DOAG | Development & Tools | 19.11.2014

# Eine Architektur für mobile Anwendungen bei der Bundesagentur für Arbeit


IT-Systemhaus

# Bundesagentur für Arbeit (BA)

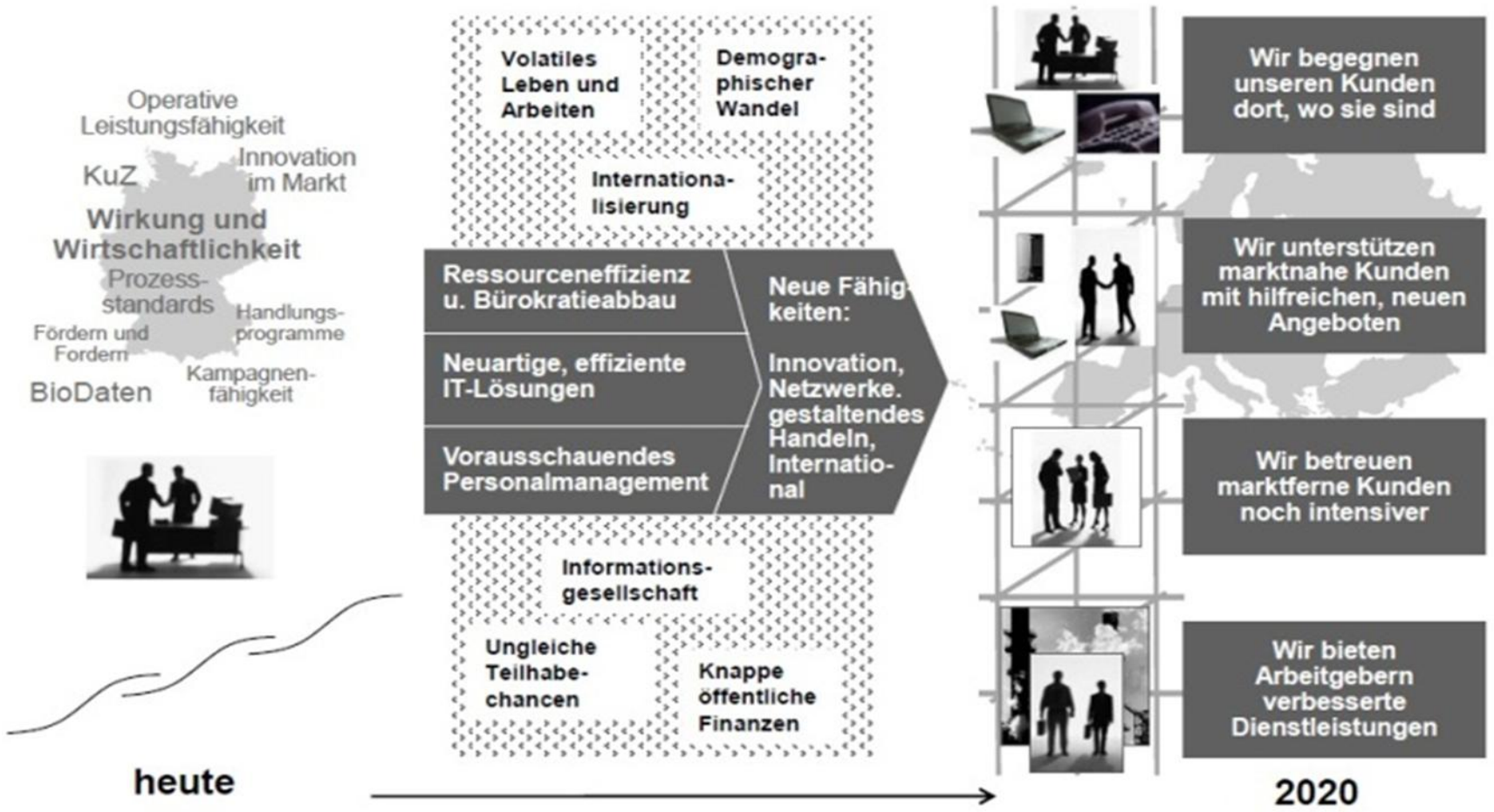
- **zentraler Dienstleister am Arbeitsmarkt**
- **Körperschaft des öffentlichen Rechts mit Selbstverwaltung**
  - Zentrale mit
  - 10 Regionaldirektionen
  - 156 Agenturen für Arbeit
  - 600 Geschäftsstellen und
  - 7 besondere Dienststellen in
  - 1.650 Liegenschaften
- **knapp 100.000 BA-Mitarbeiter/-innen**

# Kurzprofil – BA-Informationstechnik

- Hauptsitz: Nürnberg
- IT-Mitarbeiter/-innen: 2.100
- Vernetzte PC: 160.000
- BA-IT-Verfahren: 120

- monatlich:
- E-Mail-Volumen: 35 Mio.
- Überweisungen: 17 Mio. mit 8 Mrd. Euro
- Postsendungen: 12 Mio.
- Druckseiten: 50 Mio.

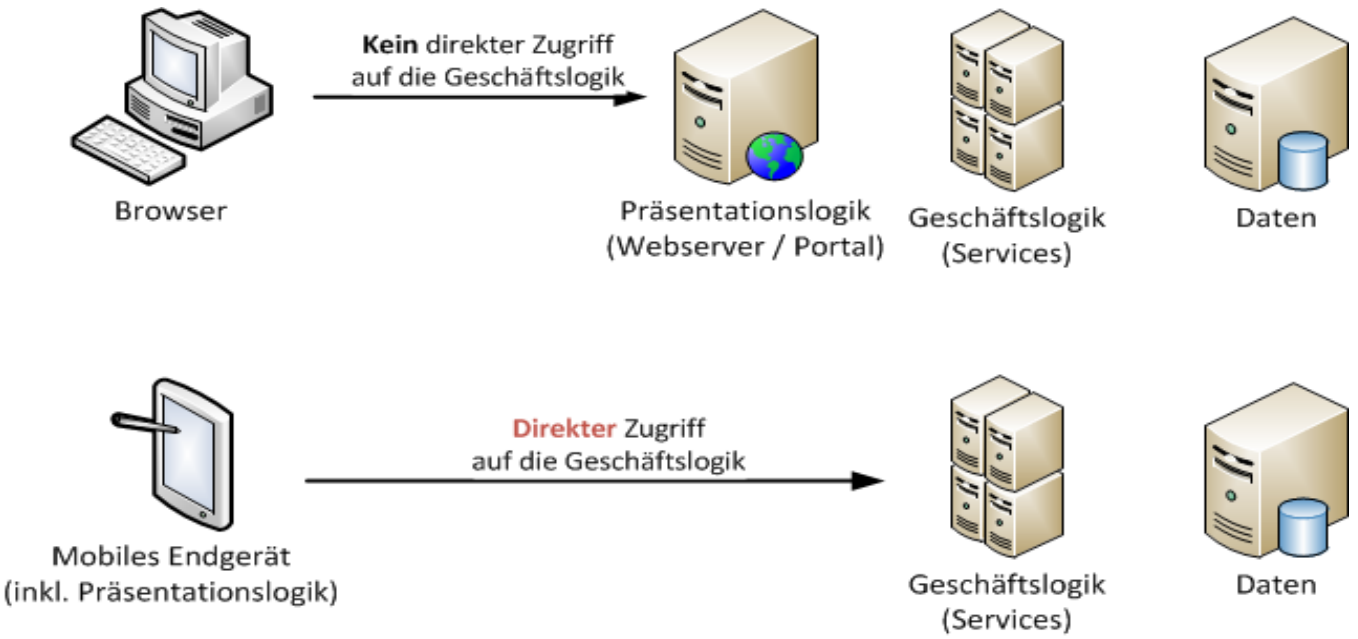
# BA 2020 – vom serviceorientierten Dienstleister zum kundenorientierten Lösungsanbieter



# Zielgruppenspezifische Apps

- **Zielgruppen**
  - Bürgerinnen/Bürger
  - Unternehmen
  - Institutionen
- **App sind auf Zielgruppe zugeschnitten**
  - Design
  - Ansprache
  - Funktionalität
- **Herausforderung**
  - Backend-Services (Anwendungen) eher orthogonal zu App Funktionalität
    → Orchestrierung von Backend-Services
  - 3 Releases pro Jahr
    → für App unrealistisch

# Architektur einer 3 Schicht Web-Applikation




- **Services würden exponiert da Frontend auf mobilem Gerät**
  - viele Angriffsvektoren
  - direkte Abhängigkeit

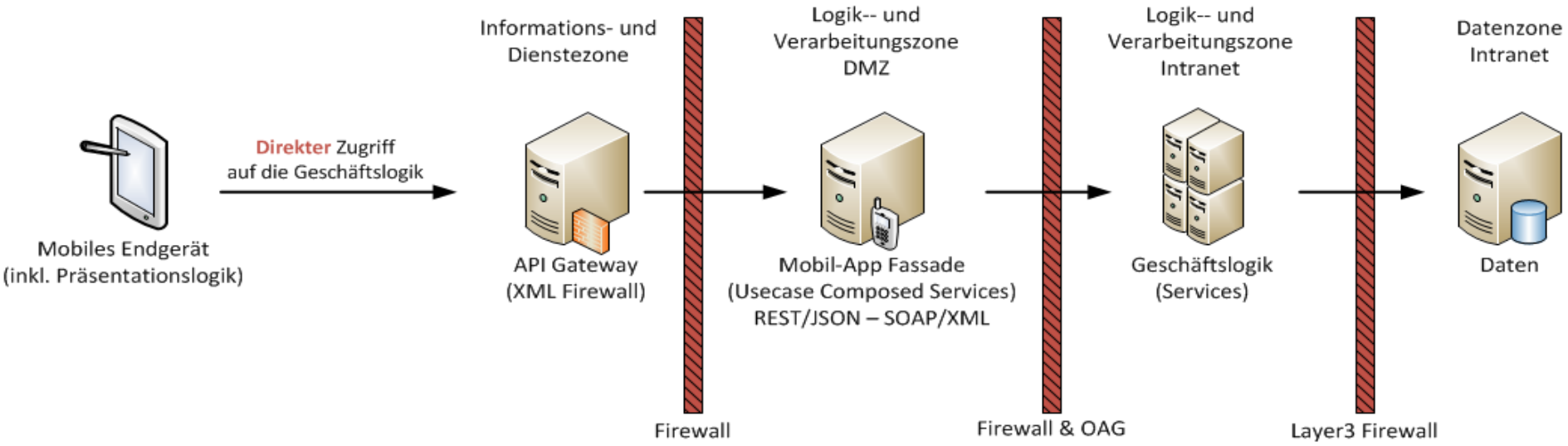
# Risikobetrachtung öffentlicher Apps

- **Schutz der Backendsysteme vor Überlast**
  - Systeme müssen auch für die „tägliche Arbeit“ zur Verfügung stehen
  - Die exponierten Dienste müssen gegen unberechtigten und übermäßigen Gebrauch geschützt werden
  - Gewisse APIs sollen nur von genehmigten Nutzern benutzt werden (Nutzervereinbarung)
- **Datenschutz**
  - Persönliche Informationen sollen nur soweit wie nötig exponiert werden
  - Zugriff nur nach Authentisierung des Nutzers
  - Keine Speicherung auf dem Endgerät
- **Vertraulichkeit**
  - Daten werden über öffentliche Netze transportiert
  - Einfaches Abhören möglich → Transportverschlüsselung

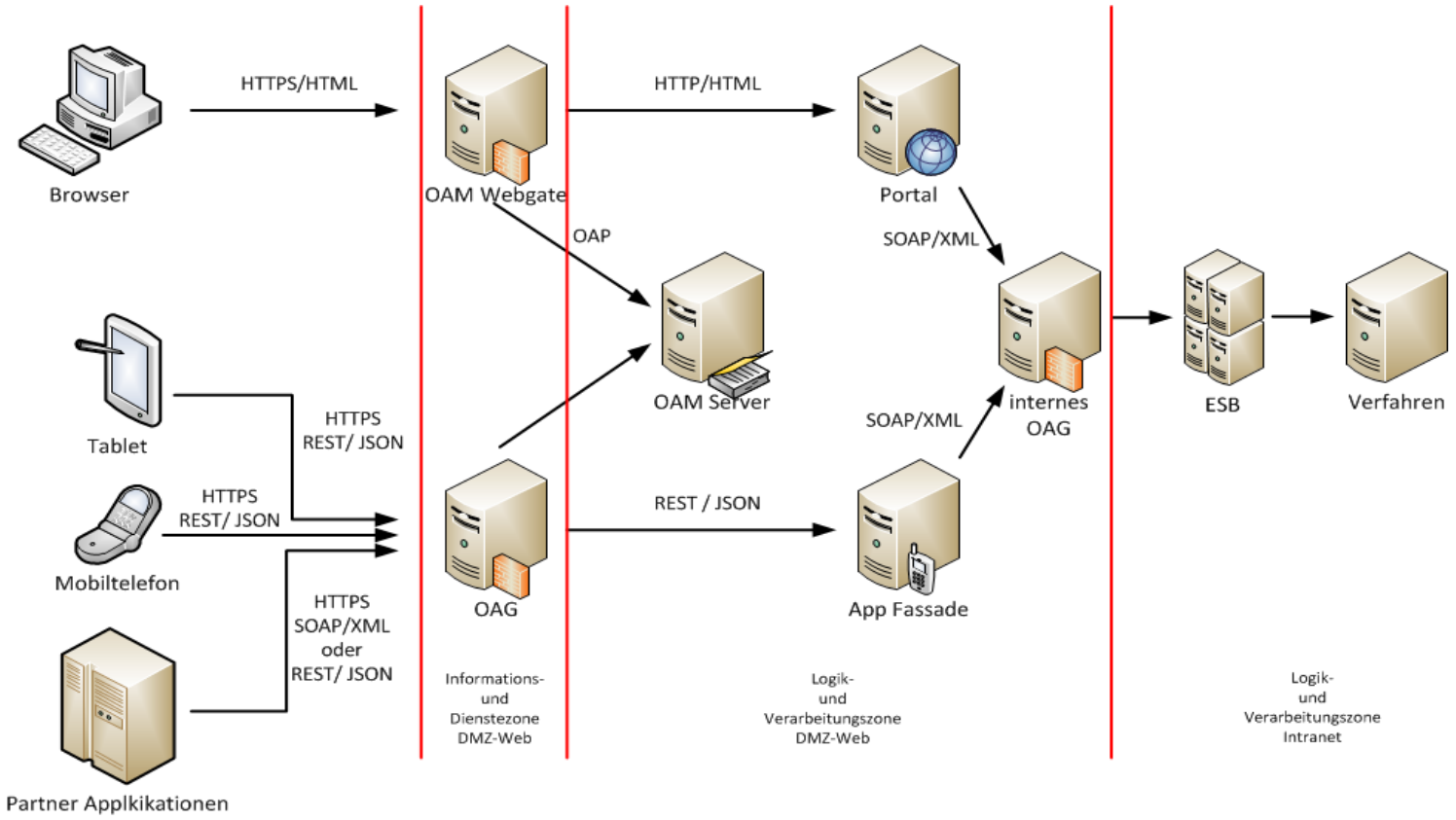
# Anforderungen an eine Architektur für mobile Anwendungen

- **Zugriffe auf persönliche Informationen sollen über die in der DMZ-Web vorhandenen Sicherheitsarchitektur authentisiert werden (OAM)**
- **Absicherung der API für mobile Anwendung**
  - Throttling
  - Deaktivierung von Client-Typen (API Konsumenten)
- **Trennung von Backend-Services und mobiler Anwendung**
  - Backend-Services dürfen nicht exponiert werden
  - Hohe Flexibilität der API
  - Entkopplung der Release-Zyklen

# Zielarchitektur für mobile Anwendung



# Zielarchitektur für mobile Anwendung (Gesamtbild)



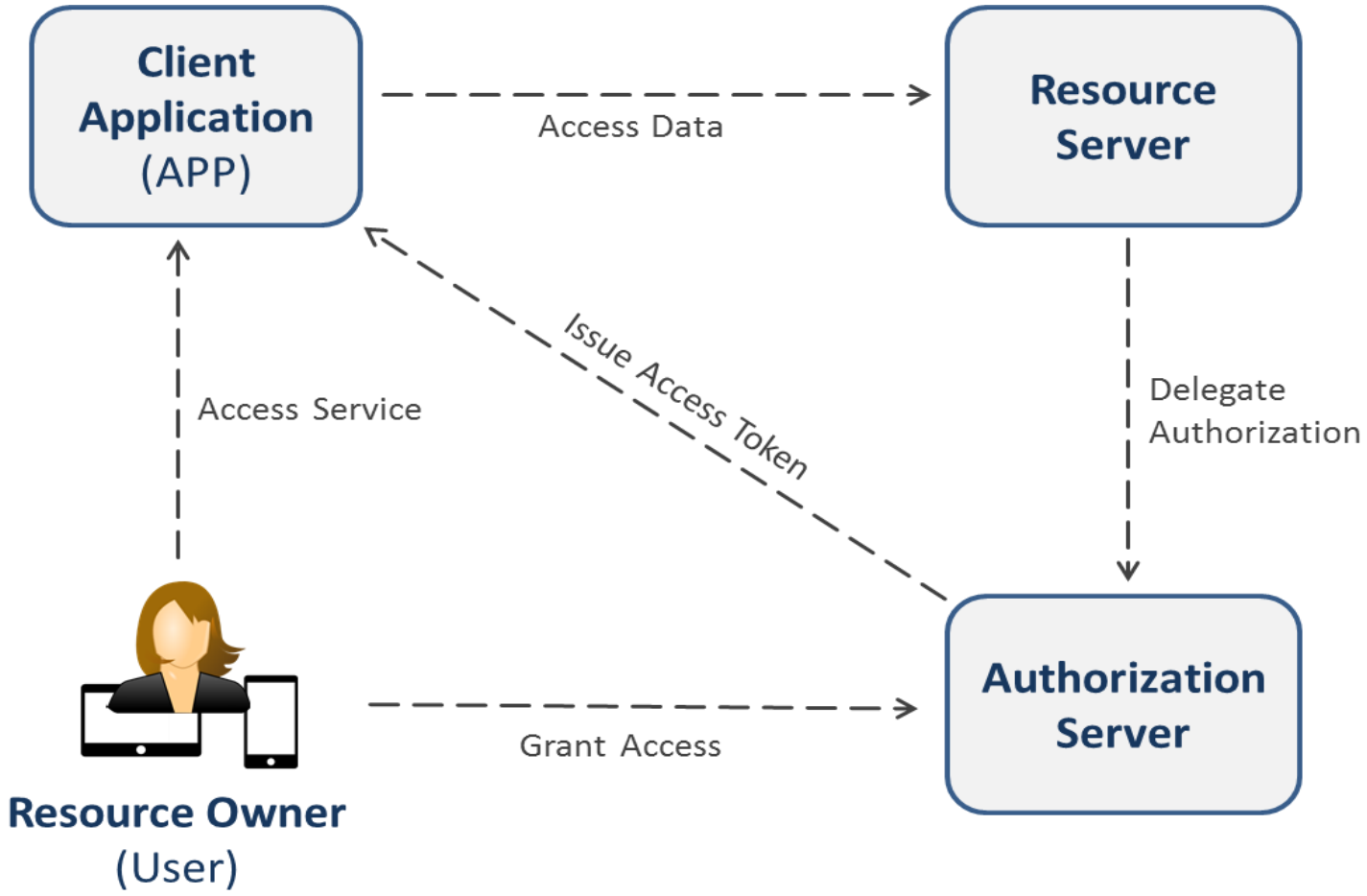
# App Fassade

- **Protokollwandlung REST/SOAP**
- **1:1 Zuordnung zu App**
  - Life-Cycle der API an App gebunden
  - Änderungen nur bei Anforderungen an die APP
- **Integrationspunkt**
  - Service Orchestrierung
- **Lose Kopplung von API und  Backend-Services**
  - Life-Cycle
  - Funktionale Änderungen der App
- **Limitiert/Kapselt die Schnittstelle der Backend-Services**
  - funktional auf eine App zugeschnitten
  - minimales Set an Operationen
- **Realisierung**
  - Plain Java (Web Applikation/Jersey → Weblogic Server)
  - SOA Suite 12c

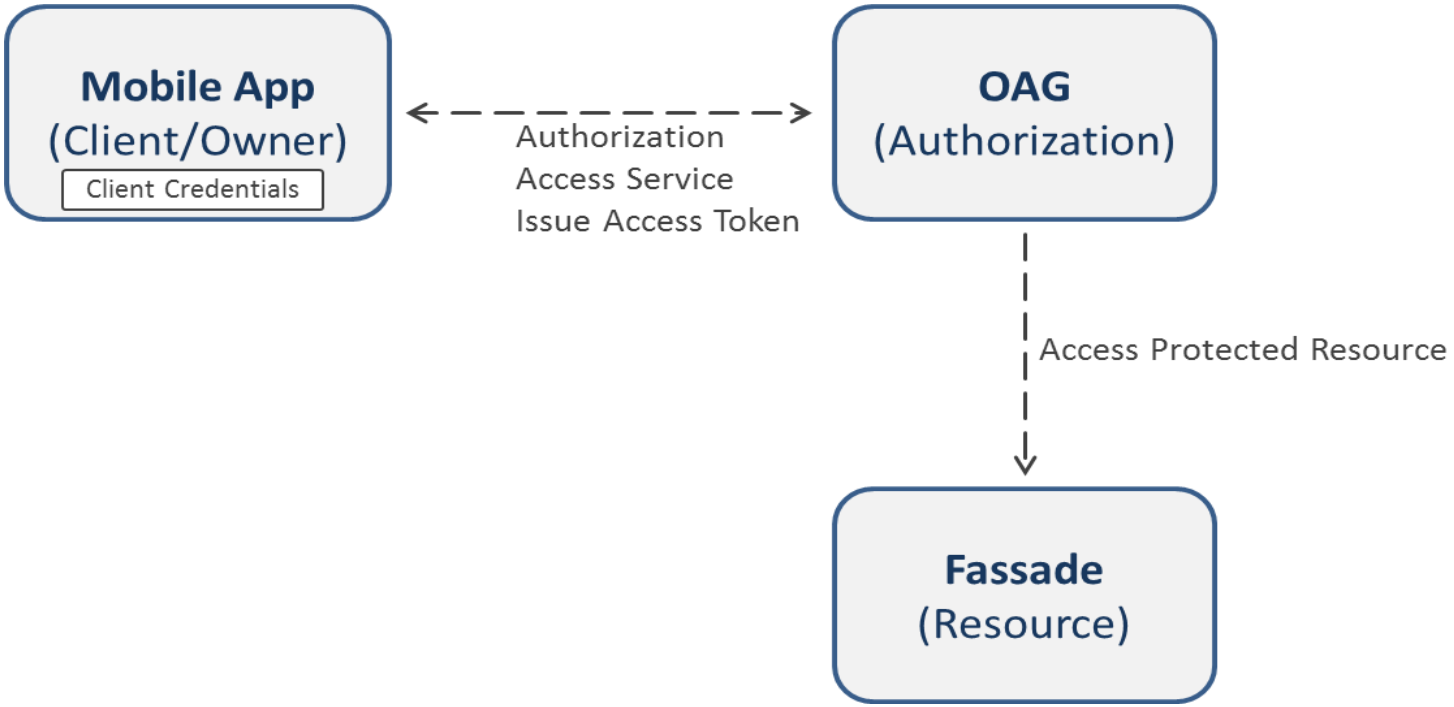
# OAuth 2.0

- **Protokoll zur Authorisierung von Web-Zugriffe auf**
- **OAuth Terminologie**
  - Resource Server
    - Enthält die zu schützende Ressourcen.
  - Resource Owner
    - Entität die die Ressourcen verwaltet und Zugriff auf diese gewähren kann.
  - Authorization Server
    - Autentisiert den Resource Owner und erstellt eine zeitlich beschränkte Autorisierung für den Zugriff auf die Ressource
  - Client
    - Möchte auf Ressourcen zugreifen, welche eigentlich für ihn nicht zugänglich sind.
    - Holt sich Genehmigung vom Resource Owner.

# Oauth (3 beiniges OAuth)



# OAuth (2 beiniges OAuth)



# OAuth bei mobilen Anwendungen

- **App ist Resource Owner**
  - Zugriff auf eigene API entspricht geschützten Daten
- **Authentication**
  - Client Credentials
    - Authentifizierung der App
    - werden mit der App ausgeliefert
  - Resource Owner Password Credentials
    - Authentifizierung des Benutzer
    - notwendig für Zugriff auf benutzerspezifische Ressourcen
    - zusätzlich zu Client Credentials
    - können im Key-Store abgelegt werden
- **Access Token**
  - kurzlebig, nur für Session

# OAG

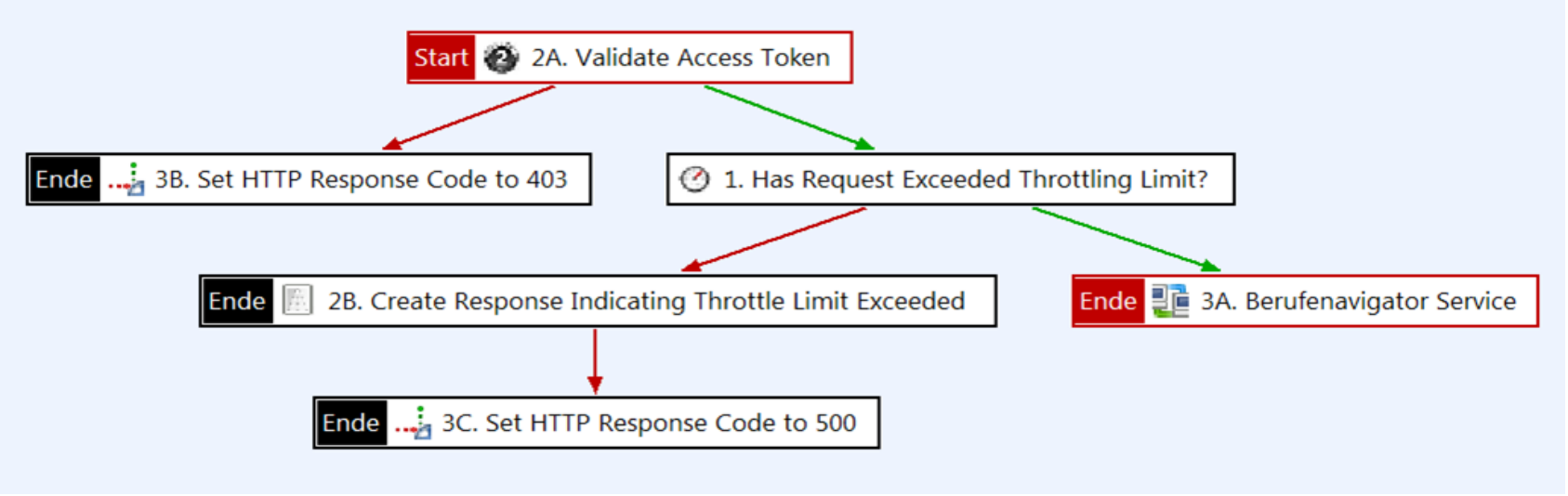
- **API Gateway und XML-Firewall**
  - Absicherung von SOAP und REST Schnittstellen
  - Proxy für Zugriffe auf Services
  - Kann Prüfungen auf XSS, SQL-Injection, XML-Bombs, Viren etc. ausführen
  - Einige Schutzmechanismen vor DDoS Angriffen
- **enthält OAuth Implementierung**
- **ist Authorization Server und Resource Server in einem**
  - Erstellt Tokens über eigenen Authentisierungsendpunkt
  - Prüft Token bei Zugriff auf die Resource
    - Vorteil: REST API muss nicht für OAuth angepasst werden
    - Nachteil: Schützt nur vor externen Zugriffen welche über den OAG geleitet werden
- **OAuth kann mit anderen Regeln kombiniert werden**
- **Einfach zu implementieren**

# Code-Beispiel:
## Request Token; Zugriff auf Ressource

```
// Request Access Token
Form form = new Form();
form.add("client_id", "9ae1ed85-cf1c-4513-a070-2c880948e80a");
form.add("client_secret", "49485d9f-5566-4d24-bf58-1e37ed9444e9");
form.add("grant_type", "client_credentials");

JSONObject result = client.resource("https://localhost:8089")
        .path("api/oauth/token")
        .entity(form, MediaType.APPLICATION_FORM_URLENCODED)
        .post(JSONObject.class);

token = result.getString("access_token");

// Access Resource
Occuption o = client.resource("http://localhost:9001")
          .path("berufenavigator/1.1/occupation/5427")
          .header("Authorization", "Bearer " + token);
          .accept(MediaType.APPLICATION_JSON)
          .get(Occupation.class);
```

# OAG Policy

- **Policy für App**
  - Token prüfen
  - Throttling (ClientID, Client IP)
  - Routing API Version



# Entwicklungsumgebung

- **Testen der mobiler Anwendungen**
  - Emulator
    - Implementierung / Entwicklertest unter „echtem“ Plattform OS
  - physikalisches Gerät
    - erfordert (meist) W-LAN AP → IT Sicherheit
    - Frontend, Integration, Abnahmetest
  - virtuelle Geräte per Cloud Service
    - fachlicher Test,  Fehleranalyse von gerätespezifischen Problemen
- **Testen der Fassade**
  - per Unit-Test mit Jersey Client
  - mit Mock für Backend-Service komplett in IDE testbar

# REST/JSON API – Best Practices I

- **URL-Pattern**
  - http://{server|domain}/{context}/{version}/{resource}
  - Beispiel: http://api.arbeitsagentur.de/berufenavigator/1.0/occupation/10023
- **Pfad adressiert Ressource  (Entität) bzw. Relation**
  - /occupations/{id}
  - /occupations/{id}/videos
  - Ressource nur im Plural
- **Request Method definiert CRUD Aktion**
  - Get  nur Lesen → keine Änderung des Zustands
  - Post  neu anlegen
  - Delete  löschen
  - Put  aktualisieren

# REST/JSON API – Best Practices II

- **andere Aktionen**
  - Aktion als Attribut bzw. Feld einer Ressource
  - PUT /customers/7542/orders/123/prioritize
- **Paging/Sorting**
  - GET /customers/7542/orders?sort=-date&offset=0&limit=20
  - Gesamtanzahl in HTTP-Header: X-Total-Count
- **Suche**
  - Parameter in Request-Query (REST Lehre)
    - GET /customers/7542/orders?date_from={date}&date_to={date}&min_amount=...
    - bei vielen Suchparametern unübersichtlich
    - Umsetzung am Server umständlich (Jersey API)
  - ➢ Suche als Aktion mit Objekt
    - POST /customers/7542/orders/search
    - Suchparameter werden als JSON Objekt abgebildet
    - Linksetzung nicht möglich, unkritisch da API für mobile Anwendung
    - Einsatz bei komplexer Filterung (eher ‚matching' als Suche)

# Zusammenfassung / Fazit

- **REST APIs brauchen eine eigene Architektur**
  - Schutz der Backend-Systeme vor unvorhergesehener Last
  - Entkoppliung des Lebenszyklus der App von den Diensten
    - Agile App Entwicklung
    - Häufige Anpassungen an neue Geräte
  - Limitierung der exponierten Dienste
    - Minimirung der Angriffsvektoren

- **Implementierung als Proof of Concept**
  - OAuth kann sehr einfach mit dem OAG implemetiert werden
  - Eigene REST API kann von den App Entwicklern selbst definiert werden (weniger Abstimmungsbedarf mit den Fachanwendungen)
  - App-Entwicklung braucht eigene Entwickluings- und Testumgebungen mit neuen Anforderungen, die im Geschäftsumfeld meist blockiert sind