

ERWEITERTE REST KONZEPTE UND ARCHITEKTUREN

Hochschule Hannover
University of Applied Sciences and Arts

FAKULTÄT IV – WIRTSCHAFT UND INFORMATIK
<http://f4.hs-hannover.de>


STEFAN NITZ

Erweiterte REST Konzepte und Architekturen

Erweiterte Konzepte

Resource-Naming, HATEOAS, Status-Codes & Co.

Resource-Naming

- Resource-Naming = Benennung von URI's 
- Wohl am häufigsten diskutiertes Thema im REST-Umfeld
 - ▣ Häufigste Debatte: „Warum sollte man sich überhaupt Gedanken über die Namensgebung von REST-API's Gedanken machen? So lange die URI eindeutig ist spielt der Name an sich keine Rolle, da per Hypermedia durch die API navigiert wird.“
- URI's sollten „vorhersagbar“ (**predictable**) und hierarchisch (**hierarchical**) sein
 - ▣ Trägt zum Verständnis und „Useability“ der API bei
 - ▣ **Vorhersagbar:** konsistente Namensgebung
 - ▣ **Hierarchisch:** Struktur und Beziehungen untereinander



DO!

- **Merke:** das ist kein MUSS für einen RESTful Webservice, lediglich eine Verbesserung der Lesbarkeit und Nutzbarkeit der API (letztendlich handelt es sich bei allen URI's um einfache Strings)

“Design for your clients, not for your data. As with everything in the craft of Software Development, naming is critical to success.” [2]

Resource-Naming Anti Patterns

- Falsche Verwendung der HTTP-Verben

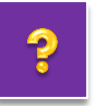
Falsche Update Operationen

- 1) GET `http://api.example.com/services?op=update_customer&id=12345&format=json`
- 2) GET `http://api.example.com/update_customer/12345`
- 3) GET `http://api.example.com/customers/12345/update`
- 4) PUT `http://api.example.com/customers/12345/update`



Repräsentationen

- Repräsentation = **Daten** + **Format** einer Ressource
 - ▣ Daten: Struktureller Aufbau (Verschachtelung, Datenformate, Hypermedia, ...)
 - ▣ Format: XML, JSON (auch andere Formate denkbar, dies sind aber die häufigsten Vertreter)
- Unterstützung verschiedener Repräsentationen
- Definition des Formats aus Client-Sicht:
 - ▣ (1) Durch den „Accept-Header“ von HTTP
 - Accept: application/json | application/xml
 - <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.1>
 - ▣ (2) Durch eine Art „File-Extension“
 - GET <http://shop.de/customers.xml> | <http://shop.de/customers.json>
 - GET <http://shop.de/customers?format=xml>
 - ▣ Variante 1 nutzt bereits vorhandenen HTTP-Header, somit *HTTP-konform* und „sauberer“
 - ▣ Variante 2 jedoch weit verbreitet in der *Industrie*, da das Format sofort ersichtlich ist und tools wie „curl“ einfacher zu bedienen sind



Repräsentationen: HATEOAS

- **HATEOAS** := Hypertext As The Engine of Application State
- Im Idealfall genügt eine URI (root) um eine API zu verstehen und zu nutzen
- Navigiert wird durch Hypermedia Links, die in den Repräsentationen enthalten sind
 - ▣ Es genügt also nicht nur die „reinen“ Repräsentationen (nur Daten) ohne weiterführende Links anzubieten

“Every addressable unit of information carries an address [...]. Query results are represented by a list of links with summary information, not by arrays of object representations.” *Fielding*.

Repräsentationen: HATEOAS

- Ressourcen (bzw. dessen Repräsentationen) sollten Hypermedia enthalten um auf andere Ressourcen zu verweisen
- Beispiel:

(1) **GET** <http://shop.de/customers>

```
[  
  {'href': 'http://shop.de/customers/4711'},  
  {'href': 'http://shop.de/customers/8011'},  
  {'href': 'http://shop.de/customers/9000'}  
]
```



Die Anfrage gibt eine Liste von Kunden zurück. Allerdings nicht dessen Roh-Daten sondern Links zu dessen Repräsentationen.

(2) **GET** <http://shop.de/customers/4711>

```
{  
  'name': 'Susi Strolch',  
  'age': '22',  
  'orders': 'http://shop.de/customers/4711/orders'  
}
```

Über den Link `/customers/4711` kann somit zu dem speziellen Kunden navigiert werden. Dieser enthält wiederum einen Link zu seinen Bestellungen.

Repräsentationen: HATEOAS

□ Vorteile

- Lose Kopplung zwischen Client und Server (einzige Bindung ist die „Root“-URI)
- Hypermedia wird in seiner Ursprungsform verwendet
- Im Idealfall kann die API beliebig angepasst werden, ohne dass ein Client davon betroffen ist (kein „brechen“ der API)
- Applikations-übergreifende Interaktion durch Hypermedia

Repräsentationen: HATEOAS

□ Nachteile

- Kann viel Netzwerk-Traffic erzeugen, wenn nur die reinen Links enthalten sind
 - Gerade bei mobilen Endgeräten stellt dies einen großen Nachteil dar
 - Daher gilt es eine Balance zwischen den reinen Daten und Verknüpfungen via Hypermedia zu finden; häufig wird eine Mischform umgesetzt.
- Steigerung der Komplexität bei der Implementierung
 - Ggf. hohes semantisches Verständnis der API notwendig
 - Ggf. aufwendiges rekursives auflösen von Links notwendig (ähnlich einem Crawler)

□ Empfehlung: In der Praxis wird häufig eine Mischform verwendet

Repräsentationen: HATEOAS

□ Mischform am gleichen Beispiel: GET <http://shop.de/customers>

```
[
  {
    'name': 'Susi Strolch',
    'age': '22',
    'links': [
      {
        'ref': 'self',
        'href': 'http://shop.de/customers/4711'
      }
    ]
  },
  {
    'name': 'Peter Pansen',
    'age': '38',
    'links': [
      {
        'ref': 'self',
        'href': 'http://shop.de/customers/8011'
      }
    ]
  },
  ...
]
```

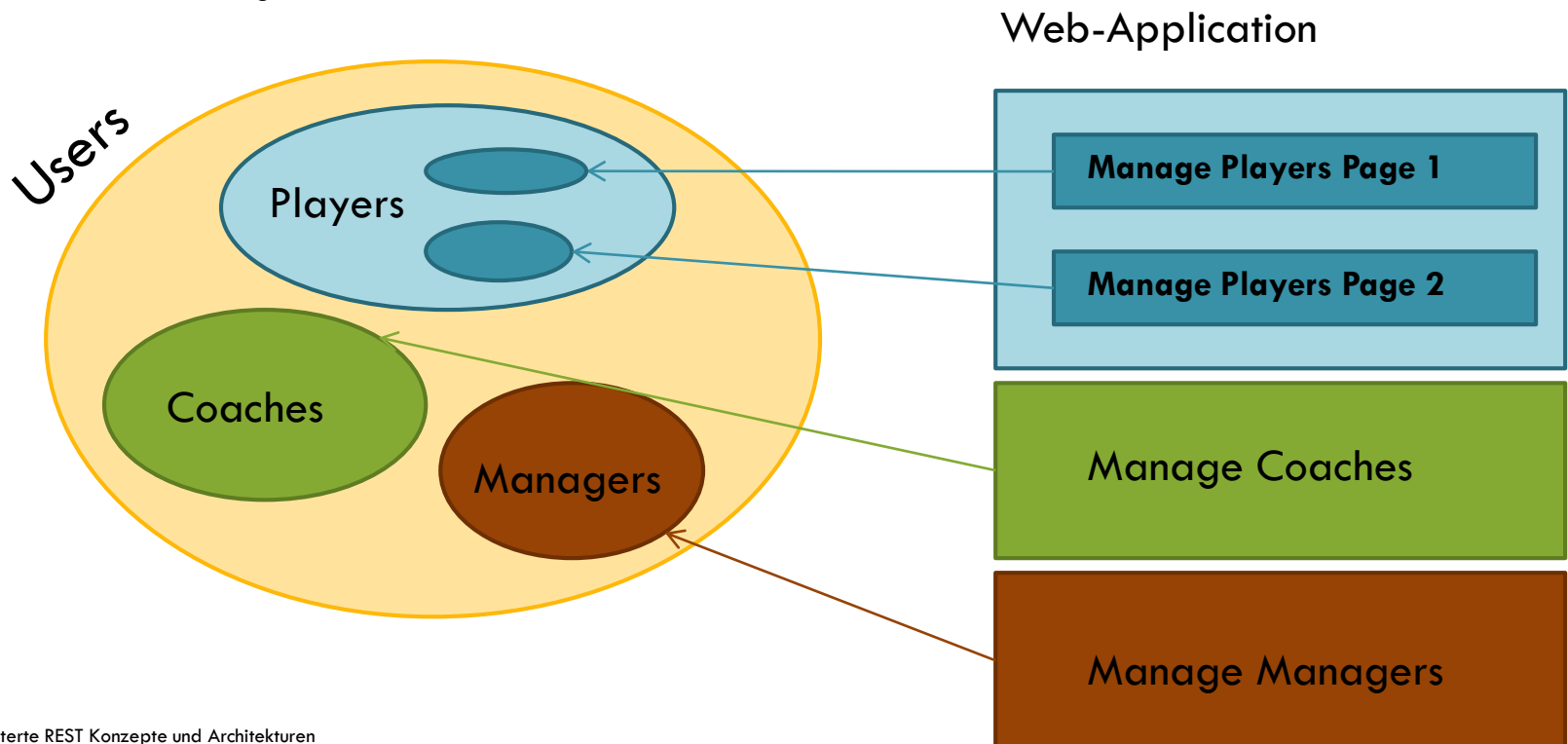
Mischform: Die Repräsentation enthält bereits die wichtigsten Attribute des Kunden. Für weitere Informationen wird eine Referenz auf “sich selbst” mitgeliefert.

Links: Der Aufbau des Links mit einem “ref” und “href” Attribut wird häufig verwendet und stammt aus dem “Atom Link Model” (siehe <http://www.xml.com/pub/a/2004/06/16/dive.html>).

Diese Datenstruktur ist eine *mögliche* Variante Verknüpfungen zu realisieren, da die verwendeten Attribute “ref” und “href” eine gewisse Semantik aus dem besagten Atom Link Model mitbringen.

Paginierung, Filterung, und Sortierung

- Generell: Techniken bzw. Mechanismen um die erwartete Datenmenge zu minimieren
 - ▣ Netzwerk-Traffic minimieren
 - ▣ Bandbreiten-schwachen Geräten (z.B. mobilen Endgeräten) die Möglichkeit bieten die Datenmenge einzuschränken




Paginierung

- Daten werden meist Seitenweise dargestellt mit der Option durch die Seiten zu „blättern“ (vor und zurück)
- Deshalb werden Daten aus Performanz-Gründen häufig „nachgeladen“
- Ähnlich dem „Cursor-Prinzip“ in Datenbanken muss es also einen Mechanismus zum navigieren durch die Datenmenge geben
- Zwei Möglichkeiten der Realisierung:
 - ▣ Query-String Parameter (offset/limit Ansatz)
 - *myshop.com/customers?offset=0&limit=25*
 - **Offset** gibt den Startwert in der Datenmenge an
 - **Limit** setzt die Anzahl der maximal zu erwartenden Datensätze
 - ▣ HTTP Range-Header
 - Request → Range: items=0-24
 - Response → Content-Range: items 0-24/66

Erweiterte Konzepte

Filterung

- Filtern bedeutet, bestimmte Datensätze anhand von nicht zutreffenden Kriterien aus der Datenmenge auszuschließen
- Beispiele:
 - ▣ „Gib mir alle Städte in Deutschland dessen PLZ mit 30 beginnt“
 - ▣ „Gib mir alle Produkte dessen Preis größer als 500€ ist“
- Einführung eines **filter** Query-String Parameters 
 - ▣ Key/Value Paare werden per „::“ definiert
 - ▣ Als Trennzeichen zwischen Attributen dient das Pipe-Zeichen „|“
 - ▣ Nicht Case-Sensitive (filtern nach *lastName=hansen* findet „Hansen“, „hansen“, „HanSeN“, ...)

GET <http://shop.de/customers>
?filter=last_name::hansen|age::12



Last Name	First Name	Age
Hansen	Hugo	12

Filterung

- Diese Definition eines Filters ist mit Absicht einfach gehalten
- Bezogen auf den jeweiligen Use-Case sind komplexere Filter-Möglichkeiten denkbar z.B.
 - ▣ Logische Operatoren (and, or, not equal, greater than, less than, ...)
 - ▣ Funktionen (endswith, startswith, concat, hour/minute/day...)
- Das standardisierte *Open Data Protocol* (OData) definiert z.B. eine spezielle Ausdruckssprache (ABNF Grammatik)
 - ▣ Beispiele:
 - `/Products?$filter=Price le 200 and Price gt 3.5`
 - `/Customers?$filter=startswith(CompanyName, 'Alfr') eq true`
 - `/Employees?$filter=day(BirthDate) eq 8`
 - ▣ <http://www.odata.org/documentation/uri-conventions/#FilterSystemQueryOption>

Sortierung

- Legt die Reihenfolge der einzelnen Entitäten der zurückgegebenen Daten fest
- Nicht zwingend erforderlich, da dies in der Theorie auch Client-seitig durchgeführt werden kann
 - ▣ Allerdings: Server-seitige Sortierung oft performanter, da Clients z.T. Ressourcen-arm sind (z.B. mobile Endgeräte)
- Einführung eines **sort** Query-String Parameters
 - ▣ Per default, Sortierung in aufsteigender Reihenfolge (ascending)
 - ▣ Ein Attribut mit einem „-“ als Prefix wird absteigend sortiert (descending)
 - ▣ Als Trennzeichen zwischen Attributen dient das Pipe-Zeichen „|“

GET `http://shop.de/customers`
`?sort=last_name | first_name | -age`



Last Name	First Name	Age
Hansen	Hugo	12
Stolch	Anne	40
Stolch	Susi	23

Versionierung von Services

- Services können sich im Laufe der Zeit durch neue Anforderungen ändern
- Eine Änderung an einem Service kann bestehende Konsumenten des Services „brechen“
 - Attribut-Namen können sich ändern (Clients basieren häufig auf bekannten Schemata; eine Änderung „bricht“ den Client)
 - Neue Attribute kommen hinzu (auch wenn viele Clients ggf. fehlertolerant sind, können andere Clients davon betroffen sein)
 - Inhalte von Attributen können sich ändern (kann zu Problemen bei vorhandener Validierung führen)



□ Versionierung per URI

- ▣ /api/<version-nr.>/customers → Bsp: `http://myshop.de/api/v2/customers`
- ▣ Viele „große“ API's wie Twitter oder Facebook nutzen dieses Konzept
- ▣ Allerdings fragwürdig hinsichtlich des REST Konzeptes
 - Es wird nicht das HTTP build-in Header-Feld zur Versionierung genutzt
 - Eigentlich sollte eine neue URI nur dann eingeführt werden, wenn eine neue Ressource eingeführt wird und nicht dessen Änderung des Schemas

“The URI should be simply to identify the resource--not its ‘shape’.”

- ▣ Häufig eingesetzt, da es in der Praxis einfach umzusetzen ist (sowohl für die Implementierung als auch zum Testen eines Services)

□ Versionierung per HTTP-Header

- ▣ Nutzung von **Accept** und **Content-Type**
 - ▣ **Accept** spezifiziert Media Typen die der Client als Response erwartet bzw. womit der Client umgehen kann
 - ▣ **Content-Type** wird sowohl vom Client als auch vom Server genutzt um das Format des Requests/Responses zu definieren
-
- ▣ **Beispiel version=1**

Request

```
GET http://myshop.de/cutomers/1234  
Accept: application/json; version=1
```

Response

```
HTTP/1.1 200 OK  
Content-Type: application/json; version=1  
{'id': '1234', 'name': 'Peter Paulinski'}
```

□ Versionierung per HTTP-Header

▣ Beispiel, version=2

Request

```
GET http://myshop.de/cutomers/1234  
Accept: application/json; version=2
```

Response

```
HTTP/1.1 200 OK  
Content-Type: application/json; version=2  
{ 'id': '1234', 'firstName': 'Peter', 'lastName': 'Paulinski' }
```

▣ **Merke:** Die URI ist für beide Aufrufe die gleiche!

□ Versionierung per HTTP-Header

- Es können auch mehrere Media-Typen im „Accept“ Header definiert werden:

Request

```
GET http://myshop.de/cutomers/1234
Accept: application/json; version=1,
       application/xml; version=1
```

- Der Server liefert eins der beiden Formate zurück, je nachdem welches er bevorzugt:

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; version=1
{'id': '1234', 'name': 'Peter Paulinski'}
```

Erweiterte Konzepte

HTTP Status Codes

- Definieren den Status einer Anfrage (Request) innerhalb einer Antwort (Response)
- Wichtig, damit der Client entscheiden kann wie er die Antwort verarbeitet
- Status Codes sind im HTTP-Protokoll verankert und somit standardisiert
 - ▣ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
 - ▣ Kategorien
 - 1xx: Meta
 - 2xx: Success
 - 3xx: Redirection
 - 4xx: Client-Side Error
 - 5xx: Server-Side Error
- Allerdings: Fehler-Codes können abhängig von der Service-Implementierung unterschiedliche Semantiken haben
 - ▣ Beispiel: **Service A** verwendet „200-OK“ für eine erfolgreiche Lösch-Operation (DELETE) wo hingegen **Service B** „204-No-Content“ dafür verwendet
 - ▣ Daher lohnt sich oft ein Blick in die Dokumentation des Services

Die häufigsten HTTP Status Codes

- **200** (OK) – Genereller Code für einen erfolgreichen Request.
- **201** (CREATED) – Ressource wurde erfolgreich angelegt (via POST oder PUT).
- **204** (NO CONTENT) – Wird häufig bei DELETE verwendet
- **304** (NOT MODIFIED) – Wird häufig bei Caching verwendet.
- **400** (BAD REQUEST) – Genereller Fehler wenn z.B. das Format nicht valide war, Datentypen nicht korrekt waren oder Daten im Request fehlen.
- **401** (UNAUTHORIZED) – Fehlercode wenn ein Token zur Authentisierung fehlt oder abgelaufen ist.
- **403** (FORBIDDEN) – Nicht autorisierte Anfrage an die Ressource (Token abgelaufen oder keine Rechte dafür).
- **404** (NOT FOUND) – Angefragte Ressource wurde nicht gefunden (oder der Service will 401/403 “verstecken”).
- **409** (CONFLICT) – Falls es zu einem Konflikt gekommen ist (z.B. haben mehrere Clients gleichzeitig versucht eine Ressource zu ändern oder z.B. ein “Root-Knoten” gelöscht werden soll, aber noch weitere Ressourcen unter diesem Knoten hängen).
- **500** (INTERNAL SERVER ERROR) – Genereller Serverfehler wenn z.B. die dahinter liegende Datenbank nicht verfügbar ist.

REST „Anti Patterns“

1. Alle Requests durch GET/POST tunneln

Beispiel: `http://example.com/some-api?method=deleteCustomer&id=123`

2. Caching ignorieren

- Server-Response: „Cache-control: no-cache“; führt Caching Ad-absurdum
- ETags können (und sollten) seit HTTP 1.1 verwendet werden, um anzuzeigen wie lange eine Ressource noch gültig ist

3. Status-Codes ignorieren

- Bereits vorhandene Semantik von HTTP Status-Codes sollte genutzt werden
- Beispiel: Nicht alle Fehler als „Status-Code 500“ zurückgeben und speziellen Fehlertext anhängen

4. Falsche Verwendung von Cookies

- In Ordnung für Informationen wie einem Authentication-Token
- Allerdings keine Informationen im Cookie hinterlegen, die nicht irgendwo bereits standardisiert sind (z.B. HTTP Basic Authentication verwenden anstatt User+Passw. im Cookie zu speichern)

5. Keine Verwendung von Hypermedia

- Representationen enthalten keine weiterführenden Links ☹️
- Idealerweise benötigt der Client nur eine einzige URI, der Rest geschieht durch „hypermedia“

Architekturen

Wo gliedert sich REST in die Architektur ein?

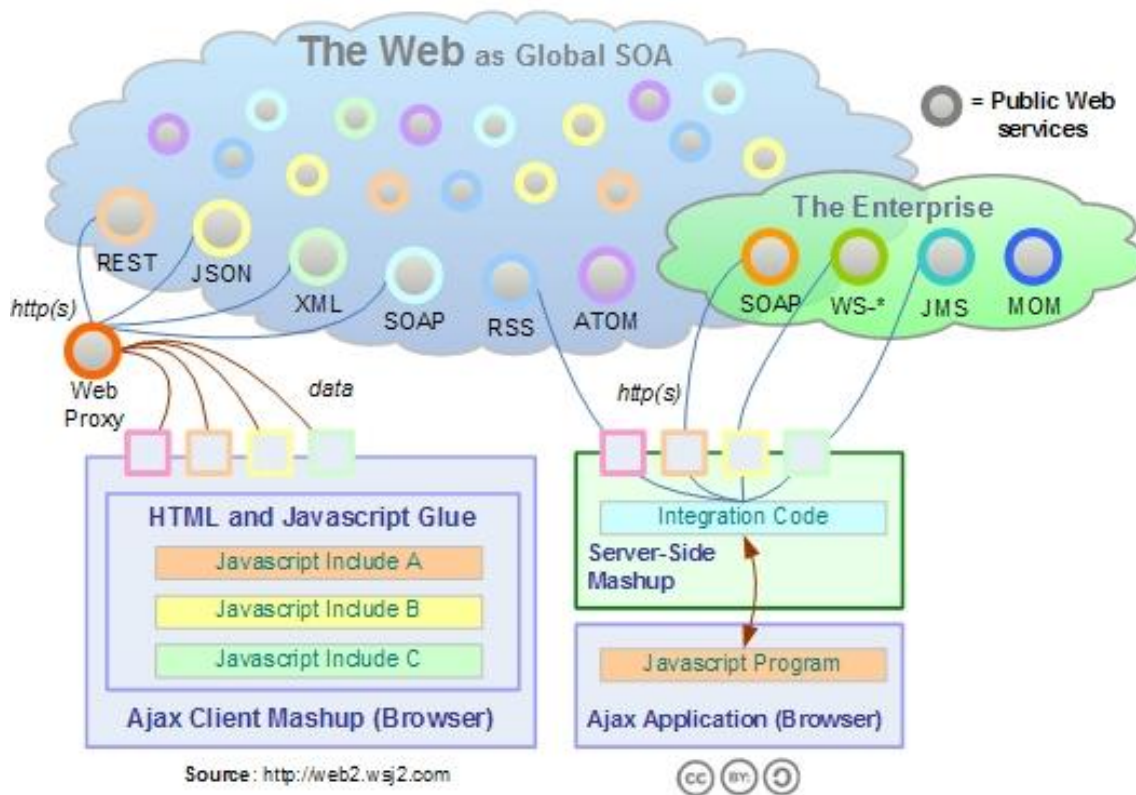
REST-Architekturen und Architekturebenen

- REST als Integrationsansatz (z.B. Web-Mashups)
- Integration von REST in die eigene Architektur
- Web-API Architektur
- Frontend-Architektur (Single Page Applications)

REST als Integrationsansatz (z.B. Web-Mashups)

Web Mashup Styles

In-Browser | Server-side



- REST als Integrationsansatz im Web

- Das Web als eine Art SOA implementiert mit RESTful Web-Services

Quelle: <http://blog.sherifmansour.com/wp-content/uploads/2007/11/webmashupstyles.jpg>

Integration eines REST-Services in die eigene Architektur



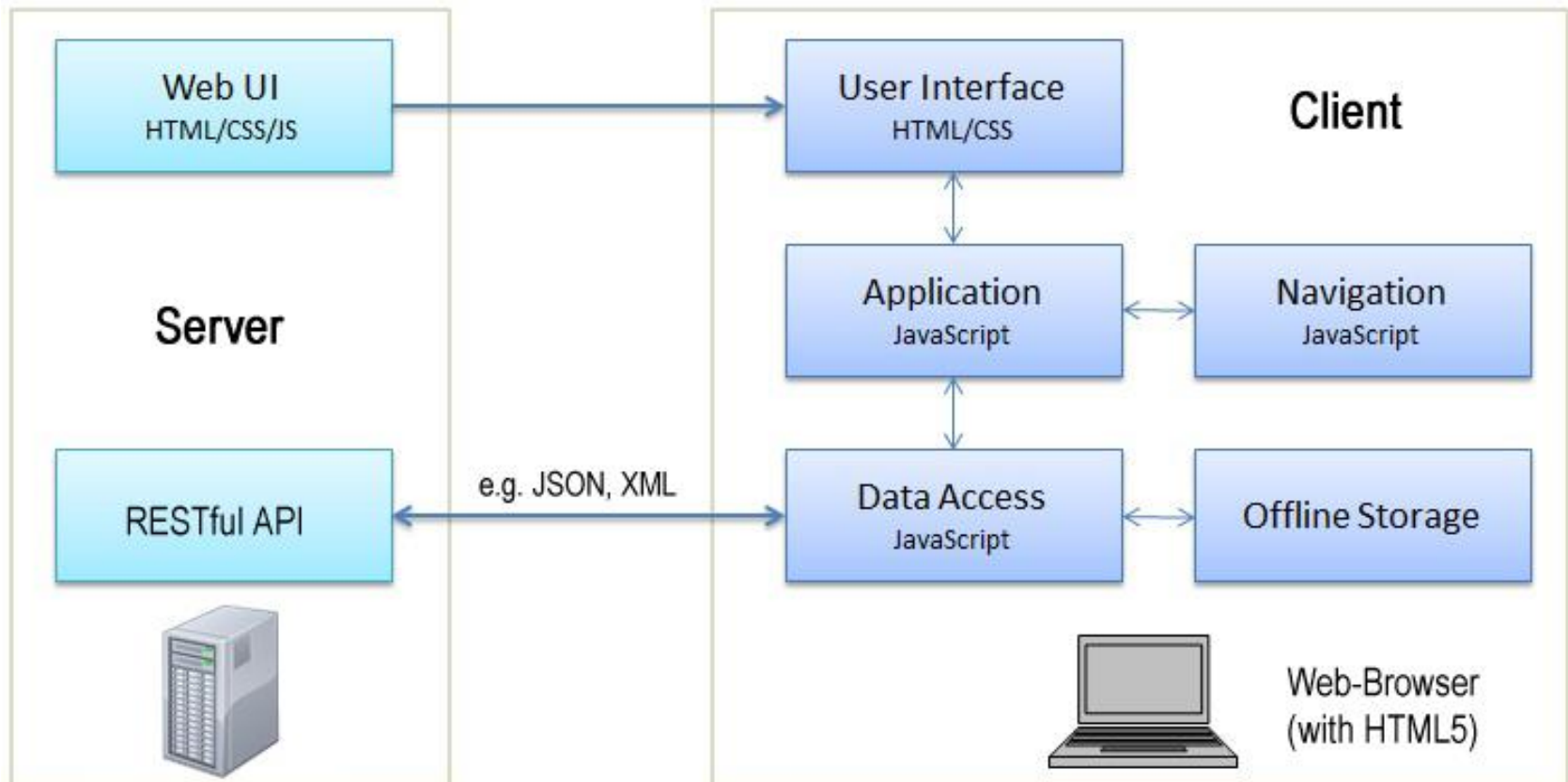
- Schnittstelle dient dazu von externen Systemen genutzt zu werden
- REST sollte in der eigenen Architektur **nicht** eine erweiterte Schnittstelle zur Datenbank darstellen
- Die REST Schnittstelle ist weit oben anzusiedeln, da sie (teils) Geschäftsprozesse anstößt und die Validierung von Geschäftsregeln sowie Daten vornimmt

Web-API Architektur

- RESTful API für den externen Zugriff auf Dienste im Web
- Es müssen die REST “Spielregeln” eingehalten werden
- Dazu kommen weitere Punkte die beachtet werden sollten:
 - ▣ API-Dokumentation
 - ▣ Standardformate verwenden (JSON, XML, ...), am Besten ein bereits bestehendes Format dessen Semantik bekannt ist (z.B. ATOM)
 - ▣ Clientbibliotheken anbieten in untersch. Sprachen
 - ▣ Authentisierung/Authorisierung von Endanwendern
 - Falls Dienst von End-User Applikation aufgerufen wird (im Namen eines im System befindlichen Users)
 - Im Web-Umfeld bietet sich OAuth oder OpenID an
 - ▣ API-Keys verwenden
 - “Shared-Secret” zum authentisierten Zugriff auf die API

Architekturen

Frontend-Architektur (Single Page Applications)



Nach: <http://devmanagement.files.wordpress.com/2012/09/spa.jpg>

Zusammenfassung

- REST ist kein Standard, sondern ein Architekturstil
 - ▣ ..also eine Sammlung von verschiedenen Ansätzen, die ein harmonisierendes Gesamtbild – einen Architekturstil – bilden
- 5 REST Kernprinzipien
 - ▣ Ressourcen, Hypermedia, Standardmethoden, Repräsentationen und statuslose Kommunikation
- SOAP vs. REST (oder auch *Äpfel* vs. *Birnen*)
 - ▣ Vom Protokoll her: REST leichtgewichtig, SOAP standardisiert, verschiedene Gewichtsverteilung im Architektur-Dreieck (Instanzen/Operationen/Datentypen)
- HATEOAS: eines der wichtigsten aber auch in der Praxis (leider) weniger gesehenen Prinzipien im Umgang mit REST
- Spezialthemen: Paginierung, Sortierung, Filterung, Versionierung
- Skalierbarkeit
 - ▣ ...Session-Zustände im Hauptspeicher zu verwalten macht eine Infrastruktur langsam und verhindert dessen Skalierbarkeit
- Caching
 - ▣ ...Caching lässt unsere Architektur skalieren und macht Clients und Server performanter!
- Sicherheit
 - ▣ ...zu 80% fährt man mit Basic Authentication + SSL in der Praxis sehr gut.
- Architekturen: Mashup - RESTful SOA, “normaler” Dienst, Web-API, Frontend SPA

Referenzen

Dieses Kapitel basiert zum Großteil auf:

- [1] REST und HTTP, Stefan Tilkov, dpunkt Verlag, 2009.
- [2] RESTful Service Best Practises – Recommendation for creating Web Services, Todd Fredrich, Pearson eCollege, 2013, (verfügbar unter: RestApiTutorial.com).

Weitere Quellen sind:

- [3] Architectural Styles and the Design of Network-based Software Architectures, Roy Thomas Fielding, Dissertation, 2000 (verfügbar unter: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>)
- [4] RESTful Web Services, Leonard Richardson & Sam Ruby, O'Reilly, 2007.
- [5] REST Anti-Patterns, Stefan Tilkov, InfoQ, 2008, (verfügbar unter: <http://www.infoq.com/articles/rest-anti-patterns>).
- [6] Hypermedia Discussion: http://www.infoq.com/interviews/tilkov-rest-hypermedia?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk