Universität Rostock

Traditio et Innovatio

Serviceorientiertes Design für Internet-GIS

Dr.-Ing. Peter Korduan

Universität Rostock

Agrar- und Umweltwissenschaftliche Fakultät

Professur für Geodäsie und Geoinformatik

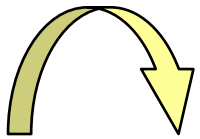
Universität Rostock Traditio et Innovatio

Inhalt

- Einleitung
- Wozu serviceorientiertes Design?
- Aufteilung in Dienste
- Schnittstellen
  - Joins
  - Multi-Gets
  - Versionen
- Konzepte für die Produktionsumgebung
  - Load-Balancing
  - Caching
  - Messaging
- Anwendungsbeispiel im Internet-GIS
- Zusammenfassung und Ausblick

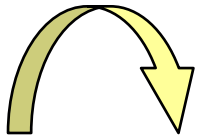
Einleitung

- Probleme bei Internet-GIS-Anwendungen
  - immer umfangreicher und komplexer
  - mehr Nutzer und Zugriffe
  - mehr schreibende Zugriffe
  - aufwendigere Prozesse



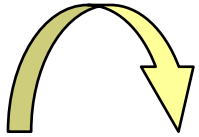
Schlechtere Performance

- Probleme bei der Softwareentwicklung
  - größere Entwicklerteams notwendig
  - Übersicht über Quellcode wird schwieriger
  - Tests dauern zu lange
  - Nachfrage nach wiederverwendbaren Komponenten steigt



Schlechte Skalierbarkeit, Entwicklungs-kosten steigen

- Datenquellen sind schon Dienste
- Auch Anwendung in Dienste aufteilen



Service-Orientiertes Design

Entwurfsziele

- Isolation

  Unabhängige Entwicklung, Optimierung und Testung
  - Business logic isolation

    getrennter Anwendungscode – gleiche Datenbasis
  - Shared system isolation

    getrennter Anwendungscode – eigene Datenbasis
  - Full isolation

    Auch getrennte Hardware oder Virtual Machine

- Robustness
  - Große Belastungen ohne Verlust von Funktionalität
  - Erreichbarkeit durch Kapselung, Zugriff über eine API
  - Beispiel: Datenbankzugriff über API

Entwurfsziele

- Scalability
  - Optimierung je nach Aufgabe möglich
  - Jeder Dienst separat optimierbar
  - Möglichkeiten für Optimierung völlig unterschiedlich
    - z.B. für Lesen, Schreiben, Rechnen
  - Beispiel:
    - Routinganfragen erfordern mehr Prozessoren und Backgroundprozessierung
    - mehr Daten erfordern mehr Festplattenplatz
    - Mehr Schreiboperationen erfordern Warteschlangen
  - Zahl der Entwickler skalierbar
  - Anzahl der Codezeilen überschaubarer

Entwurfsziele

- Agility
  - Änderung an einer Stelle wirkt sich wenig auf andere Stellen aus.
  - Kürzere Tests
  - Einfachere Aktualisierung der Produktionsumgebung
  - Schlüssel: Versionen für Schnittstellen
- Interoperability
  - Internal interoperability: Verknüpfung von Systemen in unterschiedlicher Programmiersprache
  - External nteroperability: Verknüpfung von Systemen unterschiedlicher Hersteller
- Reuse
  - Wiederverwendung von Diensten in gesamter Anwendung
  - z.B. Implementierung von räumlichen Funktionen wie Streckenberechnung

Aufteilung der Anwendung in Dienste

- Entscheidungsunterstützung
  - Welche Daten haben eine hohe Lese- und eine niedrige Schreibfrequenz?
    z.B.: WMS-GetMap, WFS-GetFeature
  - Welche Daten haben eine hohe Schreib- oder Änderungsfrequenz?
    z.B.: WFS-Transaction Insert und Update, Annotations
  - Welche Daten-Joins kommen häufig vor?
    z.B.: Flurstück – Eigentümer, Dokument - Geoname
  - Welche Teile der Anwendung haben ein klar definiertes Design und klare Anforderungen?
    z.B.: Legendenerstellung, Flächenberechnung

Schnittstellen (API)

- Möglichst einfach und unverändert
- Festlegung über Konvention (Data is the API)
- Typische Methoden für ressourcenorientierten Zugriff:
  - Create, GetMetaInformation, Get, Update, Delete
- Beispiele:
  - GET: http://139.30.111.16/honv/geoname/6
  - GET: http://139.30.111.16/honv/geoname/6.json
  - GET: http://139.30.111.16/honv/geonames/search/name/Rostock
  - POST: http://139.30.111.16/honv/geoname/6
- Parameter in Query-Strings in fester Reihenfolge
  => erhöht cache hit ratio

Joins

- Auch verschachtelte Ressourcen sollten einfach abzufragen sein
- Beispiele:
  - GET: http://documentarchiv.de/geonames/Rostock/docs/list
  - GET: http://localhost/gemeinde/13051040/gemarkungen
- Join sollte auf dem höchsten Level stattfinden
  - Im Application Server, nicht in Datenbank (habe ich mal anders gesehen)
  - Warum? Es sind viele Stellen wo die Joins auftreten könnten
  - Vermeidung von redundanten Anfragen
  - Beispiel
- Gründe
  1. Ermöglicht einfacheres Caching

     Services sollten nur Daten von ihrem eigenen Datenspeicher cachen => einfacheres Caching
  2. Auf Application Server Ebene ist es einfacher nur die Daten zu verknüpfen, die gebraucht werden (Anwendungslogik)

Multi-Gets

- Mehrere Ressourcen gleichzeitig anfragen
- Beispiel:
  - GET: http://139.30.111.16/honv/geonames?ids=1,2,3,4
  - GET: http://139.30.111.16/honv/geonames/list
  - GET: http://139.30.111.16/honv/geonames/last/5.json
- Nachteile:
  - Wie geht man mit Fehlern um?
  - Was ist, wenn eine Ressource nicht vorhanden ist?
    - Eine mögliche Lösung: Hash mit Ids als Keys und Single Resources als Values
  - Listen noch mal cachen? Nicht bei beliebigen Kombinationen von Parametern
- Häufige Multi-Gets können auch als Single Resource gesehen werden
  - Beispiele: Alle Eigentümer eines Grundstücks, die letzten 10 Kommentare zu einem POI, alle Bibliotheksdokumente zu einem Ortsnamen in MV

Versionierung von Schnittstellen

- Beispiele:
  - http://localhost/api/v1/honv/gename/rostock
  - http://localhost/honv/geoname/rostock?version=1
  - http://localhost/api/v2/honv/geoname/rostock
  - MIME type im Accept Header von HTTP

    GET /honv/geonames/rostock HTTP / 1.1

    Accept: application/mdi-de.org.honv-v1+json

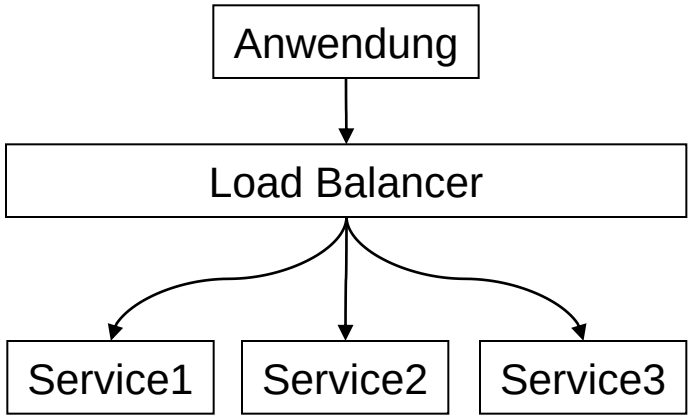
- Diskussion
  - Als Parameter ermöglicht optionale Angabe => immer Verwendung der aktuellsten Version
  - Update der Dienste führt zu Konflikt mit Clients, die keine Versionen angeben
  - Nicht alle Cache-Server berücksichtigen diesen Accept Header
  - Version in URI ist einfacher zu programmieren als im Header

Load Balancing

- Zur Verteilung der Last auf mehrere Prozesse und Server
  - Steigert die Zuverlässigkeit
  - Ermöglicht Skalierbarkeit
  - Verringert latency
- Horizontale Skalierung wird gut von Diensten unterstützt
- Software Balancer zwischen Anwendungsprozessen und Diensten
- Algorithmen zur Verteilung der Anfragen
  - Zufällige Zuweisung
  - Round-Robin
    - Gleichmäßige Umverteilung
  - Least-Connection
    - In Abhängigkeit von Belastung
  - URI-Based
    - Abhängig von URI

latency = $t_{\text{Ergebnis empfangen}} - t_{\text{Anfrage senden}}$
throughput = 1/latency

Vertikales Skalieren:
Steigerung der Ressourcen (z.B. Speicher, CPU)
Horizontales Skalieren:
Einsatz von mehr Servern

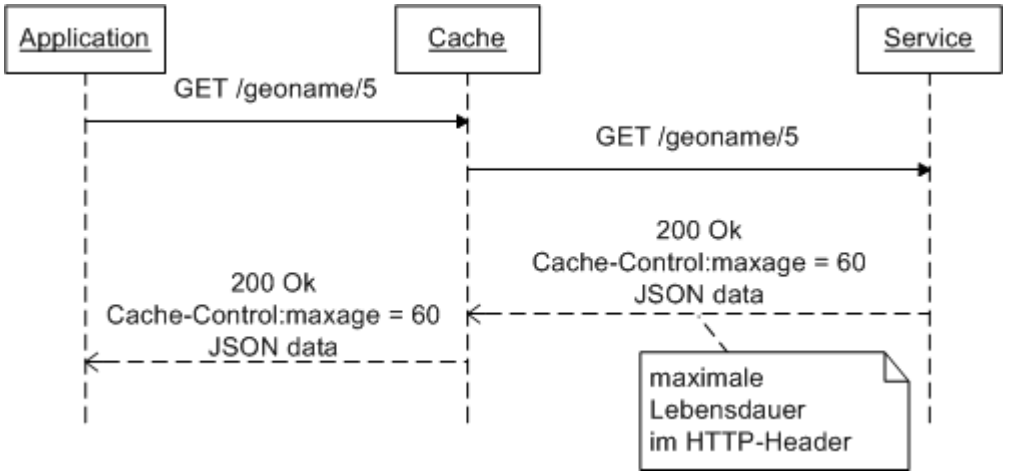


Caching

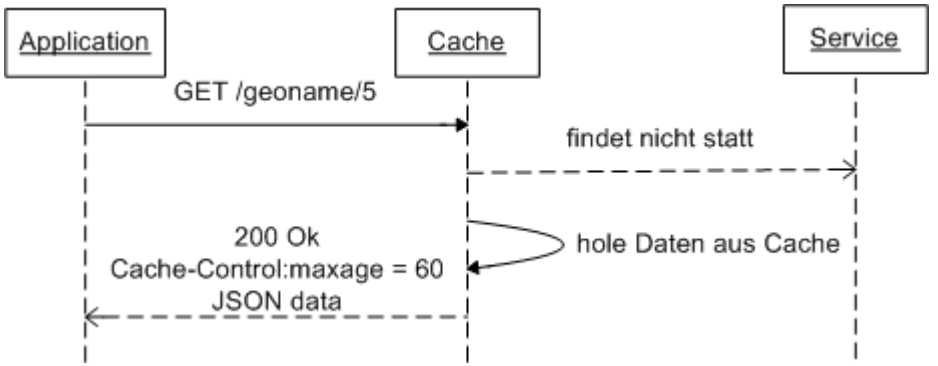
- Verringert die Zugriffszeit drastisch
- Erhöht den Durchsatz
- Implementierungen
  - Intern: Speichern von angefragten Daten im Diensteprozessspeicher
    z.B. berechneter Mittelpunkt eines Ortes
  - Extern: Caching-Unterstützung der HTTP-Spezifikation,
    besonders sinnvoll für eindeutige Ressourcen
  - Kombinationen aus beidem
- Ablauf der Gültigkeit
  - Zeitbasiert
  - Manuell
  - Schlüsselbasiert

Ablauf des Cache nach Zeit

1. Anfrage
   - Eintrag existiert nicht im Cache
   - Weiterleitung zum Dienst
   - Antwort im Cache gespeichert mit Zeitstempel
   - Antwort an Client



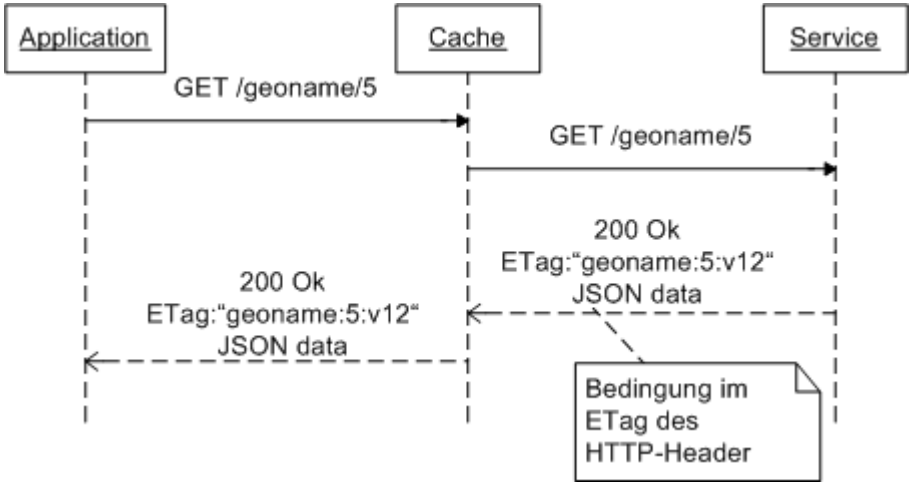
2. Anfrage
   - Eintrag existiert im Cache
   - Zeit ist noch nicht abgelaufen
   - Antwort an Client



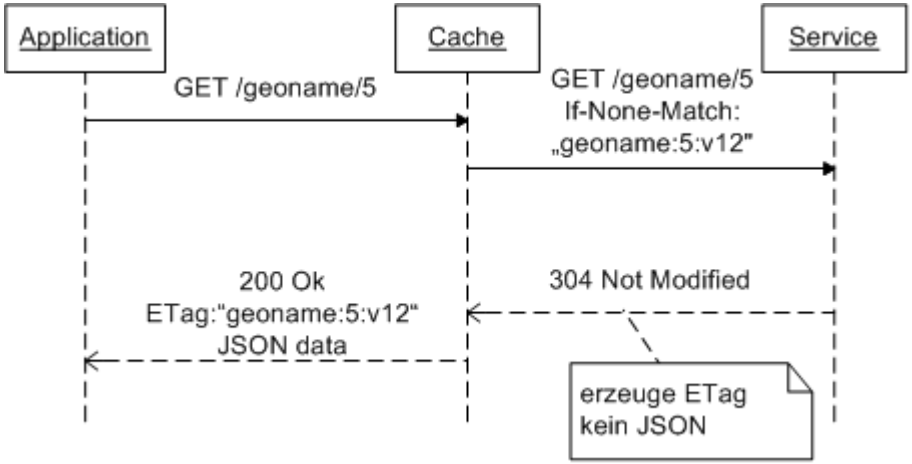
Grafiken geändert nach P. Dix 2000

Ablauf des Cache nach Bedingung

1. Anfrage
   - Eintrag existiert nicht im Cache
   - Weiterleitung an Dienst
   - Antwort im Cache speichern mit Versionsnummer
   - Antwort an Client



2. Anfrage
   - Eintrag existiert im Cache
   - Weiterleitung an Dienst
   - Keine neue Version, not modified im ETag zurück an Cache
   - Antwort an Client

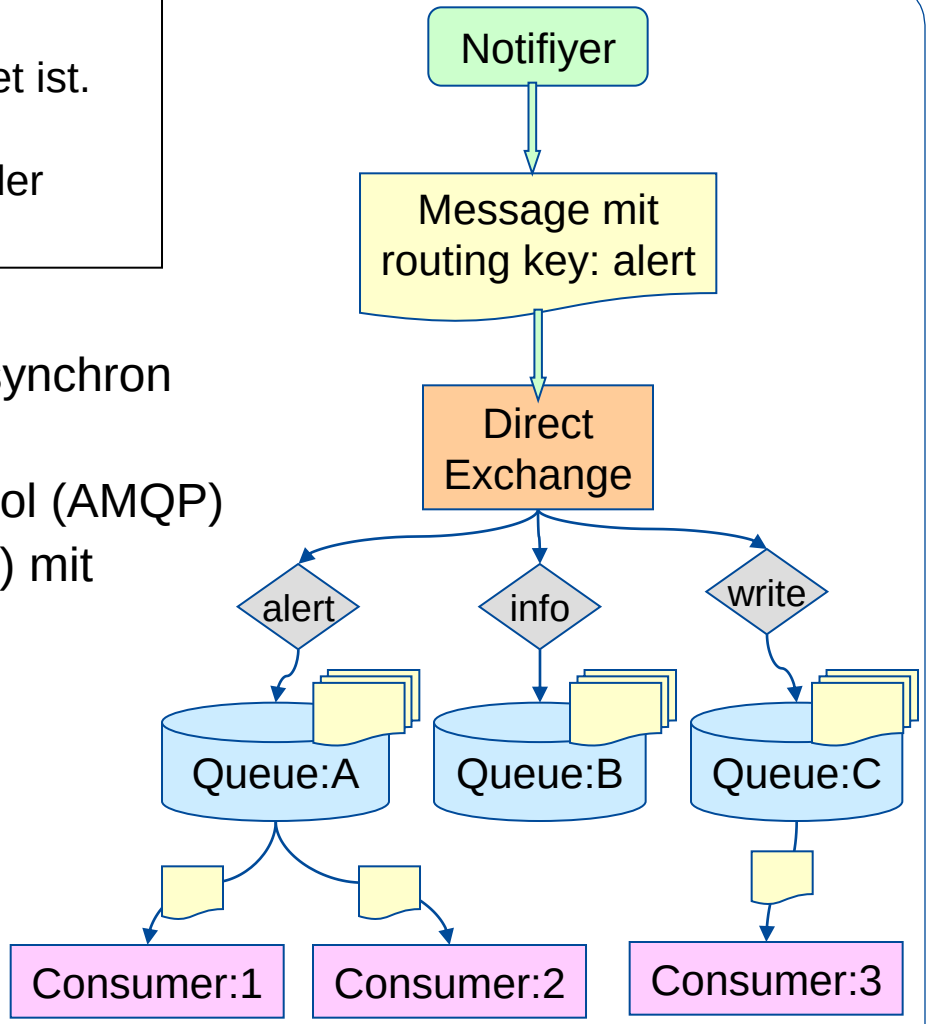


Grafiken geändert nach P. Dix 2000

Messaging für asynchrone Kommunikation

Synchron:
Client wartet solange bis Anfrage abgearbeitet ist.
Asynchron:
Beauftragung eines anderen Prozesses mit der Abarbeitung der Anfrage.

- Lesen geht schnell => synchron
- Schreiben und Rechnen dauert => asynchron
- SR/AW
- Advanced Message Queueing Protocol (AMQP)
- Methoden der Vermittlung (Exchange) mit RabbitMQ
  - Fanout Exchange
  - Direct Exchange
  - Topic Exchange
- Typische Anwendung für Write und Notification
- Inkonsistenzen separat behandeln!



Grafik geändert nach P. Dix 2000

Anwendungsbeispiel für Internet-GIS

- Wie müsste ein typischer Workflow für die Änderung eines Geodatensatzes aussehen.

- Client holt Karte (Kachel im Cache, dahinter MapServer, dahinter WFS)
- Client wählt Objekt in Karte über GetFeatureInfo und bekommt ID
- Client holt Feature über Link auf Resource GetFeature, die auch gecached ist (incl. updated_at).
- Änderung im Client
- Ggf. Vorvalidierung im Client (dazu ggf. DescribeFeature)
- Sendet geänderten Datensatz an MessageQueue (incl. updated_at).
- Updateserver prüft ob update_at noch stimmt und schreibt. Wenn nicht oder bei Fehlern sendet Bericht an Client zur Konfliktbehandlung.

GRAFIK zum Workflow wäre hilfreich

Zusammenfassung und Ausblick

- Service-Orientiertes Design hat viele Vorteile
  - der Wichtigste: Skalierbarkeit
- Trennung in Lese-, Schreib- und Berechnungsfunktionen
- einfache API‘s !!!
- Load Balancing und Caching für lesende Zugriffe
- Messaging für schreibende Zugriffe

- Es sind einige Überlegungen erforderlich!!!
- Keine Scheu vor der Aufteilung in Dienste.
- Frameworks für sehr schnelle Entwicklung von Diensten verfügbar.
- Schrittweise vorgehen, mit Data-API‘s beginnen.
- Erst Aufteilen, dann Optimieren.
- Es wird eine Weile dauern, aber mit jedem Dienst wird es einfacher.

Vielen Dank für die Aufmerksamkeit

- Gefördert durch das BMBF

Bundesministerium für Bildung und Forschung

- Hauptquelle für diesen Beitrag:

Dix, P. 2010: Service-oriented design with Ruby and Rails. Addison-Wesley Professional Ruby Series

- Weiteres zu Internet-GIS:

Korduan, P., Zehner, M.L. 2008: Geoinformation im Internet. Wichmann Verlag