Microservice Architecture

**Service 1**

**Service 2**

**Gateway**

**Client**

**FrontendAPP**

**Service Discovery**

# API Gateway

Security

Authentication

Adaptor

(Service Discovery)

Authorization

Static Content

Cache

Protocol Adapter

Monitor

Router & Load Balancer

Microservice 1

Microservice 2

Multiple copies

Multiple copies

Multiple copies

New Deploy (Canary Release)

Self manged API gateways:

• Apache

• HAProxy

• NGINX

• Spring Cloud Gateway

• Zuul

# Spring Cloud Security

# OpenID and OAuth

Example: Ordering stuff form ecommerce website.

Step 1: User accesses web app and does not have a session with it.

Step 2: Web app redirects user to the authorize endpoint on the auth server. The redirect URL contains the scopes openID and order.me

This means that the web app is requesting a identity (openID) and place orders on the users behalf (order.me)

Web App

1,9 10 X

Male icon 8 2 5 6 7

12

Auth Server

3,4 11,13,14

REST API

Step 3: Auth server redirects user to its login page because the user isn’t logged in.

Step 4: User logs in, starts a session with the auth server, and is redirected back to the authorize endpoint.

Control is given back to the user, who sees a page asking if the user permits the web app to access their identity and manage their orders on their behalf.

Step 5: User authorizes access. Auth server redirects the user back to the web app with a onetime code in the query params of the redirect.

Step 6: Web App hits the token endpoint with onetime code in the query params.

Auth server validates the code.

Step 7: Auth server responds with an access token (random string), and ID token (signed JWT).

Web app verifies the ID token, consumes its contents, and starts an authenticated session, and saves the access token in session

Step 8: Web app now gives control back to the user and responds with an order form.

Step 9: User fills out and submits the order form

Step 10: The web app submits the order to the REST API with the access token that was stored in session.

Step 11: The REST API needs to validate the token. It sends the token to the Auth server’s token verification endpoint.

Step 12: The Auth server responds with the permissions (scopes) that the token grants. The REST API now knows that the request is authorized.

Step 13: But wait! Before saving the order, the REST API wants to populate it with other user information not contained in the requests, e.g. address, phone number. The REST API make a request with same token to the /user info endpoint.

Step 14: The auth server responds with the user’s information. The REST API can now save order.

Step X: control is finally given back to user.

OpenID Connect

JSON Web Token (JWT)

JSON Web Algorithm (JWA) & JSON Web Key (JWK)

JSON Web Signature (JWS)

JSON Web Encryption (JWE)

OAuth2

# Oauth2

Oauth2 is an **authorisation** **framework** that enables **applications** to obtain **limited access to user accounts** on an **HTTP service**.

the main aim is to allow a user to **delegate** access to a user owned resource. Delegating it to an application.

And now the scenario for OAuth:

* User is on example.com and wants to import his contacts from mycontacts.com
* example.com (the “Consumer” in OAuth lingo) redirects the user to mycontacts.com (the “Service Provider”)
* User authenticates himself to mycontacts.com (which can happen by using OpenID)
* mycontacts.com asks the user whether he wants to authorize example.com to access his contacts
* User makes his choice
* mycontacts.com redirects the user back to example.com
* example.com retrieves the contacts from mycontacts.com
* example.com informs the user that the import was successful

# OpenID Connect

First the scenario for OpenID:

* User wants to access his account on example.com
* example.com (the “Relying Party” in OpenID lingo) asks the user for his OpenID
* User enters his OpenID
* example.com redirects the user to his OpenID provider
* User authenticates himself to the OpenID provider
* OpenID provider redirects the user back to example.com
* example.com allows the user to access his account

# User Account and Authentication (UAA) Server (for intra App APIs)

UAA provides enterprise-scale identity management features.

UAA provides identity-based security for applications and APIs. It supports open standards for authentication and authorization, including the following:

* OAuth
* OpenID Connect
* SAML
* LDAP
* SCIM

The major features of UAA include the following:

* User Single Sign-On (SSO) using federated identity protocols
* API security with OAuth
* User and group management
* Multi-tenancy support
* Support for [JWT](https://jwt.io/) and **opaque** as a token format
* Token revocation
* Operational flexibility  
  + Operate and run as a [BOSH release](http://bosh.io/releases/github.com/cloudfoundry/uaa-release?all=1), which allows multi-cloud deployment capabilities
  + Push as an app to Cloud Foundry
* Database flexibility, including support for MySQL, Postgres, and SQL Server
* Auditing, logging, and monitoring
* Token exchange for SAML and JWT bearers
* [Rest APIs](https://docs.cloudfoundry.org/api/uaa/index.html) for authentication, authorization, and configuration management

# Gateway Security Architecture (for inter APIs)

Authentication Server

Map<token, Detail>

|  |  |  |
| --- | --- | --- |
| Token | Date | Expire |
| eGVsc3IzYRhji=.  zH3tD5KZJHfds.  AKyt7ygui887u | 15.7.19 | 10:15:12 |
| yLhjo776… | … | ….. |

Header

Username: Rick

Password:12356

UI

GET /token

GET /api1

Header

Authorization

Bearer

eGVsc3IzYRhji=.

zH3tD5KZJHfds.

AKyt7ygui887u

Header

Authorization

Bearer

eGVsc3IzYRhji=.

zH3tD5KZJHfds.

AKyt7ygui887u

JWT

eGVsc3IzYRhji=.

zH3tD5KZJHfds.AKyt7ygui887u

Secret

751AD7J8AQPTDXC0O8C83

Authorisation Server

Map<usr\_role, api\_list>

|  |  |
| --- | --- |
| USR\_ROLE | API\_LIST |
| Admin | AP1, API2, API3 |
| Dev | API4 |
| End User | API1, API2 |

Secret

751A…….

Secret

751A…….

API3

API1

API2

Secret

751A…….