# Web-Developer
## Schnittstellen

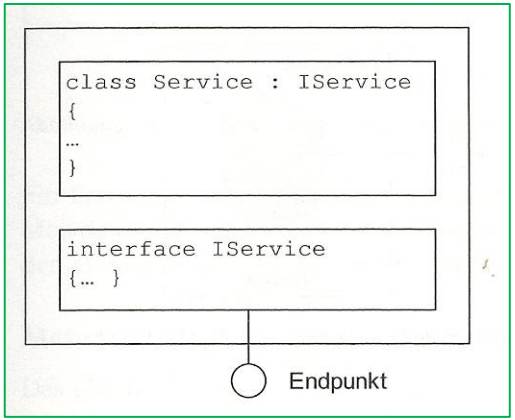
# SCHNITTSTELLEN

# Was ist sind ein Service und Schnittstellen?

- Ein Service ist eine Sammlung von Operationen, die eine funktionale Einheit bilden und von beliebigen Anwendungen konsumiert werden können.
- Ein Service bildete einen in sich abgeschlossenen Prozess innerhalb eines komplexeren Geschäftsprozesses ab.
- Prozesse werden nach den Anforderungen der Fachabteilungen geformt und entsprechen meistens Industriestandards.
- Die Funktionalität wird als Geschäftslogik in einer Klasse implementiert.
- Die Implementierungsdetails werden gekapselt.
- Die Funktionalität wird mit einer passenden Schnittstelle als **Contract** veröffentlicht.
- Die eigentliche Implementierung bleibt dem Anwender verborgen (Kapselung)

Vgl. (Kotz/Hölzl, 2009, S. 13.

# Was ist sind ein Service und Schnittstellen?




Quelle: (Kotz/Hölzl, 2009, S.13.)

- Schnittstellen sind per Definition invariant und dürfen nach der Veröffentlichung nicht mehr verändert werden.
- Schnittstellen, weitere Informationen (Beschreibung der Funktionalität, erwartete Parameter, Sicherheitsanforderungen) werden mithilfe standardisierter Metadaten als Endpunkte bekannt gegeben.
- Die Funktionalität eines Services wird durch einen oder mehreren austauschbaren interoperablen Endpunkten verfügbar gemacht.

Vgl. (Kotz/Hölzl, 2009, S. 13f.)

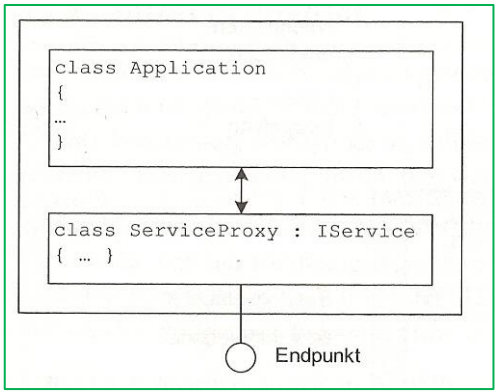
# Was ist sind ein Service und Schnittstellen?

Ein Endpunkt besteht aus den drei elementaren Bereichen:

- Wohin soll eine Nachricht gesendet werden (Addresse, URL)?
- Wie soll eine Nachricht gesendet werden (Spezifikation der Bindung)?
- Wie soll die Nachricht aussehen (JSON, XML, etc.)?

Vgl. (Kotz/Hölzl, 2009, S. 13f.)

# Was ist sind ein Service und Schnittstellen?




Quelle: (Kotz/Hölzl, 2009, S.14.)

- Service sind meist interoperabel und es werden nur standardisierte Daten ausgetauscht.
- Sie sind nicht mit einer spezifischen Technologie verknüpft.

**Lose Kopplung von Clients!**

- Für die Nutzung des Services wird eine Proxy-Klasse erstellt, die die gleichen in der Schnittstelle definierten Funktionen wie die Klasse des Services enthält.
- Die Anwendung erzeugt eine Instanz der Proxy Klasse. Für die Anwendung ist der Service nun wie eine „lokale“ Klasse.

Vgl. (Kotz/Hölzl, 2009, S. 13f.)

# Was ist sind ein Service und Schnittstellen?

Zustandslosigkeit erreichen

- Service und Client dürfen keinen Informationen zwischen zwei Nachrichten aufheben.
- Jeder Service agiert unabhängig vom anderen.
- Es dürfen keine externen Abhängigkeiten bestehen.
- Zwei parallel zugreifende Clients müssen völlig unabhängig voneinander arbeiten.
- Service muss fehlertolerant sein.
  Verwendeten Ressourcen werden nach einen Aufruf wieder in einen konstanten Zustand versetzen - unabhängig davon ob der Aufruf erfolgreich war oder nicht.

Vgl. (Kotz/Hölzl, 2009, S. 17.)

# REST - Representational State Transfer

## Warum REST?

Lose Kopplung

- Unabhängige Teilsysteme anstelle monolithische Riesensysteme
- Kommunikation von isolierten Systemen über Schnittstellen
- Uniforme Schnittstelle und Hypermedia = Verknüpfung mithilfe von Links

Vgl. (Tilkov et al. 2015, S. 2f.)

# REST - Representational State Transfer

## Warum REST?

Interoperabilität

- Kommunikation von Systemen mit technisch stark unterschiedlichen Implementierungen durch festgelegte gemeinsame Standards.
- HTTP – Keine Software ist in Bezug auf die Größe des Geltungsbereiches so umfassend wie HTTP.
- HTTP als Protokoll und URIs als Identifikationsmechanismus werden praktisch in jeder Umgebung unterstützt – vom Großrechner bis hin zum Embedded-System

Vgl. (Tilkov et al. 2015, S. 3.)

# REST - Representational State Transfer

## Warum REST?

### Wiederverwendung

- Höhere Wiederverwendungsraten senken die Kosten in der Softwareerstellung.
- REST bietet die Möglichkeit von sich überlappenden Schnittstellen und die Unterstützung der ungeplanten Wiederverwendung (serendipitous reuse).
- Der Feind der Wiederverwendung ist die mangelhafte Vorhersehbarkeit der Anforderungen.
- REST hat nur **eine** Schnittstelle – jeder Client, der diese Schnittstelle verwenden kann, kann mit dem REST-basierten Service kommunizieren.

Vgl. (Tilkov et al. 2015, S. 3.)

# REST - Representational State Transfer

## Warum REST?

Performance und Skalierbarkeit

- Dienste sollen so schnell wie möglich antworten.
- Eine Größtmögliche Anzahl an Anfragen sollen in einem definierten Zeitraum beantwortet werden können.
- Performance hängt ab von internen Architektur und der Verteilungsarchitektur (= Kommunikation der verteilten Komponenten über das Netz miteinander)
- REST hat einen großen Einfluss auf die Verteilungsarchitektur.
  Eine Vielzahl an Anfragen können aus dem Cache gelesen werden!
- Beste Skalierbarkeit durch den Verzicht auf einen sitzungsbezogenen Status!

Vgl. (Tilkov et al. 2015, S. 4.)

# REST - Representational State Transfer

## Grundprinzipien

### Eindeutige Identifikation

- Web nutzt ein einheitliches Konzept um Instanzen eindeutig zu identifizieren, die URI!
- Durch die URI wird ein globaler Namensraum gebildet, die Elemente sind dadruch weltweit eindeutig.
  (e.g. Verschicken von Links auf einen Kunden, auf ein Produkt)
- Menschenlesbare URI sind leicht zu interpretieren (Anm. für die REST jedoch irrelevant).

Vgl. (Tilkov et al. 2015, S. 11f.)

# REST - Representational State Transfer

## Grundprinzipien

Hypermedia

```
{
  "amount": 23,
  "customer":{
      "href": "http/example.com/customers/12345"
  },
  "product": {
      "href": "http/example.com/products/4554"
  },
  "cancel": {
      "href": "./cancellations"
  }

 }
```

- = Konzept von Verknüpfungen (Links)
- Verknüpfungen funktionieren anwendungs-übergreifend.
- Da der URI global ist, können alle Ressourcen, aus denen das Web besteht miteinander verknüpft werden.
- Steuerung des Applikationsflusses indem der Server dem Client über Hypermedia-Elemente (Links) mitteilt, welche Aktionen er als nächstes ausführen kann.

Vgl. (Tilkov et al. 2015, S. 13.)

# REST - Representational State Transfer

## Grundprinzipien

Standardmethoden

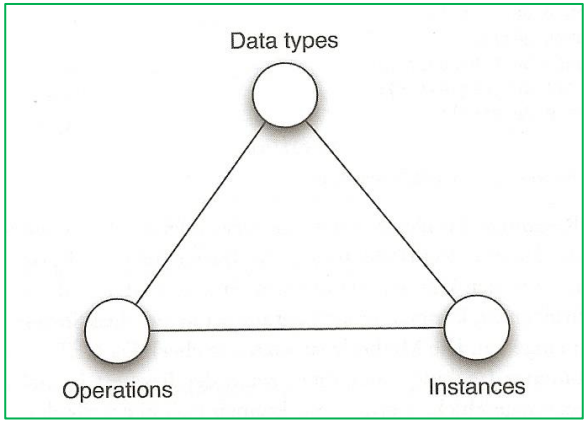
- Jede Ressource unterstützt die gleiche Schnittstelle, den gleichen Satz von Methoden.
  (Wird die Schnittstellen-Methode nicht implementiert, erscheint zumindest eine Fehlermeldung)
- Methoden: GET / POST (bekanntesten) und PUT, DELETE , HEAD und OPTIONS
- Mit **GET** kann eine Repräsentation / Darstellung einer Ressource abgeholt werden, da für alle Ressourcen das gleiche Interface verwendet wird.
  GET ist **„safe“** – denn sie gehen bei einem GET-Aufruf keine Verpflichtung ein!
- GET unterstützt ein effizientes und raffiniertes Caching (Request gelangen nicht zum Server).
- GET ist idempotent, d.h., erhält man auf ein GET keine Antwort, kann die Anfrage nochmals verschickt werden – ob die erste Anfrage nun angekommen ist, oder nicht.

Vgl. (Tilkov et al. 2015, S. 15.)

# REST - Representational State Transfer

## Grundprinzipien

Wie soll man allein mit GET, PUT, POST und DELETE auskommen?




Quelle: (Tilkov et al, 2015, S.16.)

- Punkte: Anzahl unterstützter Operationen, die verwendeten Datentypen und die Anzahl der Instanzen (Services, Fassaden, Ressourcen)
- In gewohnter Weise wird an den Operations und Data type geschraubt.
- Bei REST bleibt die Anzahl an Operationen konstant.

Vgl. (Tilkov et al. 2015, S. 16.)

# REST - Representational State Transfer

## Grundprinzipien

Wie soll man allein mit GET, PUT, POST und DELETE auskommen?

Beispiele:
Durch REST werden der Anwendung unzählige Kunde-URIs hinzugefügt (für jeden Kunden eine).

Im COBRA Stil (wie er in SOAP zum Einsatz kommt) ist der URI der ausschließlich der „Endpunkt“ – quasi die Pforte in ein Universum von Ressourcen. Der Zugang ist nur jenen gestattet, die den Schlüssel dazu haben.

Eine Web-Anwendung, in der sich der URI in der Adresszeile nie ändert. Die Anwendung als Ganzes wurde mit einer URI ausgestattet.

Vgl. (Tilkov et al. 2015, S. 16.)

# REST - Representational State Transfer

## Ressourcen und Repräsentation

Wie weiß ein Client, was er mit den Daten machen soll, die von einem GET- oder POST-Request bekommt?

- HTTP trennt die Verantwortlichkeit für Daten und Operationen.
  D.h. ein Client der ein bestimmten Datenformat verarbeiten kann, ist in der Lage mit einer Ressource zu interagieren, dien eine Repräsentation in diesem Format zur Verfügung stellt.
- Die Operationen sind überall dieselben.

Vgl. (Tilkov et al. 2015, S. 16.)

# REST - Representational State Transfer

## Ressourcen und Repräsentation

```
GET /customers/12345 HTTP/1.1
Host example.com
Accept: application/xml


GET /customers/12345 HTTP/1.1
Host example.com
Accept: text/x-vcard
```

Quelle: In Anlehnung an Tilkov (2015), S.17.

- Auf den ersten GET-Request werden Daten in XML angefordert.
- Im zweiten Fall könnte die Kundenadresse in einem vCard-Format zurückgeliefert werden.
- Das Format wird auf Basis von HTTP Content Negotiation angefordert (Attribut: Accept)
- Pro Ressource können damit mehr als eine Repräsentation zurückgeliefert werden – auch eine HTML Seite!

Vgl. (Tilkov et al. 2015, S. 17f.)

# REST - Representational State Transfer

## Statuslose Kommunikation

- REST sieht nicht vor, dass ein clientspezifischer Status vorübergehend (transient) serverseitig über die Dauer eines Requests hinweg abgelegt wird (z.B. Sessions).
- Der Server interessiert sich für den Client nur, wenn er gerade einen seiner Requests verarbeitet; danach kann er seine Existenz wieder vergessen.
- Der Status muss **beim Client** gehalten werden oder beim Server in einen **Ressourcenstatus umgewandelt** werden (d.h., Ablage in einen Datenspeicher mit einer eindeutigen ID).

Beispiel anhand des Klassikers Warenkorb:

Der Warenkorb wird zu einer eigenen Ressource, die über einen URI (z.B shopping/warenkorb/12345) zur jeder Zeit aufgerufen werden kann.

Vgl. (Tilkov et al. 2015, S. 18.)

# REST - Representational State Transfer

## Statuslose Kommunikation

- Die Kopplung des Clients an den Server wird durch den Verzicht auf einen Sitzungsstatus verringert. Mehrere aufeinanderfolgende Abfragen müssen nicht von der gleichen Serverinstanz bearbeitet werden.
- Die **Skalierbarkeit** wird dadurch vereinfacht!

Vgl. (Tilkov et al. 2015, S. 18.)

# REST - Representational State Transfer

## Accidentally REST (= REST-konform aus Versehen)

- Von Accidentally REST spricht man, wenn man dem Schnittstellenentwurf relativ klar ansehen kann, das der Designer REST nicht verstanden hat, aber sich zufällig dennoch an die Vorgaben hält [Backer2005].
- Indikator: In der URI ist ein Methodennamen enthalten, die Methode aber „safe“ im HTTP-Sinne ist („safe“ sind GET und HEAD).

Beispiel:

| | |
|---|---|
| http://example.com/customterservice?operation=findCustomer&id=12345 | REST konform |
| http://example.com/customers/1234 | REST konform |
| http://example.com/customerservice?operation=deleteCustomer&id=12345 | DESIGN SMELL **NICHT REST konform** |

Vgl. (Tilkov et al. 2015, S. 18.)

# REST - Representational State Transfer

## Verben - GET

- Entsprechend der Spezifikation dient GET dazu, die Informationen, die durch die URI identifiziert werden, in Form einer Entity (Repräsentation) abzuholen.
- GET ist als ‚**safe**' (sicher) definiert und ist **idempotent**.
- GET kann ‚Seiteneffekte' erzeugen (z.B. Eintrag in Log-Datei) – damit dürfen jedoch keine Zustandsänderungen angefordert werden (z.B. Insert, Delete, Update)
- **Safe (sicher)** bedeutet, dass der Benutzer mit dem Aufruf von GET und dem damit verbundenen Lesen von Daten **keine** Verpflichtung eingeht.
- GET macht ca. 95% der Interaktionen im Web aus!
- GET sollte immer ein sinnvolles Ergebnis zurückliefern – zumindest eine Fehlermeldung.

Vgl. (Tilkov et al. 2015, S. 54f.)

# REST - Representational State Transfer

## Verben - HEAD

- HEAD ist dem GET sehr ähnlich. Es wird aber keine Repräsentation zurückgeliefert, sondern nur die Metadaten
- Der Client kann sich über die Metadaten informieren, ohne die eigentlichen Daten transferieren zu müssen (Gibt es die Ressource überhaupt? Wann ist der Zeitpunkt der letzten Änderung?)
- Es müssen die gleichen Metadaten zurückgeliefert werden, wie bei einem GET auf die gleiche URI.
- HEAD ist wie GET ‚**safe**' (sicher) und **idempotent**.

Vgl. (Tilkov et al. 2015, S. 55f.)

# REST - Representational State Transfer

## Verben - PUT

- Mit PUT wird eine bestehende Ressource aktualisiert oder, falls sie noch nicht besteht, erzeugt.
- PUT wirkt sich immer nur auf die Ressource aus, die mit dem URI adressiert wird.
- Bei PUT wird immer der gesamten Inhalt übertragen.
- PUT ist somit das Gegenteil von GET.
- Das Format wird mit dem Content-Type-Header (contentType) definiert, die Daten / Zustand der Ressource wird im Entity Body der HTTP Nachricht übermittelt. Die Daten müssen nicht vollständig sein.
- PUT ist IDEMPOTENT: mehrmaliges Aufrufen verursacht die gleichen Seiteneffekte ein einmaliges.
- Methoden müssen korrekt umgesetzt werden, um nicht gegen die Erwartungen des Clients zu verstoßen.

Vgl. (Tilkov et al. 2015, S. 56)

# REST - Representational State Transfer

## Verben - POST

- POST bedeutet, dass eine neue Ressource unter einer bestimmten URI angelegt wird.
- Im weiteren Sinne wird POST für alle Zwecke eingesetzt, in denen keine anderen Methoden möglich sind, d.h., immer dann, wen eine beliebige Verarbeitung angestoßen werden soll.
- POST gibt nicht die URI der Ressource an (ID der Ressource), die neu angelegt werden soll, sondern nur den URI für das Anlegen selbst.
- HTTP-Status-Code ist: 201 - Created. Die URI wird vom Server bestimmt und über den Location-Header dem Client bekannt gegeben.
- POST ist **nicht** idempotent und Antworten können nicht gecacht werden!

Vgl. (Tilkov et al. 2015, S. 57.)

# REST - Representational State Transfer

## Verben - DELETE

- DELETE ist für das Löschen einer Ressource zuständig, deren URI im Request angegeben wird.
- DELETE ist idempotent, da mehrmaliges Löschen den gleichen Effekt erzeugt wie ein einmaliges löschen.
- DELETE ist als logische Löschen zu verstehen.
  Es kann auch dafür verwendet werden, um in der Persistenzschicht den Status zu ändern
  (e.g. Status: gelöscht / storniert).

Vgl. (Tilkov et al. 2015, S. 57f.)

# REST - Representational State Transfer

## Verben - OPTIONS

- OPTIONS liefert Metadaten über eine Ressource (Allow-Header) über die Methoden, die eine Ressource unterstützt.
- OPTIONS ist idempotent und sicher.
- Das Resultat darf nicht gecacht werden, außer es werden explizite Cache-Header gesetzt.

## TRACE, CONNECT

- TRACE dient zur Diagnose von HTTP-Verbindungen.
- CONNECT dient zur Initiierung einer Ende-zu-Ende Verbindung von SSL durch einen Proxy.

Vgl. (Tilkov et al. 2015, S. 57f.)

# REST - Representational State Transfer

## Security – SSL und HTTPS

- Verschlüsselung der Kommunikation zwischen Client und Server mittels HTTPS (über SSL bzw. TLS (Nachfolger)).
- Der Server muss über ein geeignetes Zertifikat verfügen, um seine Identität gegenüber dem Client zu beweisen.
- Wurde die Verbindung etabliert, macht SSL das Abhören der Kommunikation durch Intermediäre in der Theorie unmöglich.

  Hinweis:
  Es gibt dennoch Fehler in den kryptografischen Algorithmen (Heartbleed-Bug) bzw. Man-in-the-Middle Attacken bei der sich ein anderer Server als glaubwürdiger Server ausgibt, als jenen den man gewünscht hat.

Vgl. (Tilkov et al. 2015, S. 139f.)

# REST - Representational State Transfer

## HTTP - Authentifizierung

- HTTP Authentifizierung ist seid Version 1.0 in HTTP enthalten.
- Der Server antwortet mit dem Fehlercode **401 Unauthorized** auf Zugriffe, die für anonyme Zugriffe nicht zugelassen sind.
- Die Antwort enthält eine menschenlesbare Darstellung:
  WWW-Authenticate: Basic, Schema der Authentifizierung, hier Basic
  realm=„Private Area“, benennt den Geltungsbereich der verwendeten Authentisierung (realm = Königreich).
- **Authentication Challenge**
  Der Client kann nur auf die Ressourcen zugreifen, wenn die Authentisierungsinformationen mitgesandt werden und diese dem Schema (Basic) entsprechen.

Vgl. (Tilkov et al. 2015, S. 140ff.)

# REST - Representational State Transfer

## HTTP – Basic Authentifizierung

- Benutzername und Passwort werden durch Doppelpunkt getrennt verkettet und nach dem Base64-Verfahren codiert.
  Javascript: btoa(‘admin‘ + ':' + ‘secret‘) ->  RtaW46c2VjcmVOCg=

- Auf eine Anfrage Antwortet der Server mit 401 und den Angaben für das Authentifizierungsschema und dem Realm. Der Client verschickt den Base64-Code in einem zweiten Versuch in einem „Authorization“-Header.

```
GET / HTTP 1.00
Accept: text/html
Authorization: Basic RtaW46c2VjcmVOCg=
```

- Der Server kann die Base64-Codierung rückgängig machen und prüfen, ob diese Informationen gültig sind.

Vgl. (Tilkov et al. 2015, S. 142.)

# REST - Representational State Transfer

## HTTP – Basic Authentifizierung

**Schwächen:**

- Das Passwort wird quasi in Klartext übermittelt. Base64 ist keine Verschlüsselung, sondern eine Codierung um einen Basiszeichensatz zu unterstützen.
- Identität des Servers ist nicht sichergestellt, d.h., der Client schickt seinen Benutzernamen und Passwort an einen Angreifer.
- Man-in-the-Middle Attacken sind möglich – d.h., die Nachricht kann auf dem Weg vom Client zum Server manipuliert werden.
- Replay-Angriff: Erkennt der Angreifen nur eine einzige legitime Nachricht, kann er sie ganz oder teilweise noch einmal versenden.

Vgl. (Tilkov et al. 2015, S. 143.)

# REST - Representational State Transfer

## HTTPS + Basic Auth(entifizierung)

- HTTP Basic Authentifizierung ist für einen ernsthaften Einsatz praktisch ungeeignet.
- Die Schwächen können jedoch durch HTTPS ausgeglichen werden.
- HTTPS + Basic Auth ist die **80% Lösung**
- Ermöglicht mit begrenzten Mitteln, einem sehr geringe Risiko, in kurzer Zeit eine Lösung, die den Anforderungen gerecht wird.
- Kein Schutz vor CSRF (Cross-Site-Request-Forgery)
  In diesem Szenario wird ein ungewollter schreibender Zugriff auf die Webanwendung durchgeführt, bei der der Nutzer gerade eingeloggt ist (z.B. durch Link in einer E-Mail, präparierte Webseite).
  ASP.NET MVC schützt hier mit dem Antiforgery Token.

Vgl. (Tilkov et al. 2015, S. 143.)

# REST - Representational State Transfer

## OpenID

- Wird dort eingesetzt wo auf unterschiedliche Sites zugegriffen wird und die Anforderung einer einmaligen Authentifizierung besteht.
- Die Dienste kooperieren über ein offenes, standardisiertes Protokoll mit einem Authentifizierungsservice (OpenID Provider), bei dem man sich einmal zentral registrieren muss.
- Es interagieren der Browser des Endanwenders, eine Webanwendung und ein OpenID-Provider, der für die Authentifizierung zuständig ist miteinander:
  1. Benutzer gibt bei der Registrierung einen URI zur Anmeldung an einem OpenID-Provider an.
  2. Die Web-Anwendung prüft den URI und schickt eine Redirect-Antwort an den Client zurück – Ziel ist der Provider
  3. Der Redirect zeigt auf den Provider und enthält als Return-URI den URI der Webanwendung.
  4. Nach der Authentifizierung sendet der Provider anhand des Return-URI den Redirect zur Webanwendung. Darin sind die Informationen über den Erfolg des Authentifizierungsversuches enthalten.
  5. Die Webanwendung kann jetzt feststellen, ob die Authentifizierung erfolgreich war.

Vgl. (Tilkov et al. 2015, S. 150f.)

# REST - Representational State Transfer

## OAuth

- OAuth ist aus der gleichen Motivation wie OpenID entstanden; und zwar für die Übertragung von Rechten an eine andere Anwendung.
- Bei OAuth ist zwischen OAuth1.0 und OAuth 2.0 zu unterscheiden, die jedoch nicht kompatibel sind.
- Bei OAuth stellt ein **Service-Provider** geschützte Ressourcen zur Verfügung.
  OAuth ermöglicht dem **Consumer**, der diese Ressourcen nutzen will, einen eingeschränkten Zugriff auf diese eingeschränkten Ressourcen.
  Die Entscheidung, ob der Consumer auf eine Ressource zugreifen darf, erfolgt durch den **Endanwender**.

Vgl. (Tilkov et al. 2015, S. 150f.)

# REST - Representational State Transfer

## Oauth 1.0

- Ablauf
  1. Consumer (der die Ressource nutzen will) fordert vom Serviceprovider einen Request-Token an
  2. Consumer sendet an den Browser des Endanwenders einen Redirect auf den URI des Serviceproviders, der einen temporären Request-Token codiert.
  3. Der Serviceprovider lässt sich vom Endanwender bestätigen, dass dieser den Zugriff auf die geschützte Ressource durch den Consumer erlauben möchten und liefert als Resultat einen Verifizierungscode.
  4. Es erfolgt ein erneuter Redirect auf den Consumer; dabei wird der Verifizierungscode übermittelt.
  5. Der Consumer wandelt den temporären Request-Token anhand des Verifizierungscodes in ein Authentifizierungs-Token um.
  6. Der Consumer kann auf die geschützte Ressource des Serviceproviders zugreifen, wenn er das Authentifizierungs-Token mitsendet.
  7. Der Authentifizierungs-Token ist nur dem Consumer, nicht aber dem Endanwender bekannt!

Vgl. (Tilkov et al. 2015, S. 150f.)

# REST - Representational State Transfer

## OAuth 2.0

- Der Ablauf von OAuth 2.0 ist jenem von OAuth 1.0 im Falle einer Webanwendungen sehr ähnlich.
- OAuth 2.0 ist kein Protokoll sondern ein Framework.
- Bei der Verschlüsselung verlässt sich OAuth in großen Teilen auf TLS. OAuth 2.0 verlangt deshalb nach einer HTTPS (SSL/TLS) Verschlüsselung des Transports, ähnlich wie bei der HTTP Basic Authentication.
- Neben Core Spezifikationen wurde das sehr simple Bearer-Token-Verfahren eingeführt.
- Bearer-Token:
  Mit jedem Request wird das Token mitgeschickt (vorzugsweise im Authorization Header).

Vgl. (Tilkov et al. 2015, S. 150f.)

# REST - Representational State Transfer

## OAuth – Empfehlungen zur Absicherung der Token

- Schutz des Token vor dem Zugriff durch Dritte.
- TLS (HTTPS) richtig verwenden – Prüfung der gesamten Zertifikatskette
- Sind die Token in Cookies enthalten, dann sollten die Informationen durch das Attribut „HttpOnly“ und „Secure“ abgesichert werden. Die Token sollten zudem verschlüsselt werden. Absicherung gegen CSRF (Cross-Site-Request-Forgery).
- Begrenzte Lebensdauer der Token.
- Bearer-Token haben nichts in einer URL zu suchen, sondern gehören in den Request-Header oder Body, da diese im Browser und auch im Server in der Regel als vertraulich behandelt werden. D.h., der Token sollte nicht aus dem Verlauf / History ausgelesen werden können!

Vgl. (Tilkov et al. 2015, S. 150f.)

## REST - Representational State Transfer

FINE

Vgl. (Tilkov et al. 2015, S. 150f.)