 Template

Januar 2023

**Über arc42**

arc42, das Template zur Dokumentation von Software- und Systemarchitekturen.

Template Version 8.2 DE. (basiert auf AsciiDoc Version), Januar 2023

Created, maintained and © by Dr. Peter Hruschka, Dr. Gernot Starke and contributors. Siehe <https://arc42.org>.

# Einführung und Ziele

## Aufgabenstellung

„Ein mittelständisches Unternehmen möchte die Prozesse zur Essensbestellung der MitarbeiterInnen automatisieren. Bisher mussten die MitarbeiterInnen monatlich eine Excel Tabelle ausfüllen und diese entweder an der Kantine abgeben oder per Mail an essensbestellung@fancy-factory.de schicken.“

Erstelle eine Webanwendung, zur Automatisierung der Essenbestellungen, so soll es möglich sein, dass Mitarbeiter aus einem festgelegten Speiseplan bestellen können und diesen Einsehen.

## Qualitätsziele

**Funktionale Anforderungen:**

* mögliche Integration von Screen Readern
* Kalenderanbindung (Speisekarte)
* Filterfunktionen für Gerichte (Vegetarisch, Vegan oder nach Allergien etc.)
* Rollensystem (Mitarbeiter, Professoren, Studenten etc.)
* QR-Code zum Ausbuchen/abholen von Gerichten
* Monatliche oder Wöchentliche Übersichten (für User)
* Übersicht über alle Bestellungen (Mitarbeiter)
* Menüvorschläge
* Benachrichtigung bei neuen Menüplänen

**Nicht funktionale Anforderungen:**

* Sprache wählbar
* voll responsiv
* hohe Kontraste / mögliche Farbwahl
* Mobile First umgesetzt
* Open Source: muss kostenlos sein und frei nutzbar (Lizenz)
* Kompatibilität: muss mit Datenbank und Frontend kompatibel sein
* Betreibbar in einem Container
* Sicherheit: Das Backend sollte sicherstellen, dass alle eingehenden Anfragen validiert und authentifiziert werden.
* Zuverlässigkeit: Das Backend sollte robust sein und angemessene Mechanismen zur Fehlerbehandlung und Protokollierung haben, um Ausfälle zu minimieren
* Wartbarkeit: Der Code des Backends sollte gut dokumentiert sein
* Qualität: Best Practices sollen verwendetet werden, damit die codequalität sicher gestellt werden kann
* Gute Performance, es sollten mehrere Hunderte Lese- und Schreibvorgänge in wenigen Sekunden verarbeitet werden können.
* Datensicherheit: Datenbank sollte selbst auf eigenen Servern betrieben werden können

## Stakeholder

| Rolle | Kontakt | Erwartungshaltung |
| --- | --- | --- |
| *Product Owner* | *Markus Völk* | *funktionsfähige Software, strukturiertes Vorgehen* |
| *Developer*  Hungernde  Kantinenmitarbeiter | *Marcel Hasselberg*  -  - | *Gute Wartbarkeit/Strukturierung*  *Reibungslose/Intuitive Essensbestellung*  Automatisierte Reibungslose Essenbestellung |

# Randbedingungen

Organisatorisch:

* Zeitraum von 2 Monaten zur Entwicklung
* Zeitliche Einschränkungen des Entwicklerteams

Technisch:

* WebApp (Desktop und Smartphone)
* Bestellung bis Donnerstag 18:00 Uhr möglich
* Öffentlicher Zugriff möglich

# Kontextabgrenzung

## Fachlicher Kontext

****

Die Software soll keine Möglichkeit zur Zahlung der Bestellungen haben.

## Technischer Kontext

Die gesamte Anwendung läuft auf einem Linux Server, der eine öffentliche IP-Adresse hat. Auf diesem läuft dann ein Docker Compose Netzwerk, in welchem sich die drei Container befinden. Hierbei werden nur der Frontend und der Backend Container von außen erreichbar gemacht, indem ihre Ports an die öffentliche IP-Adresse weitergeleitet werden.   
Im Backend Container befinden sich die ganzen API’s mit welchen auf die Datenbank zugegriffen wird. Deshalb ist es nicht nötig den Datenbank Container nach außen sichtbar zu machen, denn innerhalb des Docker Compose Netzwerks können alle Container untereinander kommunizieren. So ist es möglich das über die API’s im Backend Container Daten von der Datenbank geholt und geändert werden können.   
Der Client holt sich also vom Frontend Container die benötigten Ressourcen wie die Benutzeroberfläche. Wenn der Client dann Daten von der Datenbank anfordert, werden die Datenbankinhalte, über die API die im Backend Container definiert ist bereitgestellt.

Beispiel anhand des logins:

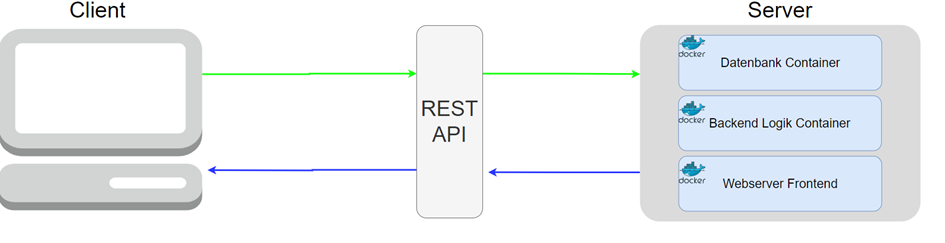
1. Client-Anfrage: Der Benutzer ruft die Login-Seite auf (IP:80/login.html) und gibt seine Anmeldeinformationen ein.
2. Der Frontend Container empfängt die Anfrage und gibt dem Client login.html zurück
3. Eingabe von Anmeldeinformationen: Der Benutzer gibt seine Anmeldeinformationen ein und sendet das Formular ab.
4. POST-Anfrage an Backend: Der Browser des Clients sendet eine POST-Anfrage mit den Anmeldeinformationen an die API im Backend-Server.
5. Backend-Verarbeitung: Der Backend-Server überprüft die Anmeldeinformationen und führt die Authentifizierungslogik durch, indem er auf die Datenbank zugreift und prüft, ob der Nutzer mit diesem Passwort existiert.
6. Der Backend-Server sendet eine Antwort an den Client. Diese beinhaltet bei erfolgreichem Login einen JWT-Token mit welchem der Client bei folgenden Anfragen erkannt wird und sich so auf bestimmte Dinge autoriesen kann.

# Lösungsstrategie

**Deployment:**

1. Verbindung auf den Server mittels ssh
2. Aktuellen Stand mit git pull ziehen
3. Die Docker Compose Umgebung neu starten

**Architektur:**

****

Clientseite:

Auf dem Client werden folgende Komponenten betrieben:

* Die Weboberfläche, die dem Benutzer die Interaktion mit dem System ermöglicht.
* Die Frontend-Logik, die die Benutzerinteraktion steuert. Dazu gehört die Verarbeitung von Aktionen wie das Zusammenfassen von Gerichten im Warenkorb zu einer Bestellung und deren Übermittlung als JSON an den Server. Ebenso wird die Verwaltung von Benutzereinstellungen und die Anforderung von Bestellhistorien hier abgewickelt.

Die Entscheidung, die Weboberfläche und die Frontend-Logik auf dem Client zu betreiben, bietet den Benutzern eine reaktionsschnelle und benutzerfreundliche Benutzererfahrung.

Serverseite:

Die Serverkomponenten sind in Docker-Containern organisiert, um eine einfache Austauschbarkeit und Bereitstellung zu gewährleisten. Dies ermöglicht die Ausführung der Container auf verschiedenen Arten von Servern. Außerdem erleichtert es dem Entwicklerteam die agile Entwicklung der Anwendung.

Kommunikation zwischen Containern: Die genaue Kommunikationsstrategie zwischen den Containern wird noch definiert. Eine potenzielle Technologie zur Verwaltung des Container-Netzwerks und zur Kontrolle der Container ist Docker Compose. Hiermit können Container orchestriert und verwaltet werden.

Serveranwendungen: Auf dem Server laufen folgende Anwendungen in separaten Containern:

* Datenbank-Container: Enthält die PostgreSQL-Datenbank und ihre Daten.
* Backend-Logik-Container: Hier befindet sich die gesamte Backend-Logik, die für die Verarbeitung von Anfragen vom Client und die Bereitstellung von Daten zuständig ist.
* Webserver-Frontend-Container: Ein Container, der einen Webserver ausführt auf welchem Angular gehostet wird. Dies ist typischerweise ein Nginx, Node.js oder Apache-Server. Der Webbrowser des Clients erstellt beim Zugriff dann eine HTTP-Anfrage an diesen Webserver, um die Ressource (HTML-Datei, CSS-Datei, JavaScript-Datei usw.) anzufordern, die für die angezeigte Webseite benötigt wird. Der Webserver soll dann dem Client die angeforderten Ressourcen bereitstellen. Der Webbrowser des Clients empfängt die HTTP-Antwort des Servers und rendert den Inhalt entsprechend.

REST-API:

Die Kommunikation zwischen Client und Server erfolgt über eine REST-API.

Die REST-API wurde gewählt, da sie eine einfache, plattformunabhängige und skalierbare Möglichkeit bietet, Daten zwischen Client und Server auszutauschen. Durch die Verwendung von standardisierten HTTP-Methoden. Die REST-API ermöglicht es dem Client, mit dem Server zu kommunizieren, Anfragen zu senden und Daten zu empfangen.

Zum Austausch von Daten wie Bestellungen oder Zusammenfassung soll das Datenaustauschformat JSON verwendet werden. Durch die Verwendung von JSON als Datenaustauschformat wird eine effiziente Übertragung von Informationen ermöglicht, was zu einer verbesserten Leistung und Interoperabilität führt.

**Datenbank (MariaDB / PostgreSQL / MySQL):**

Alle drei Datenbanken sind Industrie Standard und werden da auch benutzt. Im gesamten sind alle drei Relationale Datenbanken von der Funktionalität sehr ähnlich. Kurz gesagt mit jeder dieser drei Datenbanksystemen könnte unser Projekt gut umgesetzt werden. Wir haben uns jedoch für PostgreSQL entscheiden, da es schon Vorerfahrung im Team gibt, es gute Open-Source Lizenzen und eine Community hat. Falls das Projekt mal weiterentwickelt wird bietet PostgreSQL noch viele weitere Komplexe Funktionalitäten.

**Backend (Flask, Django):**

Beide Frameworks bringen ihre Vor- und Nachteile. Es ist jedoch nicht klar, ob die Vorteile von Django mit diesem Projekt vollkommen ausgenutzt werden können. Die Anfängerfreundliche Entwicklung mit Flask ist allerdings ein sehr großer Vorteil, da nicht alle Mitglieder des Teams Vorerfahrungen mit Python haben.   
Somit ist unser Framework für das Backend Flask.

**FrontEnd (React.js, Vue.js, Angular):**

Letztendlich hat sich die Auswahl durch die bisherigen Erfahrungen des Entwicklers für die Kombination aus Angular und Flask entschieden. Dies bietet eine leistungsstarke Lösung für die Entwicklung moderner Webanwendungen. Durch die Integration dieser beiden Frameworks können Entwickler eine klar strukturierte, skalierbare und performante Anwendung entwickeln. Fortführend wird auch eine klare Trennung zwischen Front- und Backend, durch eine Anbindung Angular durch eine Restful API von Flask, sichergestellt. Dahingehend werden Skalierbarkeit und Wartbarkeit der Anwendung weiter unterstützt.

# Bausteinsicht

## Whitebox Gesamtsystem

***<Übersichtsdiagramm>***

Begründung

*<Erläuternder Text>*

Enthaltene Bausteine

*<Beschreibung der enthaltenen Bausteine (Blackboxen)>*

Wichtige Schnittstellen

*<Beschreibung wichtiger Schnittstellen>*

### <Name Blackbox 1>

*<Zweck/Verantwortung>*

*<Schnittstelle(n)>*

*<(Optional) Qualitäts-/Leistungsmerkmale>*

*<(Optional) Ablageort/Datei(en)>*

*<(Optional) Erfüllte Anforderungen>*

*<(optional) Offene Punkte/Probleme/Risiken>*

### <Name Blackbox 2>

*<Blackbox-Template>*

### <Name Blackbox n>

*<Blackbox-Template>*

### <Name Schnittstelle 1>

…

### <Name Schnittstelle m>

## Ebene 2

### Whitebox *<Baustein 1>*

*<Whitebox-Template>*

### Whitebox *<Baustein 2>*

*<Whitebox-Template>*

…

### Whitebox *<Baustein m>*

*<Whitebox-Template>*

## Ebene 3

### Whitebox <\_Baustein x.1\_>

*<Whitebox-Template>*

### Whitebox <\_Baustein x.2\_>

*<Whitebox-Template>*

### Whitebox <\_Baustein y.1\_>

*<Whitebox-Template>*

# Laufzeitsicht

## *<Bezeichnung Laufzeitszenario 1>*

* <hier Laufzeitdiagramm oder Ablaufbeschreibung einfügen>
* <hier Besonderheiten bei dem Zusammenspiel der Bausteine in diesem Szenario erläutern>

## *<Bezeichnung Laufzeitszenario 2>*

…

## *<Bezeichnung Laufzeitszenario n>*

…

# Verteilungssicht

## Infrastruktur Ebene 1

***<Übersichtsdiagramm>***

Begründung

*<Erläuternder Text>*

Qualitäts- und/oder Leistungsmerkmale

*<Erläuternder Text>*

Zuordnung von Bausteinen zu Infrastruktur

*<Beschreibung der Zuordnung>*

## Infrastruktur Ebene 2

### *<Infrastrukturelement 1>*

*<Diagramm + Erläuterungen>*

### *<Infrastrukturelement 2>*

*<Diagramm + Erläuterungen>*

…

### *<Infrastrukturelement n>*

*<Diagramm + Erläuterungen>*

# Querschnittliche Konzepte

## *<Konzept 1>*

*<Erklärung>*

## *<Konzept 2>*

*<Erklärung>*

…

## *<Konzept n>*

*<Erklärung>*

# Architekturentscheidungen

# Qualitätsanforderungen

**Weiterführende Informationen**

Siehe [Qualitätsanforderungen](https://docs.arc42.org/section-10/) in der online-Dokumentation (auf Englisch!).

## Qualitätsbaum

## Qualitätsszenarien

# Risiken und technische Schulden

# Glossar

| Begriff | Definition |
| --- | --- |
| *<Begriff-1>* | *<Definition-1>* |
| *<Begriff-2* | *<Definition-2>* |

Archiv

Deployement unter AWS:

Ein Bild, das Text, Diagramm, Screenshot, Plan enthält.

Automatisch generierte Beschreibung