Class 9 – Spring Boot Microservice Enterprise Applications with Java

Contents

Review

- Class8
- Q&A

OAuth[1]

OAuth enables third-party applications to obtain limited access to a resource without the owner having to share their credentials.

There are two version 1.0a and 2.0. The fundamental problem they are trying to solve is the same, however they are very different implementations and there is much controversy and specification authors disagreed publicly[2].

We will focus on the concepts and look at 1.0a implementation as an example since that is simpler. 2.0 has numerous variations while it is being adopted slowly 1.0a is good for us to study the concepts.

Full RFC: http://tools.ietf.org/html/rfc5849 (It is pretty easy to read if you are interested).

Actors

- Resource-owner: the person that owns the "resource" – you are owner of your facebook/twitter *content* etc

[1] http://oauth.net/
[2] http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/

© 2017 Rahul Agarwal, All Rights Reserved. Please do not publish on the Internet or distribute.

- Authorization Server/Resource Server: server(s) that can validate the resource-owner's credentials (typically username/password) and where resources are held – facebook/twitter etc
- Client: a third-party that the resource-owner wants to authorize to access their resources from the resource server on their behalf

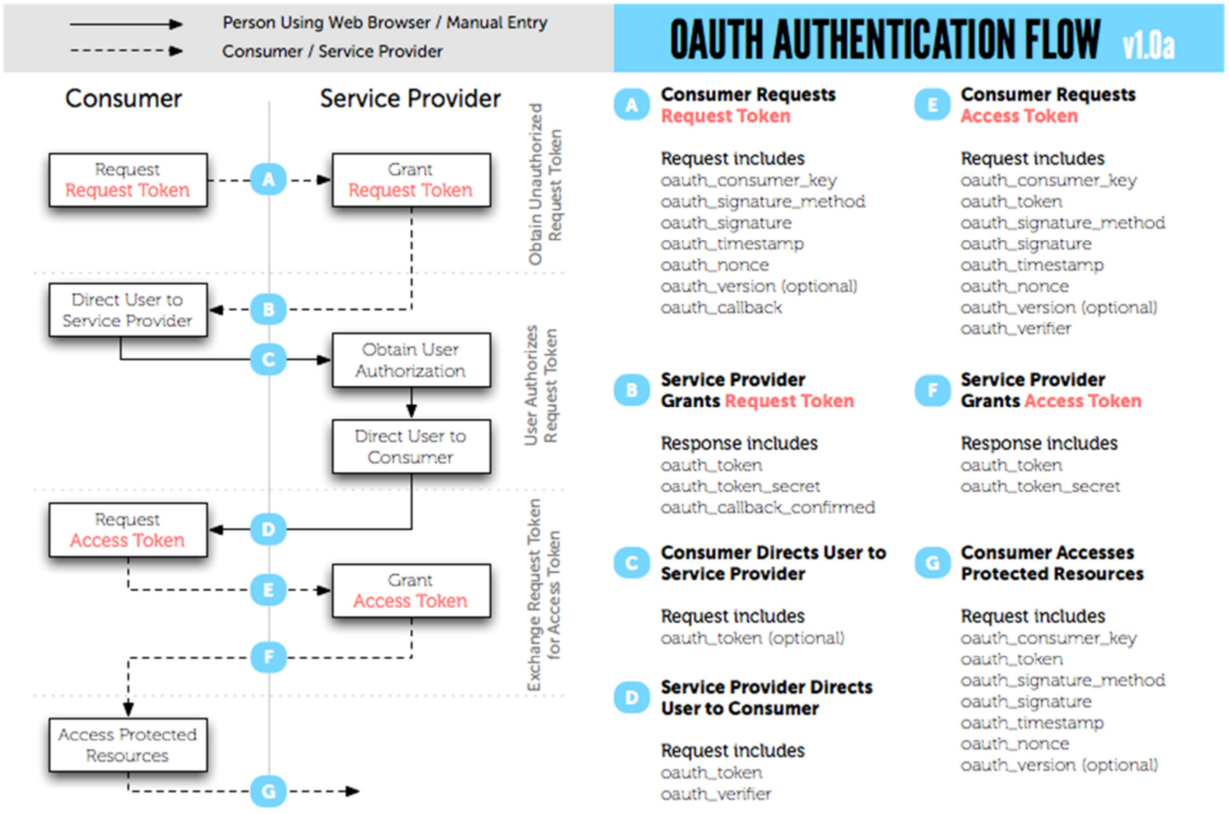
Terminology

- Client - An HTTP client (per [RFC2616]) capable of making OAuth- authenticated requests
- Server - An HTTP server (per [RFC2616]) capable of accepting OAuth-authenticated requests
- Protected resource - An access-restricted resource that can be obtained from the server using an OAuth-authenticated request
- Resource owner - An entity capable of accessing and controlling protected resources by using credentials to authenticate with the server
- Credentials - Credentials are a pair of a unique identifier and a matching shared secret. OAuth defines three classes of credentials: client, temporary, and token, used to identify and authenticate the client making the request, the authorization request, and the access grant, respectively
- Token - A unique identifier issued by the server and used by the client to associate authenticated requests with the resource owner whose authorization is requested or has been obtained by the client. Tokens have a matching shared-secret that is used by the client to establish its ownership of the token, and its authority to represent the resource owner

Let us understand the flow:

http://s3.pixane.com/Oauth_diagram.png

© 2017 Rahul Agarwal, All Rights Reserved. Please do not publish on the Internet or distribute.


Person Using Web Browser / Manual Entry
Consumer / Service Provider
OAUTH AUTHENTICATION FLOW v1.0a
Consumer
Service Provider
Request Request Token
Grant Request Token
Obtain Unauthorized Request Token
Direct User to Service Provider
Obtain User Authorization
Direct User to Consumer
User Authorizes Request Token
Request Access Token
Grant Access Token
Exchange Request Token for Access Token
Access Protected Resources
A
B
C
D
E
F
G

A Consumer Requests Request Token

Request includes
oauth_consumer_key
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)
oauth_callback

B Service Provider Grants Request Token

Response includes
oauth_token
oauth_token_secret
oauth_callback_confirmed

C Consumer Directs User to Service Provider

Request includes
oauth_token (optional)

D Service Provider Directs User to Consumer

Request includes
oauth_token
oauth_verifier

E Consumer Requests Access Token

Request includes
oauth_consumer_key
oauth_token
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)
oauth_verifier

F Service Provider Grants Access Token

Response includes
oauth_token
oauth_token_secret

G Consumer Accesses Protected Resources

Request includes
oauth_consumer_key
oauth_token
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)

Break

Scaling Web Applications

Architectural goals

- Capacity – this is probably the first goal that comes to mind. How many customers can you serve – for example in a Too Big to Fail Bank branch's ability to handle the number of customers depends on space, number of bank officers etc. In the online world the healthcare.gov not able to signup customer is a classic example
- Availability – what if you went to Amazon.com and unable to purchase something
- Reliability – did Amazon charge your credit card but did not ship the order? Partial failures and recovery
- Performance – even at speed of light it takes some milliseconds for data to travel (latency). How to minimize number of round trips
- Redundancy/Geographic distribution
- Maintainability – can you easily maintain the system and build new features/enhancements
- Configurability
- Security

Challenges

© 2017 Rahul Agarwal, All Rights Reserved. Please do not publish on the Internet or distribute.

- State – maintaining state on the server (for each connected user agent)
- Hardware – physical and technical limitations of the hardware (CPU/cores/RAM/Disk speed etc)
- In the n-tier model each layer can be hosted on separate physical hardware
- Network – different components within your n-tiers must communicate with each other and how they are configured (same network segment/distributed etc) can impact how many microseconds get added to each transaction
- In most cases (data intensive and transactional systems) the database is the bottleneck (CAP Theorem)
- The business layer (application servers) can scale more easily

Data store – I prefer to use the term “data store” which can mean both relational and non-relational. A “database” usually implies relational.

Caching

http://en.wikipedia.org/wiki/Cache_(computing)

“In computer science, a cache is a component that transparently stores data so that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower. Hence, the greater the number of requests that can be served from the cache, the faster the overall system performance becomes.”

There are all kinds of strategies for caching and you could do a PhD on the subject so we will be keeping this simple!

Caching can be applied in all kinds of situations in a web application. It is the most popular first step to relieve load on all kinds of resources.

Let us talk about a few topics:

- Connection pooling – HTTP and DB
- Distributed cache
- REST - ETag[3]

Brewer’s CAP Theorem

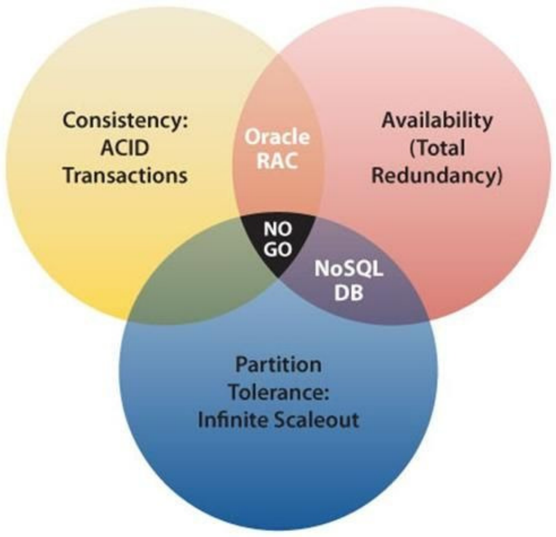
http://en.wikipedia.org/wiki/CAP_theorem

1. Consistency - Everyone sees the same data
2. Availability - “always on”, node failure tolerance

[3] http://en.wikipedia.org/wiki/HTTP_ETag

© 2017 Rahul Agarwal, All Rights Reserved. Please do not publish on the Internet or distribute.

3. Partition Tolerance - Reads & Writes work seamlessly even when data is split into subsets and there is a partial outage


Consistency: ACID Transactions
Oracle RAC
Availability (Total Redundancy)
NO GO
NoSQL DB
Partition Tolerance: Infinite Scaleout

- CA
  - Single site clusters (easier to ensure all nodes are always in contact)
  - When a partition occurs, the system blocks
- CP
  - Some data may be inaccessible (availability sacrificed), but the rest is still consistent/accurate – e.g. sharded database
- AP
  - System is still available under partitioning, but some of the data returned my be inaccurate – e.g. DNS, caches, Master/Slave replication
  - Need some conflict resolution strategy

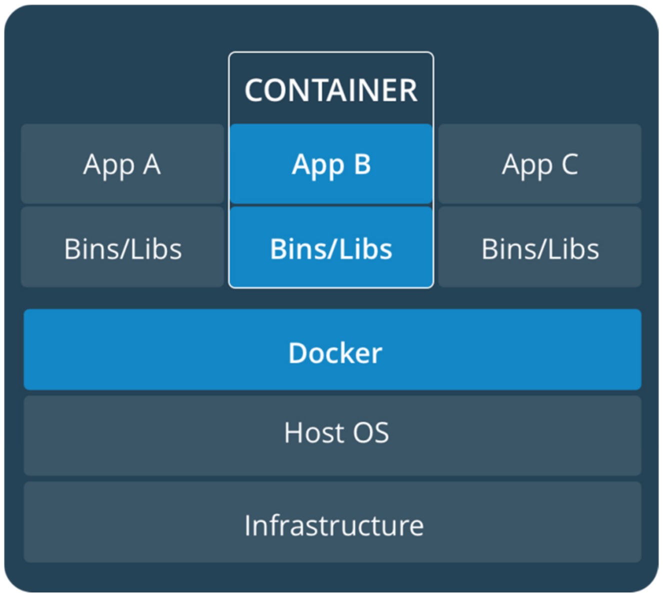
Docker[4]

Docker is the most popular "container" solution.

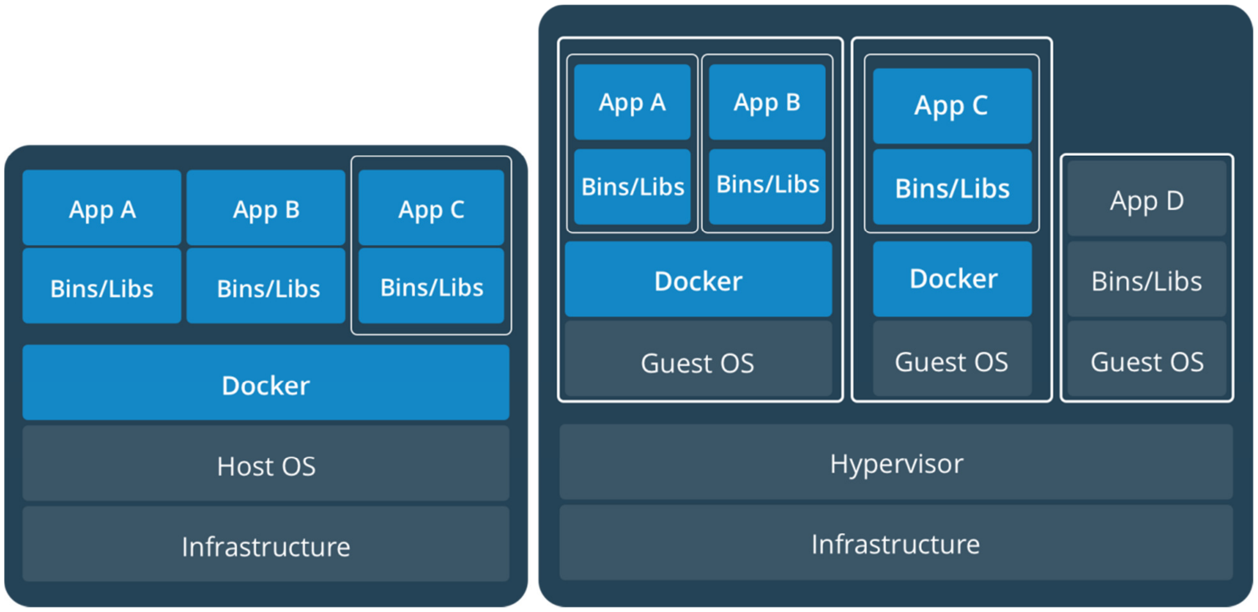
> "Using containers, everything required to make a piece of software run is packaged into isolated containers. Unlike VMs, containers do not bundle a full operating system - only libraries and settings required to make the software work are needed. This makes for efficient, lightweight, self-contained systems and guarantees that software will always run the same, regardless of where it's deployed"

[4] https://www.docker.com/what-docker

© 2017 Rahul Agarwal, All Rights Reserved. Please do not publish on the Internet or distribute.



This can also run together with Virtual Machines

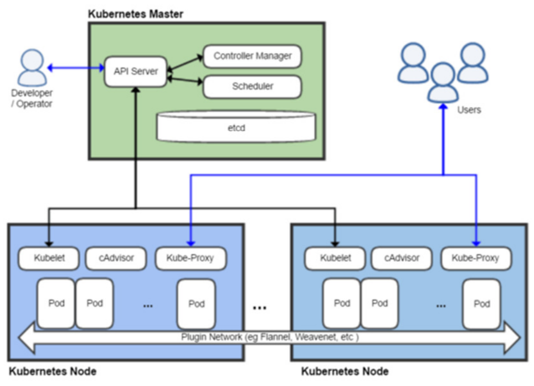


Kubernetes[5]

System to help orchestrate container deployment.

[5] https://kubernetes.io/

© 2017 Rahul Agarwal, All Rights Reserved. Please do not publish on the Internet or distribute.



To learn more https://github.com/rahulaga/craft-kubernetes-workshop

© 2017 Rahul Agarwal, All Rights Reserved. Please do not publish on the Internet or distribute.