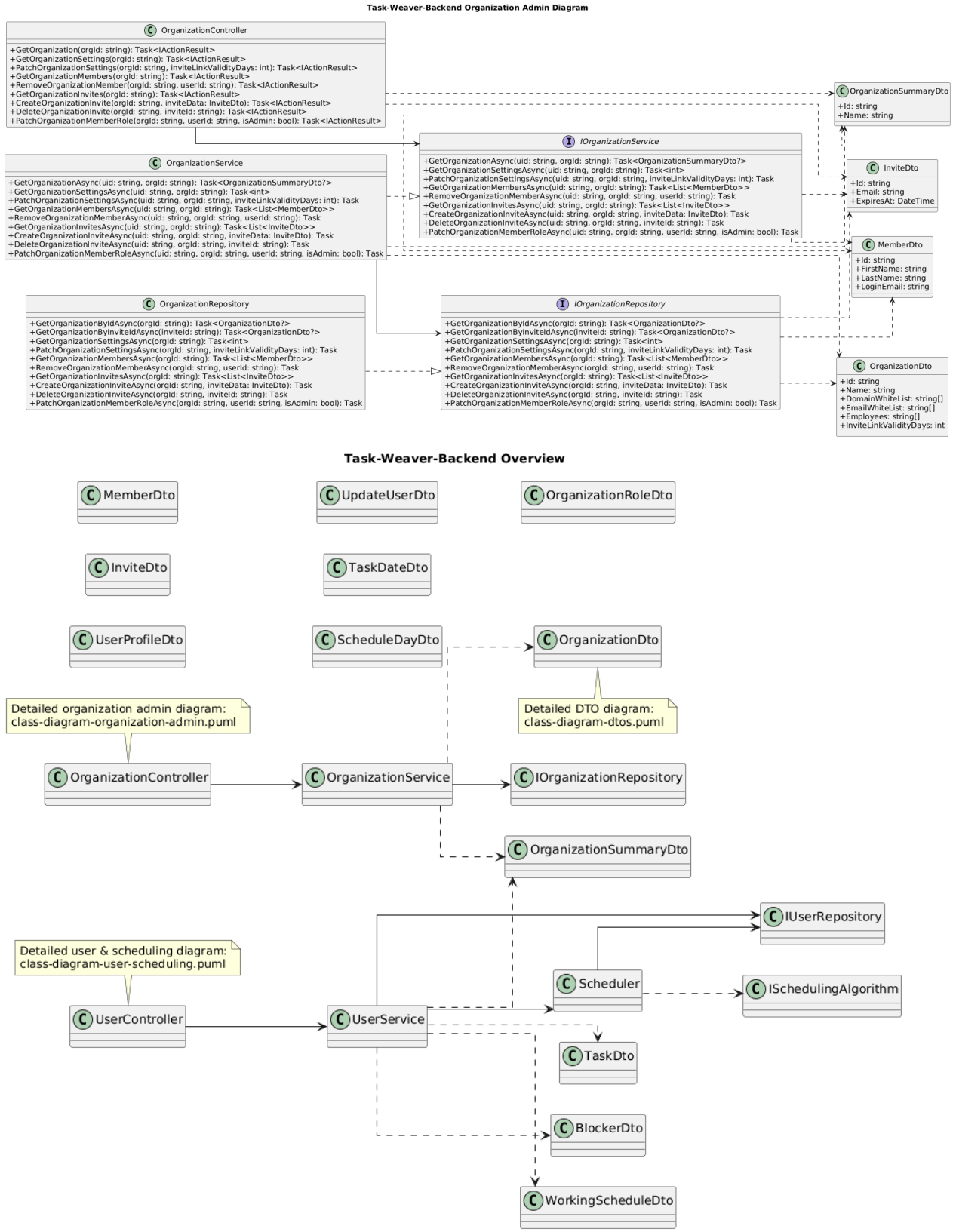


Abbildung 3: UML-Klassendiagramm

Das Klassendiagramm belegt die konsequente Übersetzung der architektonischen Systemgrenzen in den statischen Code-Entwurf. Es rechtfertigt die Nutzung von streng typisierten Data Transfer Objects (DTOs), um die interne Domain-Logik der Aufgabenplanung vollständig zu kapseln und Payload-Größen zwischen Client und Server zu minimieren.

4.3. Datenmodellierung und Persistenzstruktur (Cloud Firestore)

Für die Persistenzschicht von TaskFlow Engineering wird die dokumentenorientierte NoSQL-Datenbank Cloud Firestore eingesetzt. Der Grund der auswahl wurde bereits in Kapitel 3.3 beschrieben.
Das Schema ist so entworfen, dass es die architektonische Leitlinie der individuellen Datenisolation strikt unterstützt.

Struktur des Datenbank-Schemas

Das logische Datenmodell basiert auf zentralen Dokumenten-Kollektionen und hierarchischen Unterkollektionen (Sub-Collections), um eine klare Trennung der Nutzerdaten zu gewährleisten:

Root-Kollektionen

Das System ist in zwei primäre Einstiegspunkte unterteilt, um die Nutzer-Isolierung sowie die systemweite Datenintegrität zu ermöglichen:

- USERS: Das Zentrum für alle individuellen Nutzerdaten und persönlichen Einstellungen.
- COMPANIES: Speichert organisationsweite Metadaten, Whitelists und administrative Informationen wie z.B. “inviteLinkValidityDays”.

Die 4 Sub-Kollektionen von USERS

Um die architektonische Leitlinie der individuellen Datenisolation umzusetzen, liegen alle operativen Daten als Unterkollektionen direkt unter dem jeweiligen Nutzerdokument:

- **TASKS**: Enthält die granularen Aufgabenparameter wie difficulty, timeEstimate und den energyLevel. Hier ist auch die Referenz zur zugehörigen Organisation hinterlegt (company-Feld).

- **BLOCKER**: Speichert statische Ereignisse (z.B. Arzttermine), die Zeit belegen, aber keinen kognitiven Energiebedarf haben.

- **WORKING_SCHEDULE**: Beinhaltet die zeitlichen Verfügbarkeiten (SCHEDULE_MAP) und definiert die Arbeitstage (DAY_DETAILS).

- **EMPLOYEE_OF_COMPANIES**: Diese Sub-Kollektion bildet die Brücke zur Mandantenfähigkeit. Sie speichert die role (z.B. „Admin“ oder „Freelancer“) und den status des Nutzers innerhalb einer spezifischen Company.

Referenz-Logik und Verknüpfungen

Obwohl Firestore eine NoSQL-Datenbank ist, nutzt das Schema gezielte Referenzen, um die Integrität zu wahren:

- **Task zu Company**: Jede Aufgabe in der TASKS-Sub-Kollektion referenziert über eine ID die zugehörige Firma in der COMPANIES-Root-Kollektion.
- **Company zu User**: In der Root-Kollektion COMPANIES führt ein Array namens employees die UIDs aller zugehörigen Nutzer, um den Einladungsprozess und die Zugriffskontrolle zu steuern.
- **Identität**: Die Dokumenten-ID innerhalb der EMPLOYEE_OF_COMPANIES-Sub-Kollektion entspricht der jeweiligen Company-ID, was eine schnelle Abfrage der Nutzerberechtigungen ermöglicht.

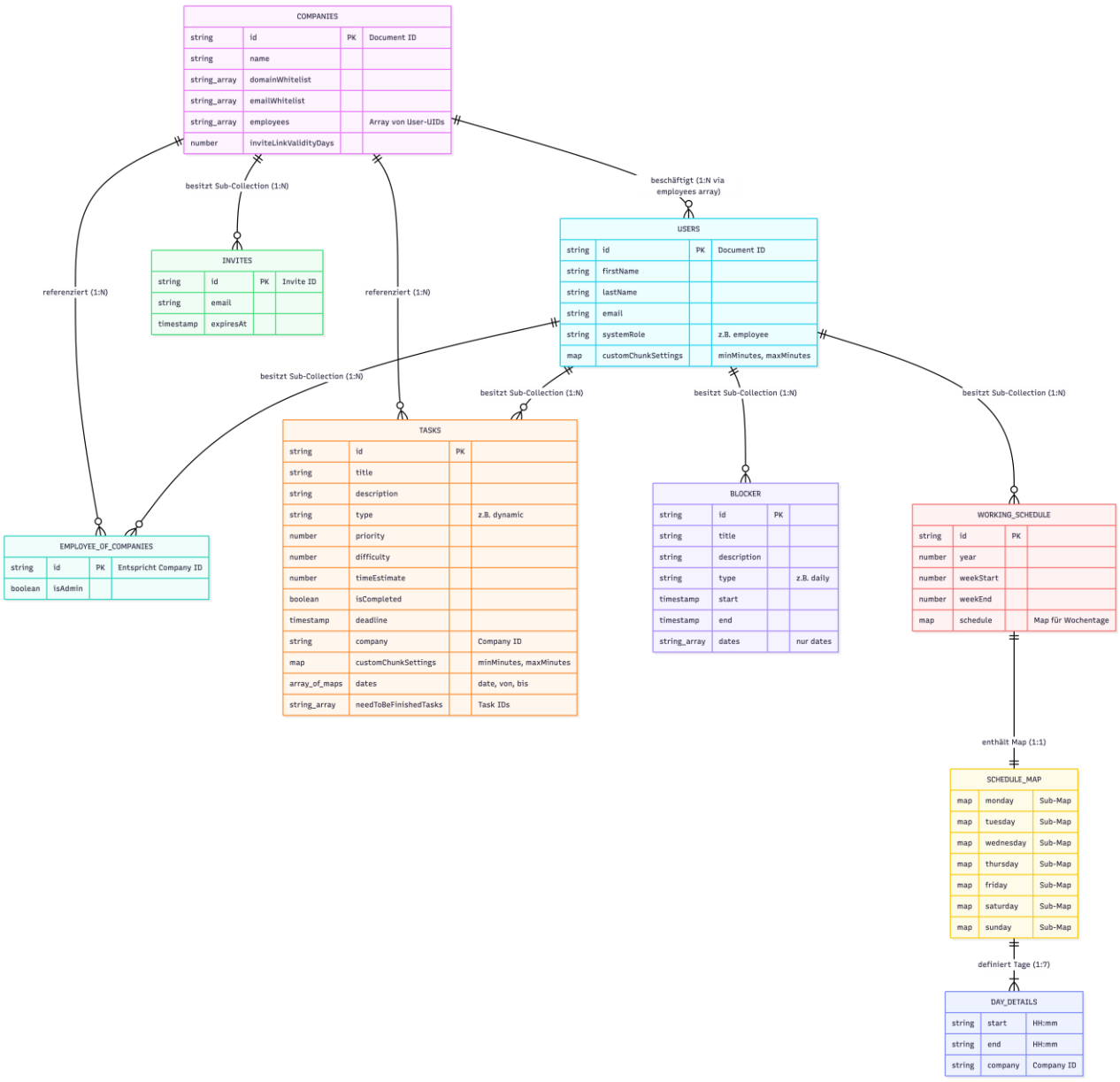


Abbildung 4: Datenbank Schema

Da NoSQL-Datenbanken keine relationalen Integritätsbedingungen (Foreign Keys) erzwingen, ist dieses Diagramm zwingend erforderlich, um die Verknüpfungslogik (z. B. Task zu Company) zu spezifizieren. Es begründet zudem die Sicherheitsarchitektur der "Sub-Kollektionen", die eine harte Isolation der individuellen Nutzerdaten physisch auf Datenbankebene garantieren.

4.4 Dynamisches Verhaltensmodell

Dieser Abschnitt spezifiziert die Verhaltenslogik und den Intersystem-Datenaustausch des TaskFlow Engineering Systems. Die Applikation löst Optimierungsprobleme im Bereich des Aufgabenmanagements unter Nebenbedingungen. Der architektonische Fokus liegt dabei auf der Einhaltung harter Kapazitätsgrenzen, um eine algorithmische Überplanung des Nutzers zu verhindern. Das nachfolgende dynamische Modell beweist mathematisch und logisch, wie die abstrakten Systemanforderungen in konkrete, ausführbare Software-Architekturen übersetzt wurden.

4.4.1. Algorithmischer Kontrollfluss (Aktivitätsdiagramm)

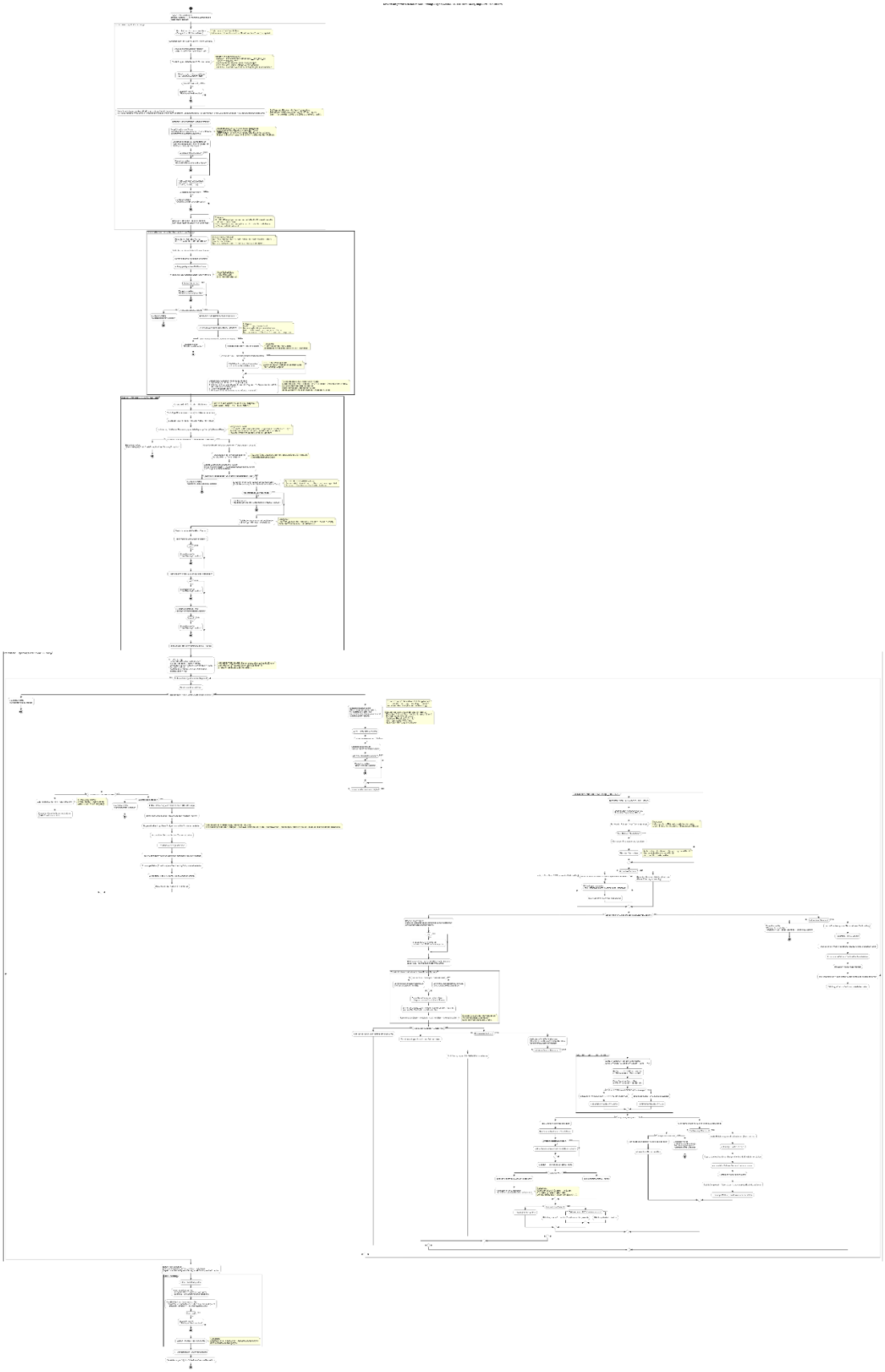


Abbildung 5: Aktivitätsdiagramm: Algorithmischer Kontrollfluss

Das vorliegende Aktivitätsdiagramm modelliert die Kern-Engine der automatisierten Aufgabenplanung. Um die im Pflichtenheft definierten harten Restriktionen (/F10/, /F40/) zu garantieren, durchläuft das Token drei aufeinanderfolgende architektonische Schutzmechanismen:

1. Vorverarbeitung
Der Algorithmus bereitet alle Aufgaben vor:

- Statische Aufgaben werden getrennt.
- Bereits erledigte Teile dynamischer Aufgaben werden erkannt und gesperrt.
- Dynamische Aufgaben werden nach **Priorität** → **Deadline** sortiert.
- Danach wird eine **abhängigkeitskonforme Reihenfolge** erzeugt.

2. Statische Aufgaben einplanen
Statische Aufgaben haben feste Zeiten und werden direkt eingetragen, sofern sie nicht mit Blockern kollidieren.

3. Dynamische Aufgaben planen (mit Backtracking)
Dies ist der komplexeste Teil.

Einplanung berücksichtigt:

- Arbeitszeiten
- Blocker
- Organisationsregeln
- Chunk-Einstellungen
- Deadline
- Bereits erledigte Teile
- **Energielevel-Regeln:**
  - **Level 3 (sehr schwer): max. 2h pro Tag**
  - **Level 2 (schwer): max. 4h pro Tag**
  - **Level 1 (normal): keine Begrenzung**

Ablauf:

1. Nächste sinnvolle Aufgabe auswählen.
2. Versuchen, die verbleibende Dauer einzuplanen.
3. Wenn das nicht geht:
   a. Eine alternative Aufgabe probieren.
4. Wenn das auch nicht geht:
   a. Backtracking: letzte Aufgabe rückgängig machen.
5. Wenn der Backtracking-Stack leer ist:
   a. Keine Lösung möglich.

4. Finalisierung

- Alle Blöcke werden zusammengeführt.
- Nach Zeit sortiert.
- Auf Überschneidungen geprüft.

4.4.2. Intersystem-Datenaustausch (Sequenzdiagramm)

Das Sequenzdiagramm dokumentiert den Datenaustausch zwischen den Systemkomponenten und zeigt das Zusammenspiel zwischen dem Frontend, dem Backend und der Cloud-Datenbank. Die grafische Darstellung ist hier notwendig, um

die zeitlichen Abläufe und die Abhängigkeiten bei einer Planänderung präzise abzubilden, was in reinem Text kaum nachvollziehbar wäre. Um Überspezifikation zu vermeiden, fokussiert sich dieses Modell auf den architektonischen Kernprozess des Systems: Das automatisierte Rescheduling nach einer manuellen Planänderung.

Ein zentraler Punkt ist dabei der „Auto-Trigger“-Mechanismus. Sobald ein Nutzender eine Aufgabe manuell verändert – etwa die Dauer anpasst oder einen Termin verschiebt – löst das System im Hintergrund automatisch eine Kette von Reaktionen aus. Folgende Entwurfsentscheidungen sind hierbei maßgeblich:

1. **Die automatisierte Plan-Aktualisierung (Engine-Trigger):** Der Prozess wird durch einen asynchronen Eingriff des Nutzenden initiiert. Das Modell macht deutlich, dass das System diese Modifikation nicht nur in der Datenbank speichert, sondern im Backend autonom einen Neuaufruf der Service-Engine auslöst. Dabei wird der Flow-Schutz berechnet und das Rescheduling für alle betroffenen Aufgaben gestartet. Dies garantiert, dass die kognitiven Limits trotz der händischen Änderung weiterhin konsequent eingehalten werden.

2. **Absicherung der Datenkonsistenz (Combined Fragment 'alt'):** Da Datenbankoperationen und komplexe Berechnungen fehleranfällig sein können, ist die Persistierung des neu generierten Arbeitsplans in eine logische Verzweigung (Alternative) gekapselt:

    - **Erfolgsfall [Success]:** Der neue Plan wird erfolgreich gespeichert und zur Reduzierung der Netzwerklast direkt als kompaktes Datenobjekt (DTO) in der Antwort an das Frontend übertragen. Die Kalenderansicht aktualisiert sich daraufhin sofort ohne Zeitverzögerung.

    - **Fehlerfall [Error]:** Sollte die Berechnung zu einem Fehler führen – beispielsweise wenn eine kognitive Kapazitätsgrenze verletzt wird – bricht das System den Vorgang kontrolliert ab. Das Backend sendet in diesem Fall einen entsprechenden Fehler-Code, der im Frontend das Clash-Management-UI aktiviert. So wird sichergestellt, dass der Nutzende transparent über das Planungsproblem informiert wird, anstatt mit einem inkonsistenten Plan zu arbeiten.

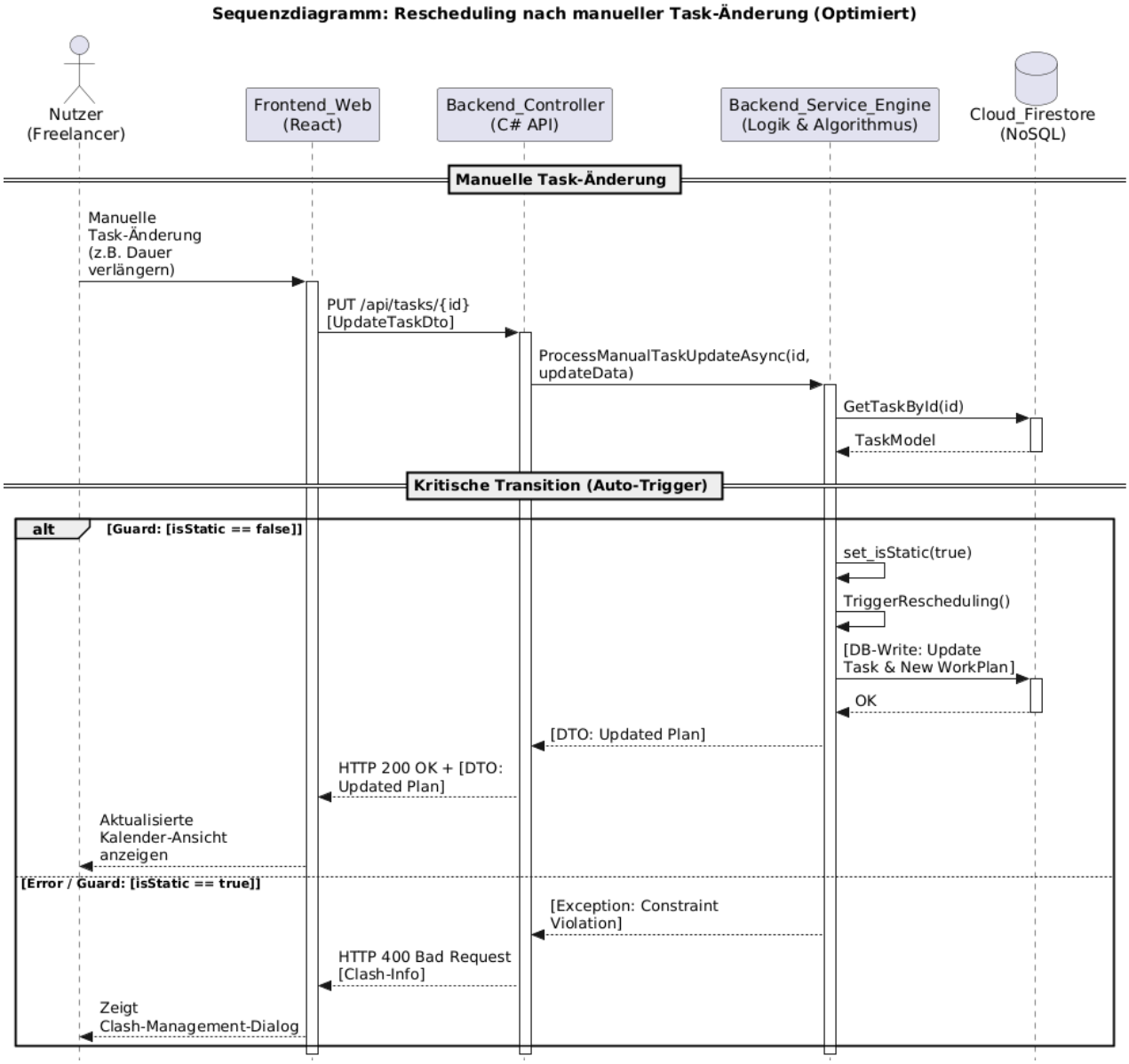


Abbildung 6: Sequenzdiagramm: Rescheduling

4.4.3. Objekt-Lebenszyklus der Aufgabe (Zustandsautomat)

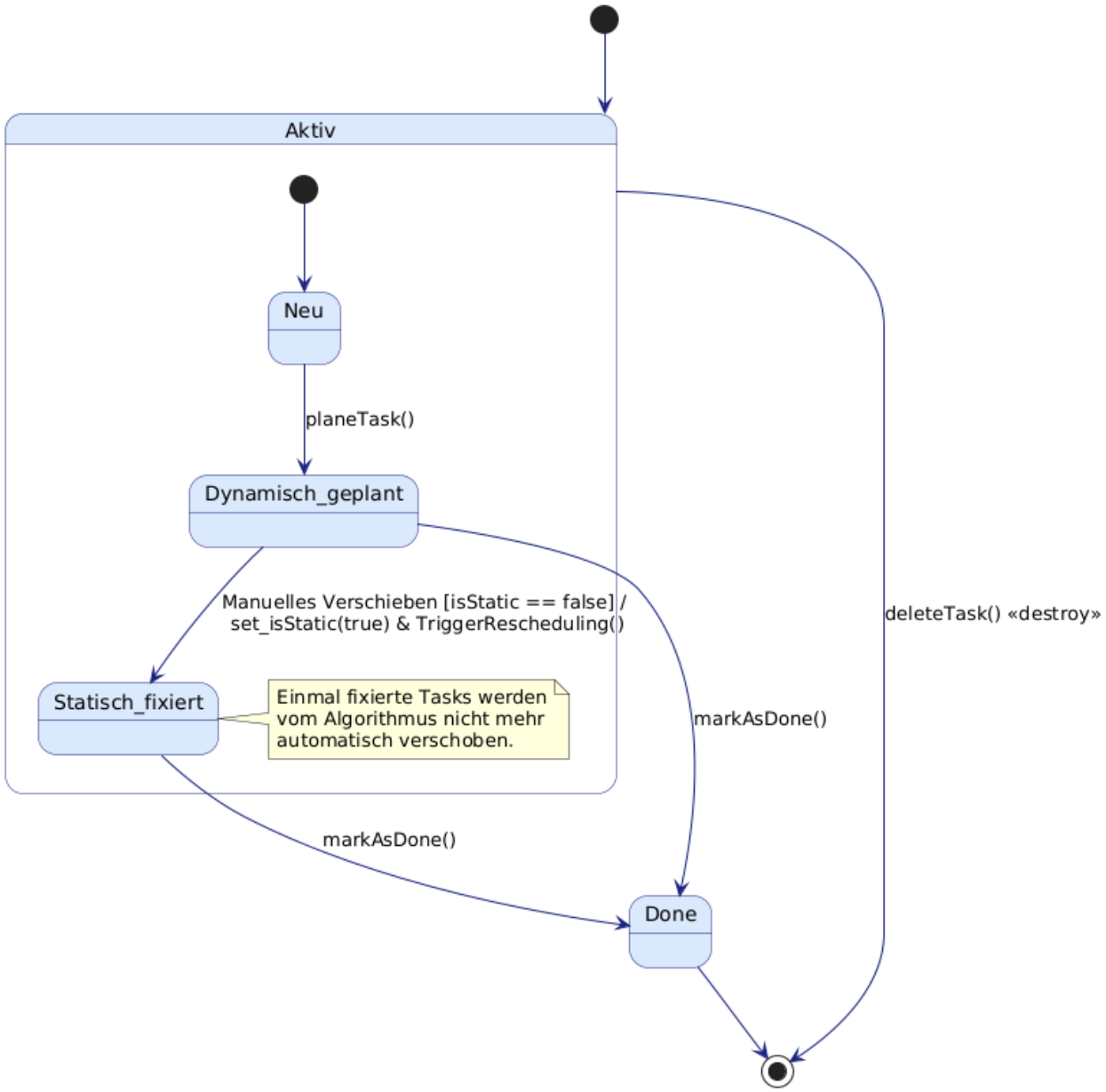


Abbildung 7: Zustandautomat: Objekt-Lebenszyklus einer Aufgabe

Das Zustandsdiagramm modelliert den Lebenszyklus einer Aufgabe und dient an dieser Stelle insbesondere dazu, eine wichtige Designentscheidung für die Implementierung aufzuzeigen: den bewussten Verzicht auf einen Rückweg von manuell fixierten Aufgaben in den dynamischen Planungs-Pool.
Wenn ein Teilbereich (Chunk) einer dynamischen Aufgabe vom Nutzenden händisch im Kalender verschoben wird, wechselt dieser zwingend in den Zustand Statisch_fixiert. Das Diagramm zeigt hier absichtlich eine Einbahnstraße, da die Möglichkeit, diese Fixierung wieder zu lösen, den Algorithmus vor enorme Probleme stellen würde. Ein simples Zurücksetzen des Status würde nicht ausreichen. Die

Engine müsste stattdessen aufwendig prüfen, ob und wie der wieder freigegebene Chunk mit den restlichen Teilen der Ursprungsaufgabe neu verschmolzen werden kann. Ohne eine solche Logik würde der Kalender bei mehrfachem Hin- und Herwechseln schnell in ineffiziente Kleinstblöcke zersplittern, was den Flow-Schutz der minimalen Bearbeitungszeit verletzen würde.
Zudem dient ein fixierter Chunk als harter Ankerpunkt für nachfolgende, abhängige Aufgaben. Fällt dieser Anker durch ein erneutes Lösen der Fixierung weg, müsste die Engine aufwendig den gesamten Abhängigkeitsbaum rückabwickeln und komplett neu berechnen. Um die Stabilität des Kernalgorithmus zu sichern und das Projekt im eng gesteckten Zeitrahmen erfolgreich abzuschließen, wurde auf dieses komplexe Hin und Her verzichtet. Der Zustand Statisch_fixiert bildet für manuell verschobene Teilblöcke somit einen fest definierten Endzustand.
Architektonisch essenziell für die Erhaltung der Systemintegrität ist die an diese Transition gekoppelte Verhaltensanweisung (Aktion): / set_isStatic(true) & TriggerRescheduling(). Diese Notation beweist, dass jede manuelle Fixierung einer Aufgabe eine atomare Kette auslöst: Der betroffene Chunk wird für zukünftige Algorithmus-Läufe gesperrt (Einmal fixierte Teilblöcke werden vom Algorithmus nicht mehr automatisch verschoben), und gleichzeitig wird ein systemweites Rescheduling für alle verbleibenden offenen Aufgaben getriggert. Dadurch garantiert das System, dass die harte 2-4-Stunden-Kapazitätsgrenze für den betroffenen Arbeitstag unter den neuen, manuell geschaffenen Bedingungen sofort neu berechnet und validiert wird. Den regulären Endzustand erreicht das Objekt durch das Ereignis markAsDone(), wodurch es in den Zustand Done wechselt. Architektonisch muss zudem der Lebenszyklus-Abbruch berücksichtigt werden, da die API das unwiderrufliche Löschen (DELETE /api/v1/user/tasks/{id}) erlaubt. Aus jedem der aktiven Zustände existiert daher implizit eine Transition deleteTask(), die das Objekt sofort aus der Persistenz entfernt, den Algorithmus bereinigt und den Lebenszyklus vorzeitig terminiert.

4.4.4. Mandantenfähigkeit und Einladungsprozess

Die Architektur erzwingt hier eine strikte funktionale Trennung. Der Einladungsprozess ist funktional auf die Registrierung externer Freelancer beschränkt. Die Zuweisung regulärer Mitarbeiter entfällt in diesem aktiven Ablauf vollständig; sie wird stattdessen architektonisch asynchron beim Login über einen Abgleich mit der WHITELIST_REGISTRY abgehandelt. Bei der aktiven Freelancer-Einladung prüft das Backend in der Firestore-Datenbank, ob die Ziel-E-Mail bereits existiert. Ist dies der Fall, wird die Referenz in der EMPLOYEE_OF_COMPANIES-Kollektion zwingend mit der festen Rolle 'Freelancer' gesetzt. Existiert der Nutzer noch nicht, wird ein 'Pending Invite' generiert.

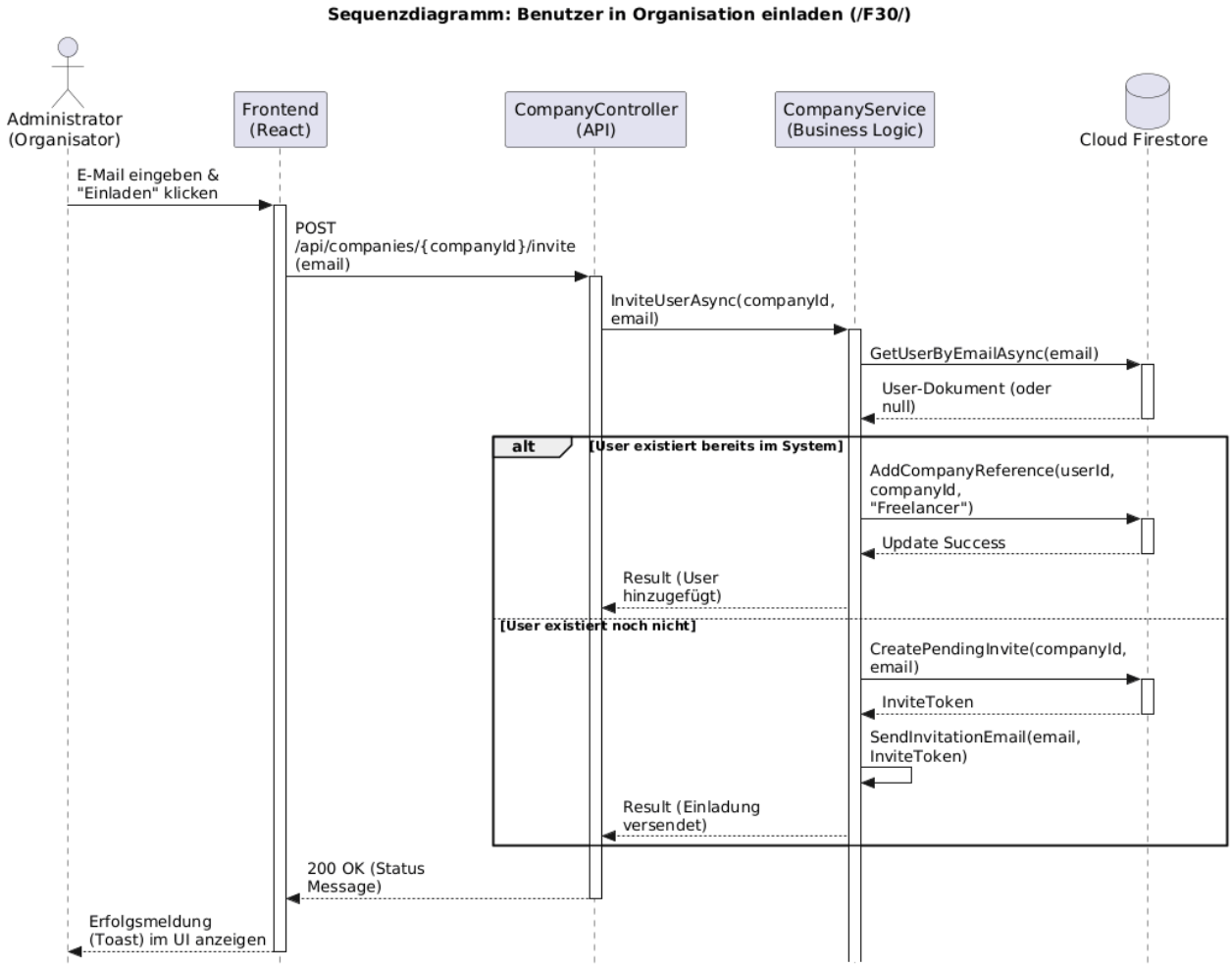


Abbildung 8: Sequenzdiagramm: Benutzer in Organisation einladen

4.5. API-Spezifikation (REST-Schnittstellen)

Die Kommunikation zwischen dem React-Frontend und dem C#-Backend erfolgt über eine RESTful API. Als Datenaustauschformat wird standardmäßig application/json verwendet. Die Authentifizierung erfolgt zustandslos über JWT-Tokens (Firebase Auth).

Globale Authentifizierungsvorgabe: Alle Anfragen an geschützte API-Routen erfordern zwingend die Mitgabe eines gültigen Firebase-JWT-Tokens im HTTP-Header. Das Frontend muss diesen bei jedem Request wie folgt mitsenden: Authorization: Bearer <Firebase_JWT>

Fehlt dieser Header, ist er fehlerhaft oder abgelaufen, lehnt das Backend die Anfrage automatisch ab (HTTP 401 Unauthorized), ohne den zugehörigen Controller-Code auszuführen. Dies wird serverseitig durch das [Authorize]-Attribut und die ASP.NET Core Authentication Middleware sichergestellt.

Globale HTTP-Statuscodes

- 200/201/204: Erfolg
- 400: Bad Request (Validierungsfehler)
- 401: Unauthorized (Token fehlt/ungültig)
- 403: Forbidden (Fehlende Rechte)
- 404: Not Found
- 409: Conflict (Zustandskonflikt)
- 422: Unprocessable Entity (Logikfehler/Algorithmus-Abbruch)
- 429: Too Many Requests
- 500: Internal Server Error

Rückgabe bei den Requests:

```
(
        “statusCode”,
        {
          "errorCode": "SPECIFIC_ERROR_CODE",
          "details": [
            { "field": "timeEstimate", "issue": "Must be greater than 0" }
          ]
        }
)
```

User & Identity (/api/v1/user)

Wichtiger Hinweis: Sämtliche Endpunkte in User & Identity beziehen sich implizit auf den aktuell authentifizierten Nutzer, der den Request ausführt.

Methode	Endpunkt	Zweck	Wichtige Parameter	Response
GET	/me	Profil des authentifizierten Nutzers abrufen	-	Erfolg: 200 OK (User-Objekt) Fehler: 401 Unauthorized
PUT	/me	Profil des authentifizierten Nutzers aktualisieren	{id: number, loginEmail: string, firstName: string, lastName: string, systemRole: string}	Erfolg: 200 OK (User-Objekt) Fehler: 401 Unauthorized
POST	/sync	Erstellt/Synchronisiert das Firestore Profil	{firstName: string, lastName: string}	Erfolg: 201 Created (User-Objekt) Fehler: 400 Bad Request
POST	/accept-invite/{inviteId}	Einladung annehmen und Organisation beitreten	-	Erfolg: 201 Created Fehler: 404 Not

				Found (Token ungültig), 409 Conflict
POST	/leave-organization/{orgId}	Organisation als Mitglied freiwillig verlassen	-	Erfolg: 204 No Content Fehler: 404 Not Found
GET	/my-organizations	Liste aller Organisationen eines Nutzers abrufen	-	Erfolg: 200 OK (Array von Company-Objekten) Fehler: 401 Unauthorized

Arbeitsprofil & Settings (/api/v1/user/work-profile)

Methode	Endpunkt	Zweck	Wichtige Parameter	Response
GET	/schedules	Ruft alle Arbeitspläne eines Nutzers ab	-	Erfolg: 200 OK (Array von Schedule-Objekten) Fehler: 404 Not Found
POST	/schedule	Neuen Wochenplan (Verfügbarkeiten) anlegen	{year: number, weekStart: number, weekEnd: number, schedule: Object[]}	Erfolg: 201 Created (Schedule-Objekt) Fehler: 400 Bad Request, 409 Conflict
PATCH	/schedules/{scheduleId}	Bestehenden Arbeitsplan bearbeiten	{year: number, weekStart: number, weekEnd: number, schedule: Object[]}	Erfolg: 200 OK (Schedule-Objekt) Fehler: 400 Bad Request, 404 Not Found
DELETE	/schedules/{scheduleId}	Arbeitsplan vollständig löschen	-	Erfolg: 204 No Content Fehler: 404 Not Found
GET	/settings	Globale Voreinstellungen für Chunk Größen (minimale und maximale Dauer zusammenhängender Arbeitsphasen)	-	Erfolg: 200 OK (Settings-Objekt) Fehler: 404 Not Found

PATCH	/settings	Aktualisiert globalen Chunkgrößen des Nutzers	{defaultMinChunkMinutes: number, defaultMaxChunkMinutes: number}	Erfolg: 200 OK (Settings-Objekt) Fehler: 400 Bad Request

Aufgaben (Tasks) (/api/v1/user/calendar)

Methode	Endpunkt	Zweck	Wichtige Parameter	Response
GET	/tasks	Liste aller Aufgaben abrufen	-	Erfolg: 200 OK (Array von Task-Objekten) Fehler: 401 Unauthorized
POST	/task	Neue Aufgabe erstellen	{type: string, priority: number, energyLevel: number, timeEstimateMinutes: number, recurrenceCount: number (nullable), deadline: string (ISO 8601), description: string, title: string, needToBeFinishedTasks: String[], company: string, customChunkSettings: {minMinutes: number, maxMinutes: number} (nullable), isCompleted: boolean}	Erfolg: 201 Created (Task-Objekt) Fehler: 400 Bad Request, 422 Unprocessable Entity (Clash)
PATCH	/tasks/{id}	Bestehende Aufgabe (teilweise) überschreiben	{type: string, priority: number, energyLevel: number, timeEstimateMinutes: number, recurrenceCount: number (nullable), deadline: string (ISO 8601), description: string, title: string, needToBeFinishedTasks: String[], company: string, customChunkSettings: {minMinutes: number, maxMinutes: number} (nullable), isCompleted:	Erfolg: 200 OK (Task-Objekt) Fehler: 400 Bad Request, 404 Not Found, 422 Unprocessable Entity (Clash)

			boolean}	
DELETE	/tasks/{id}	Aufgabe löschen	-	Erfolg: 204 No Content Fehler: 404 Not Found
POST	/tasks/{id}/move-chunk	Einen Task-Block verschieben	{sourceDate: string (ISO 8601), sourceStart: string (ISO 8601), sourceEnd: string (ISO 8601), targetDate: string (ISO 8601), targetStart: string (ISO 8601), targetEnd: string (ISO 8601)}	Erfolg: 200 OK (Task-Objekt) Fehler: 400 Bad Request, 404 Not Found, 422 Unprocessable Entity (Clash)
GET	/blockers	Alle Blocker abrufen	-	Erfolg: 200 OK (Array von Blocker-Objekten) Fehler: 401 Unauthorized
POST	/blocker	Neuen (einmaligen oder wiederkehrenden) Blocker anlegen	{title: string, description: string, type: string, recurrenceCount: number (nullable), start: string (ISO 8601), end: string (ISO 8601)}	Erfolg: 201 Created (Blocker-Objekt) Fehler: 400 Bad Request, 422 Unprocessable Entity (Clash)
PATCH	/blockers/{blockerId}	Bestehenden Blocker (teilweise) anpassen	{title: string, description: string, type: string, recurrenceCount: number (nullable), start: string (ISO 8601), end: string (ISO 8601)}	Erfolg: 200 OK (Blocker-Objekt) Fehler: 400 Bad Request, 404 Not Found, 422 Unprocessable Entity (Clash)
DELETE	/blockers/{blockerId}	Blocker löschen	-	Erfolg: 204 No Content Fehler: 404 Not Found

Organisations Management (/api/v1/organizations)

Methode	Endpunkt	Zweck	Wichtige Parameter	Response
GET	/{orgId}	Details einer spezifischen Organisation abrufen (nur Mitglieder)	-	Erfolg: 200 OK (Company-Objekt) Fehler: 403 Forbidden, 404 Not Found
GET	/{orgId}/settings	Organisations-Einstellungen abrufen (Nur für Admins)	-	Erfolg: 200 OK (CompanySettings-Objekt) Fehler: 403 Forbidden (Kein Admin), 404 Not Found
PATCH	/{orgId}/settings	Organisations-Einstellungen partiell anpassen (Nur für Admins)	{inviteLinkValidityDays: number, whiteListDomains: String[], whiteListEmails: String[]}	Erfolg: 200 OK (CompanySettings-Objekt) Fehler: 400 Bad Request, 403 Forbidden
GET	/{orgId}/members	Mitgliederliste einer Organisation abrufen (Nur für Admins)	-	Erfolg: 200 OK (Array von Member-Objekten) Fehler: 403 Forbidden, 404 Not Found
DELETE	/{orgId}/members/{userId}	Freelancer: Nutzer aus der Organisation entfernen. Employee: Nutzer zunächst aus der Organisation entfernen und anschließend vollständig löschen. (Nur für Admins)	-	Erfolg: 204 No Content Fehler: 403 Forbidden, 404 Not Found
GET	/{orgId}/invites	Offene Einladungen anzeigen (Nur für Admins)	-	Erfolg: 200 OK (Array von Invite-Objekten) Fehler: 403 Forbidden, 404

				Not Found
POST	/{orgId}/invite	Nutzer in Organisation einladen (Nur für Admins)	{email: string }	Erfolg: 201 Created ({ "token": string }) Fehler: 400 Bad Request, 403 Forbidden, 409 Conflict (User existiert)
DELETE	/{orgId}/invites/{inviteId}	Offene Einladung zurückziehen (Nur als Admin)	-	Erfolg: 204 No Content Fehler: 403 Forbidden, 404 Not Found
PATCH	/{orgId}/members/{userId}/ role	Rolle eines bestehenden Mitglieds anpassen (Nur als Admin)	{isAdmin: boolean}	Erfolg: 200 OK (Member-Objekt) Fehler: 400 Bad Request, 403 Forbidden, 404 Not Found
POST	/check-whitelist	E-Mail-Whitelist prüfen	Query: email	Erfolg: 200 OK Fehler: 400 Bad Request

4.6. Architektonische Nachverfolgbarkeit (Traceability-Matrix)

Anforderungs-ID	**Anforderung**	**Implementierendes Modell / Komponente**	**Architektonische Entwurfsentscheidung & Begründung**
/F40/	**Arbeitsplangenerierung & Kognitives Limit** Berücksichtigung des Energiebedarfs und der 2-4 Stunden Kapazitätsgrenze für tiefe Wissensarbeit.	Aktivitätsdiagramm (Algorithmischer Kontrollfluss), Backend PlanningService	Die Logik wurde isoliert in das C#-Backend ausgelagert (Separation of Concerns), um den rechenintensiven Algorithmus performant auszuführen. Ein Guard-Knoten im Kontrollfluss erzwingt den Abbruch, sobald

			das harte Limit von 2-4 Stunden erreicht ist.
/F50/	**Flow-Schutz & Anti-Fragmentierung** Aufgaben dürfen nicht in kognitiv ineffiziente Kleinstblöcke zerteilt werden.	Klassendiagramm (Firestore-Modell), Entity Task	Das Datenmodell erzwingt die Persistenz der Attribute minChunkMin und maxChunkMin. Die Engine nutzt diese DTO-Parameter, um ein Task-Switching unterhalb der definierten Mindestdauer hart zu blockieren.
/N20/	**Individuelle Datenisolation** Absoluter Ausschluss von Team-Dashboards oder Vorgesetzten-Einsicht.	Komponentendiagramm (Gesamtarchitektur), REST-API-Design	Die Architektur ist als Sandbox für das individuelle Selbstmanagement konzipiert. API-Endpunkte (z. B. /api/v1/tasks) validieren zwingend das JWT-Token und liefern ausschließlich Daten des authentifizierten Einzelnutzers. Delegation-Schnittstellen existieren architektonisch nicht.
/F30/	**Abwesenheitsmanagement (Blocker)** Blocker priorisieren und Aufgaben automatisch restrukturieren.	Sequenzdiagramm (Intersystem-Datenaustausch)	Das Speichern eines Blockers triggert auf Backend-Ebene eine asynchrone Kettenreaktion (Rescheduling), welche die bestehenden Tasks ausweichend neu berechnet, ohne den Main-Thread des Frontends zu blockieren.
/F30/	**Globale Eindeutigkeit beim Whitelisting**	Datenmodellierung (Root-Kollektion	Gemäß Pflichtenheft wird die Aktion

		WHITELIST_REGISTRY)	blockiert, wenn eine Domain/E-Mail bereits von einer anderen Organisation gewhitelistet wurde. Da Firestore keine tabellenübergreifenden Unique-Constraints besitzt, wurde ein globales NoSQL-Sperrmuster implementiert. Das Backend nutzt atomare Transaktionen auf die Registry, die bei Kollisionen kompromisslos einen HTTP 409 Conflict werfen.
/F20/ & /F30/	**Restriktive Einladungslogik**	Sequenzdiagramm 4.4.4, Backend Service	Gemäß Pflichtenheft laden Organisierende *ausschließlich* Freelancer via E-Mail-Link ein. Mitarbeiter-Accounts werden asynchron beim Login über die Domain-Whitelist abgehandelt. Das Backend und das Sequenzdiagramm erzwingen diese Trennung: Der Einladungs-Endpunkt weist die Zuweisung regulärer interner Mitarbeiter über Links architektonisch ab.