Wochenplansoftware - Architekturbewertung

DIMITRI MEIER, SAEED SHANIDAR, JAN DIECKHOFF

Analyseergebnisse – Userservice (1/6)

- **Funktionen**
  - Synchronisation mit der PersonalDB über AMQP
  - Login und Authentifikation
- **Soll als Node.js App implementiert werden**
  - Eigenentwicklung nicht unbedingt notwendig
  - Zum Vergleich: Wir verwenden LDAP zur Authentifizierung und Setzen von Berechtigungen
  - Für RabbitMQ gibt es auch ein LDAP-Plugin
  - Vorteile: bestehende Software wird verwendet, keine Implementation notwendig
  - Nachteil: Eventuell mühsame Konfiguration der LDAP-Komponente in Verbindung mit RabbitMQ, da mehrere Docker Swarms verwendet werden (Synchronisation)

Analyseergebnisse – Customerservice (2/6)

- **Funktion**
  - stellt im Wesentlichen ein Abbild der benötigten Daten aus dem SugarCRM zur Verfügung
  - Synchronisation mit dem SugarCRM über AMQP
- **Soll als Phoenix App in Elixir implementiert werden**
  - Vorteile: Phoenix ist performanter als Node.js
  - Nachteile: System kann schwer wartbar/erweiterbar werden, wenn zu viele unterschiedliche Sprachen/Frameworks verwendet werden.
- **Nutzen?**
  - SugarCRM hat eine REST-API -> Abbild scheint von geringem Nutzen zu sein
  - Man kann stattdessen direkt die API von SugarCRM ansprechen
  - Nutzen max. für höhere Ausfalltoleranz -> SugarCRM nicht erreichbar

Analyseergebnisse – Plan-UI(3/6)

- **Funktion**
  - Web-App zum Erstellen (Plan-Service) und Lesen der Pläne (Plan-DB)
- **Solls als RoR-App implementiert. Twitter Bootstrap und jQuery für das Frontend**
  - Plan-UI als Webapp zu implementieren ist fraglich, da sie eh nur aus dem Intranet erreichbar ist
  - Eigenständig lauffähige Softwarekomponente mit grafischer Benutzungsoberfläche (z. B. Mit Qt) eignet sich evtl besser.
  - Von Außen: Prüfung ob Client auf dem die Plan-UI gestartet wird sich im Intranet befindet.

Analyseergebnisse – Plan-Service (4/6)

- **Funktion**
  - Teilautomatisierte Erstellung der Pläne mittels Blackboard Agenten
- **Der Planungs-Service wird als NodeJS App in Javascript implementiert**
  - Das gewählte Framework scheint sinnvoll, da sich z. B. Über „express“ leicht REST-APIs erstellen lassen
  - Node.js ist weit verbreitet und leicht zu erlernen
  - Bei Entwicklerwechsel keine Experten für bestimmte Sprachen wie Erlang/Elixir nötig.
- **Blackboard – Agenten**
  - Backboard-Agenten zum Erkennen und lernen von Mustern aus den Eingaben der Planer erscheint sinnvoll
  - Plan-Service lernt damit, passendere Vorschläge zu wiederkehrenden Eingabemustern zu machen
  - Weniger Nachbearbeitung notwendig

Analyseergebnisse – Plan-DB(4/6)

- **Funktion**
  - Speicherung der erstellten Pläne
- **Soll PostgreSQL Datenbank umgesetzt werden (mit RoR-Wrapper)**
  - Vorteile:
    - Mit wenig Aufwand lassen sich SQL-Statements umsetzen
    - Es gibt viele gems die den Zugriff auf Datenbanken sehr einfach machen.
    - Ein Wrapper erzeugt eine gute Kapselung
  - Nachteil:
    - Zusatzaufwand
    - Mehr Hardwareresourcen
    - Delayzeiten
  - Aber prinzipiell ist das eine gute Entscheidung

Analyseergebnisse – E-Mailservice (5/6)

- **Funktion**
  - Senden von Emails an Servicekräfte
- **Soll als Node.js App implementiert werden**
  - Vorteil: Nodemailer
  - Nachteil: -

Analyseergebnisse – External API (6/6)

Architekturbewertung – Allgemein (1/2)

- ***Ist der Entwurf vollständig?***
  - Nein, die Verteilungssicht stellt eher eine Bausteinsicht des Systems dar
  - Somit fehlt die Verteilungssicht
  - Laufzeitsicht und Use-Cases fehlen
- ***Erfüllt der Entwurf alle funktionalen Anforderungen?***
  - Es werden alle funktionalen Anforderungen erfüllt
  - Die einzelnen Komponenten/Services erledigen alle Aufgaben, die aus den Anforderungen hervorgehen.
- ***Erfüllt der Entwurf alle nichtfunktionalen Anforderungen?***
  - Verfügbarkeit soll 99,9 % betragen
  - Dies kann gewährleistet werden, durch 3 unabhängige Docker-Swarms in drei verschiedenen Zonen
  - Plan-DB wird jeweils gespiegelt

Architekturbewertung – Allgemein (2/2)

- ***Erscheint Ihnen der Entwurf angemessen komplex? Ist er zu kompliziert oder vereinfacht er zu stark?***
  - Der Entwurf schafft eine gute erste Übersicht über die Systemlandschaft
  - Allerdings werden einige Architekturentscheidungen zu schwach beschrieben
    - Betrieb von drei unabhängigen "Swarms" aus jeweils vier Servern in drei Zonen
      - Wo liegen diese Zonen? Wie weit liegen diese Zonen auseinander?
    - Wie Sieht es mit der Hardwareausstattung der Server aus?
    - System soll mit der Microservices Architektur entwickelt werden
      - Wie ist die Aufteilung der einzelnen Services auf jeweils 4 Servern geplant?
      - Wie findet die Synchronisation zwischen den einzelnen Swarms statt (Consul)?

Architekturbewertung – Allgemein (3/3)

- ***Erscheint Ihnen der Entwurf angemessen komplex? Ist er zu kompliziert oder vereinfacht er zu stark?***
  - Die Verfügbarkeit von 99,9 % ist die einzige beschriebene nicht-funktionale Anforderung
  - Die External-API-Komponente ist zu schwach beschrieben
    - Wie soll diese aussehen?
    - Gibt es oder soll es einen gesonderten Client, der von außen darauf zugreifen kann?
  - Zusammengefasst:
    - Der Entwurf vereinfacht teilweise zu stark.
    - An anderen Stellen ist er jedoch wieder zu komplex: z. B.: zu viele unterschiedliche Programmiersprachen.

Architekturbewertung – Allgemein (3/3)

- ***Lässt sich die Software ggf. in mehreren parallelen Teams entwickeln, oder ist die Kopplung der Komponenten zu eng?***
  - …
  - …
  - …

Architekturbewertung – Allgemein (3/3)

- ***Wo hat der Entwurf Vor- und Nachteile gegenüber Ihrem Entwurf aus der letzten Aufgabe ?***

- Vorteile
  - Planerstellung mittels Blackboard-Agenten
  - Verfügbarkeit von 99,9 % wird gewährleistet
  - Verwendete Programmiersprachen/Frameworks sind bereits bekannt
  - Sicherheitsaspekte wurden berücksichtigt (z. B. SSL-Verbindung, oAuth2)
- Nachteile
  - Login und Authentifizierung werden selbst implementiert
  - SugarCRM-Daten werden nicht direkt gelesen sondern über den Customerservice abgebildet.