Designbeschreibung

Erstellt von:	Überprüft von:
Maurice Nowotni	Noah Schwenk

Version	Effektiv ab	Beschreibung / Änderungen
1.0	24.03.2026	Erstellung
1.1	27.03.2026	Überprüfung
1.2	27.03.2026	Feedback eingebaut
1.3	28.06.2026	Letzter Check

Inhalt

1. Allgemeines 3
2. Produktübersicht und umgesetzte Geschäftsprozesse 3
3. Grundsätzliche Struktur- und Entwurfsentscheidungen (Gesamtarchitektur) 4
4. Struktur und Entwurfsentscheidungen der einzelnen Komponenten 5

1. Allgemeines

Das vorliegende Dokument beschreibt die Architektur, die verwendeten Technologien sowie die grundlegenden Entwurfsentscheidungen für die erste Softwarestudie des geplanten Online Aufgaben-Planungssystems. Entsprechend der Projektspezifikation ist diese Softwarestudie primär als Prototyp zu verstehen.
Das vorrangige Ziel dieser ersten Entwicklungsphase ist es nicht, ein reelles, produktionsreifes „Fundament" (Version 0) für das spätere Endprodukt zu schaffen. Vielmehr dient der Prototyp dazu, softwaretechnische Kompetenzen im Team aufzubauen, erste praxisnahe Erfahrungen mit dem ausgewählten Technologie-Stack zu sammeln und architektonische Hypothesen zu validieren. Zudem soll eine Basis geschaffen werden, anhand derer eine unbeteiligte, programmiererfahrene Person die Struktur- und Entwurfsprinzipien des Systems nachvollziehen kann, bevor sie sich in die detaillierte Code-Dokumentation einarbeitet. Die in diesem Dokument getroffenen Entscheidungen bilden somit den Rahmen für die spätere, finale Systemarchitektur, auch wenn für diese Vorabversion noch keine testgetriebene Entwicklung (TDD) oder Testautomatisierung gefordert ist.

2. Produktübersicht und umgesetzte Geschäftsprozesse

Die Anwendung stellt eine webbasierte Plattform dar, über die Benutzer ihre Arbeitszeiten und Aufgaben effizient organisieren können. Für den Umfang dieses Prototyps wurde der Fokus auf die zentralen Kernprozesse der Aufgabenplanung gelegt. Folgende Geschäftsprozesse wurden, basierend auf den vorläufigen Anforderungen, prototypisch umgesetzt:

- **/F10/ Aufgabe konfigurieren:** Nutzer erhalten die Möglichkeit, in ihrem Arbeitsbereich spezifische Aufgaben anzulegen und umfassend zu verwalten. Die Aufgaben können mit wesentlichen Metadaten wie einem aussagekräftigen Titel, einer Beschreibung, der geschätzten Bearbeitungsdauer sowie einem Fälligkeitsdatum (Deadline) versehen werden. Für diesen Prototyp werden typische Interaktionen, wie das Anlegen neuer Aufgaben oder das Anzeigen von Übersichten, bereits erlebbar gemacht. Die dabei eingegebenen Daten müssen jedoch noch nicht persistiert werden. Damit man sich dennoch ein Bild von den Daten machen kann, werden in Listen und Übersichten vorerst einige hard-codierte Dummy-Datensätze angezeigt.
- **/F20/ Auf Plattform registrieren:** Um den Zugriff zu personalisieren und abzusichern, ist eine Registrierung und anschließende Authentifizierung erforderlich. Dies wird klassisch über die Kombination von E-Mail-Adresse und Passwort gelöst.
- **/F30/ Organisierende einladen:** Das System ermöglicht es, administrative Strukturen abzubilden, indem bestehende Organisierende weitere Personen in ihre Organisation einladen können.

- **/F40/ Arbeitsplan erstellen (vereinfachte Ausprägung):** Die Kernfunktion des Systems ist die automatische Generierung eines Arbeitsplans basierend auf den erfassten Informationen. Um die Logik zukunftssicher und austauschbar zu gestalten, soll hier das Strategy Pattern angewandt werden. Der zuständige Service ruft nicht direkt eine feste Berechnungslogik auf, sondern nutzt eine definierte Schnittstelle. Für diesen Prototyp wird über Dependency Injection eine simple, deterministische Strategie injiziert, die offene Zeitslots füllt. Anders als ursprünglich angenommen, werden dabei bereits erste Ideen unseres Planungsalgorithmus sichtbar sein. So ist es bereits möglich, in einem kleinen Rahmen Aufgaben zu planen. Dies legt das Fundament, um zukünftig das Ziel einer ausgewogenen und nachhaltigen Arbeitsplangenerierung zu erreichen.
- **/F50/ Individuelles Arbeitsprofil konfigurieren (bedingte Ausprägung):** Nutzer können in einem reduzierten Umfang ihr Arbeitsprofil definieren. Dies beinhaltet derzeit die Festlegung, an welchen Wochentagen für welches Unternehmen gearbeitet wird. Das System ist dabei flexibel genug konzipiert, um auch Tätigkeiten für unterschiedliche Unternehmen am selben Tag abzubilden.
- **Hinweis:** Auf den optionalen Geschäftsprozess /F60/ (Transparente Darstellung der Planungslogik) wurde im Rahmen dieses Wegwerf-Prototyps verzichtet.

3. Grundsätzliche Struktur- und Entwurfsentscheidungen (Gesamtarchitektur)

Die Architektur der Anwendung folgt einem konsequenten Client-Server-Modell in Verbindung mit einem Cloud-First-Ansatz. Dies bedeutet eine strikte physische und logische Trennung zwischen der Präsentationsschicht (Frontend) und der Geschäftslogik- bzw. Datenhaltungsschicht (Backend). Im Folgenden werden die technologischen Entscheidungen für das Gesamtsystem detailliert begründet.

3.1. Frontend-Architektur: React.js

Für die clientseitige Darstellung haben wir uns für das JavaScript-Framework React.js entschieden. Obwohl im Entwicklungsteam teilweise Vorerfahrungen mit dem Framework Angular vorhanden waren, fiel die Wahl nach einer Abwägung auf React.

- Begründung: React zeichnet sich durch eine deutlich flachere Lernkurve aus, was insbesondere für die schnelle Umsetzung eines Prototyps entscheidend ist. Zudem fördert der komponentenbasierte Ansatz von React eine hohe Wiederverwendbarkeit des Codes. Durch das Virtual DOM (Document Object Model) bietet React eine hervorragende Performance bei der Aktualisierung der Benutzeroberfläche, was für interaktive Anwendungen wie einen Kalender oder eine dynamische Aufgabenplanung essenziell ist.

3.2. Backend-Architektur: C# .NET

Das serverseitige Backend, welches die gesamte Geschäftslogik kapselt und die REST-

Schnittstellen bereitstellt, wurde in C# .NET implementiert.

- Begründung: .NET ist ein bewährtes, stark typisiertes und hochperformantes Enterprise-Framework. Da nahezu das gesamte Entwicklerteam bereits tiefgehende Vorerfahrungen mit C# und dem .NET-Ökosystem besitzt, konnte die Einarbeitungszeit minimiert werden. Die strikte Objektorientierung von C# zwingt das Team zudem förmlich dazu, saubere Architekturmuster zu etablieren. Um die Datensicherheit zu garantieren, ist das Backend so konfiguriert, dass bei jeder eingehenden HTTP-Anfrage (Request) ein JWT (JSON Web Token) validiert wird. Nur authentifizierte Nutzer, die einen gültigen Token im Header mitsenden, können die Schnittstellen passieren.

3.3. Datenhaltung: Firestore (NoSQL)

Im Bereich der Datenbanktechnologie wurde eine bewusste Entscheidung gegen eine klassische relationale SQL-Datenbank (wie z.B. PostgreSQL oder MySQL) und für die dokumentenbasierte NoSQL-Datenbank Google Cloud Firestore getroffen.

- Begründung: Ein Aufgaben-Planungssystem besitzt theoretisch viele Relationen (Nutzer -> Organisation -> Aufgabe -> Abhängigkeiten). Dennoch bietet der NoSQL-Ansatz in der aktuellen Prototyping-Phase überlegene Vorteile. Firestore ermöglicht eine extreme Flexibilität bei der Datenmodellierung. Da sich die Attribute von Aufgaben oder Arbeitsprofilen im agilen Prozess schnell ändern können, erlaubt die schemalose Natur von NoSQL sofortige Anpassungen ohne aufwendige Datenbank-Migration-Scripte. Zudem lassen sich die Daten sehr natürlich in Dokumenten-Hierarchien (Collections und Sub-Collections) strukturieren. Die Nutzerauthentifizierung fügt sich hierbei nahtlos über den angebundenen Dienst Firebase Authentication ein.

3.4. Deployment und Infrastruktur

Um den Overhead für den IT-Betrieb (DevOps) so gering wie möglich zu halten, wird das System vollständig in der Google Cloud bzw. dem Firebase-Ökosystem gehostet.

- Begründung: Das Frontend wird als statischer Build generiert und über Firebase Hosting global bereitgestellt, was extrem schnelle Ladezeiten durch Caching garantiert. Das C#-Backend wird in einen Docker-Container verpackt und über Google Cloud Run ausgeführt. Diese Serverless-Architektur sorgt dafür, dass sich das System bei steigenden Anfragen automatisch skaliert und bei Inaktivität auf null herunterskaliert, was ein hochgradig automatisiertes, wartungsarmes und ressourceneffizientes Deployment sicherstellt.

4. Struktur und Entwurfsentscheidungen der einzelnen Komponenten

Gemäß den Anforderungen aus der Literatur ziehen neue Projekte einen „doppelten Nutzen“, wenn sie etablierten Architekturkonzepten und Design Patterns folgen. Um

diese Qualität und leichte Einarbeitung zu gewährleisten, haben wir klare technologische Entwurfsentscheidungen getroffen: Die strikte Trennung von UI und Logik wird durch React und C# .NET realisiert. Für das Deployment setzen wir mit Google Cloud Run auf Serverless-Konzepte, um Infrastruktur-Overhead zu minimieren. Zudem haben wir uns bewusst für die Datenbank Firestore entschieden. Da wir für die Nutzerauthentifizierung und weitere Basisdienste ohnehin auf das Firebase-Ökosystem setzen, ermöglicht uns Firestore eine nahtlose Integration innerhalb desselben Netzwerks. So stammt die gesamte Daten-, Authentifizierungs- und Infrastrukturverwaltung aus einer Hand (Google Cloud/Firebase), was die Komplexität und den Konfigurationsaufwand des Gesamtsystems deutlich reduziert. Die folgende Abbildung illustriert den Gesamtaufbau der Systemarchitektur sowie die automatisierte Deployment-Pipeline:

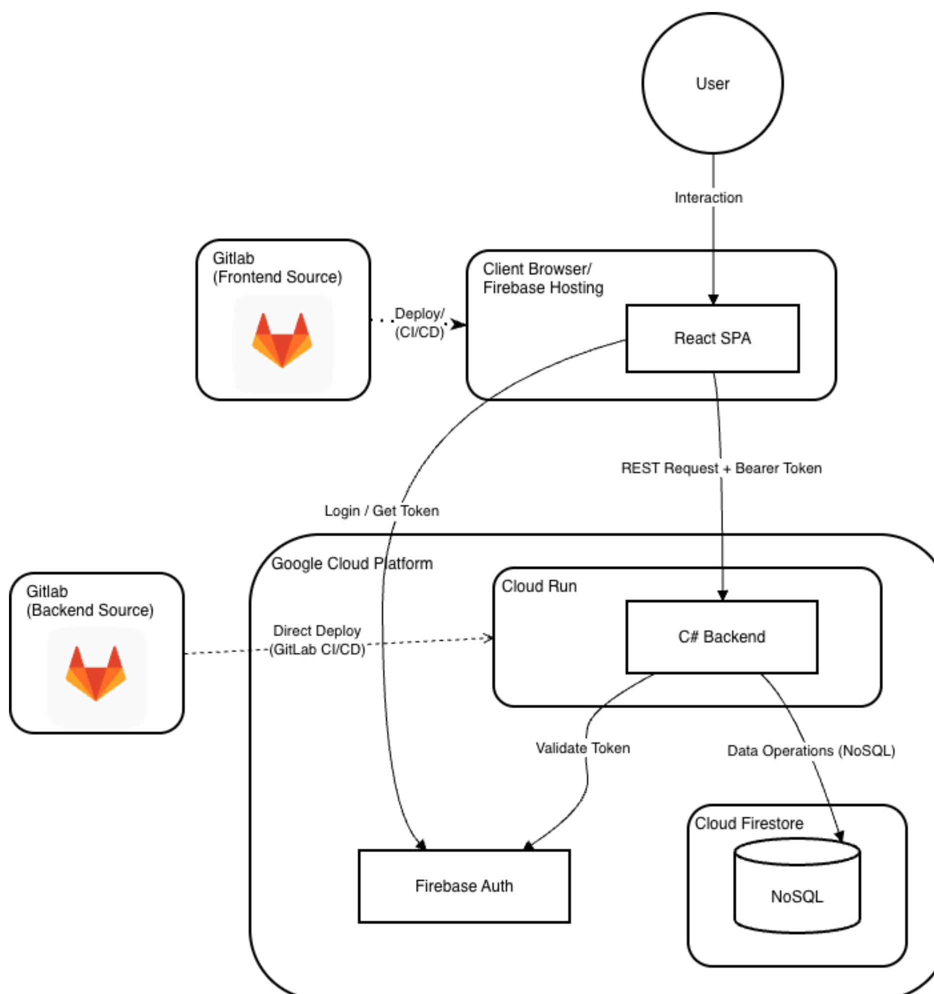

Abbildung 1: Systemarchitektur mit Cloud Run Backend, Firestore NoSQL-Datenbank und CI/CD-Flow via GitLab und GitHub.

a. Präsentationsschicht (Frontend-Spezifika)

Die Struktur des React-Frontends ist darauf ausgelegt, die Ansichten (Views) strikt von der Zustandsverwaltung (State) zu trennen.

- Verzicht auf UI-Bibliotheken (Styling): Anders als viele Webprojekte verzichten wir bewusst auf schwergewichtige UI-Frameworks wie Material-UI, Bootstrap oder Tailwind CSS. Das Styling wurde von Grund auf selbst mittels CSS Modules

implementiert (z. B. Calendar.module.css). Begründung: Dies verhindert nicht nur das unnötige Aufblähen der Anwendung, sondern gewährt dem Team die maximale Kontrolle über das visuelle Design und das Verhalten der Komponenten, ohne sich in fremde Framework-Spezifika einarbeiten zu müssen.
- Lokales State-Management: Für den Zustand der Anwendung (State) verzichten wir auf komplexe globale State-Container wie Redux. Begründung: Für die Anforderungen des Prototyps wäre Redux „Over-Engineering“. Wir nutzen stattdessen die in React integrierten Standard-Hooks (useState, useEffect). Um die Komponenten übersichtlich zu halten, wurde die Logik in eigene Custom Hooks abstrahiert (ein gutes Beispiel hierfür ist der Hook useCalendarTasks). Dieser Hook kapselt die Logik zum Abrufen, Verwalten und Aktualisieren der Aufgaben im Kalender und stellt sie der View-Komponente sauber zur Verfügung.
- Deployment via Firebase Hosting: Die fertige SPA wird über Firebase Hosting bereitgestellt. Der Deployment-Prozess ist dabei zweistufig automatisiert: Der Quellcode wird in GitLab verwaltet, zu GitHub gespiegelt und von dort aus mittels GitHub Actions final auf die Firebase-Infrastruktur ausgerollt.

b. Geschäftslogik und Datenzugriff (Backend-Spezifika)

Das C#-Backend ist das Herzstück der Anwendung und wurde konsequent nach der 3-Schichten-Architektur (Controller-Service-Repository-Pattern) strukturiert. Diese Trennung der Zuständigkeiten (Separation of Concerns) ist essenziell für die Wartbarkeit. Das Backend wird als Container-Instanz direkt aus GitLab heraus auf Google Cloud Run deployed.

- Die Controller-Schicht (Präsentation für APIs): Die Controller (z.B. TaskController) sind die Eintrittspunkte der Applikation. Ihre einzige Aufgabe ist das Routing. Sie nehmen HTTP-Requests entgegen, leiten die Parameter an die zuständigen Services weiter und wandeln die Ergebnisse in standardisierte HTTP-Responses (wie 200 OK oder 400 Bad Request) um. Hier findet keine Geschäftslogik statt.
- Die Service-Schicht (Geschäftslogik): In der Service-Schicht ist die zentrale Geschäftslogik, insbesondere der Planungsalgorithmus, verortet. Um die grundsätzlichen Struktur- und Entwurfsentscheidungen so zu gestalten, dass sich neue Entwickler leicht einarbeiten und den Code anpassen können, wird der Algorithmus nicht hart codiert (hard-coded). Stattdessen implementieren wir das Entwurfsmuster *Strategy Pattern*. Der Service kommuniziert ausschließlich über eine feste Schnittstelle mit der Berechnungslogik. Der Algorithmus zur automatischen Arbeitsplangenerierung ist als vollständig isolierter Service umgesetzt. Jedes Mal, wenn über den Controller eine neue Aufgabe hinzugefügt wird, wird dieser Service getriggert. Er holt sich die notwendigen Rahmendaten, führt die Neuberechnung der freien Zeiten aus und übergibt die aktualisierten Daten an die darunterliegende Datenschicht. Für den aktuellen Prototyp wird über diese Schnittstelle als erste Idee des Algorithmus eine vereinfachte Strategie

injiziert, die Aufgaben testweise in einem kleinen Rahmen (z. B. für einen festen Tag) einplant.

- Die Repository-Schicht (Datenzugriff): Diese Schicht kapselt die gesamte Kommunikation mit der NoSQL-Datenbank (Firestore). Begründung: Wenn sich das Team in Zukunft entscheiden sollte, von Firestore auf eine SQL-Datenbank zu wechseln, muss lediglich die Repository-Schicht ausgetauscht werden; die Controller und Services bleiben davon völlig unberührt.

4.3. Implementierte Design Patterns im Backend

Um eine hohe Code-Qualität zu gewährleisten, wurden folgende Entwurfsmuster (Design Patterns) verbindlich im Backend etabliert:

- Strategy Pattern: Die Logik zur Arbeitsplangenerierung ist nicht hart im Backend verdrahtet, sondern wurde über eine Schnittstelle in austauschbare Strategie-Klassen ausgelagert. Nutzen: Dies ermöglicht eine hohe Flexibilität für zukünftige Entwicklungsstufen. Der aktuell stark vereinfachte Algorithmus des Wegwerf-Prototyps kann so später nahtlos gegen komplexere, KI-gestützte oder auf Nachhaltigkeit optimierte Planungsalgorithmen (z. B. als Premium-Feature) ausgetauscht werden, ohne dass die Kernarchitektur der Controller- oder Service-Schicht angepasst werden muss.
- Dependency Injection (DI): Abhängigkeiten werden nicht innerhalb einer Klasse hart erzeugt (new Repository()), sondern von außen über den Konstruktor injiziert. Nutzen: Dies entkoppelt die Klassen voneinander. Auch wenn aktuell noch keine Testautomatisierung gefordert ist, bereitet dieses Muster die Architektur optimal auf spätere Komponententests vor, da Abhängigkeiten bei Tests leicht durch "Mock-Objekte" ersetzt werden können.
- Data Transfer Objects (DTOs): Für die Kommunikation zwischen Client und Server werden spezielle DTO-Klassen (z. B. UpdateUserDto oder CreateTaskDto) verwendet. Nutzen: Sie stellen sicher, dass die internen Datenbankmodelle nicht nach außen „durchsickern“ und reduzieren die Payload-Größe, da exakt nur die Daten über das Netzwerk geschickt werden, die der Client auch benötigt.
- Extension Methods (Erweiterungsmethoden): Die Konfiguration der Services, Datenbankverbindungen und Authentifizierungs-Filter nimmt oft viel Platz ein. Diese Logik wurde in sogenannte Extension Methods (z. B. services.AddApplicationServices()) ausgelagert. Nutzen: Die Startdatei der Anwendung (Program.cs) bleibt dadurch extrem kurz, sauber und lesbar.
- Singleton Pattern: Zentrale Ressourcen, die den gesamten Lebenszyklus der Anwendung überdauern müssen – wie beispielsweise die Verbindungsinstanz zum Firestore-Datenbank-Client – werden nach dem Singleton-Muster registriert. Nutzen: Das System erstellt beim Start exakt eine Instanz dieser Verbindung, die für alle nachfolgenden Anfragen ressourcenschonend wiederverwendet wird. Dies verhindert Memory Leaks und überflüssigen Netzwerk-Overhead beim Aufbau von Datenbankverbindungen.