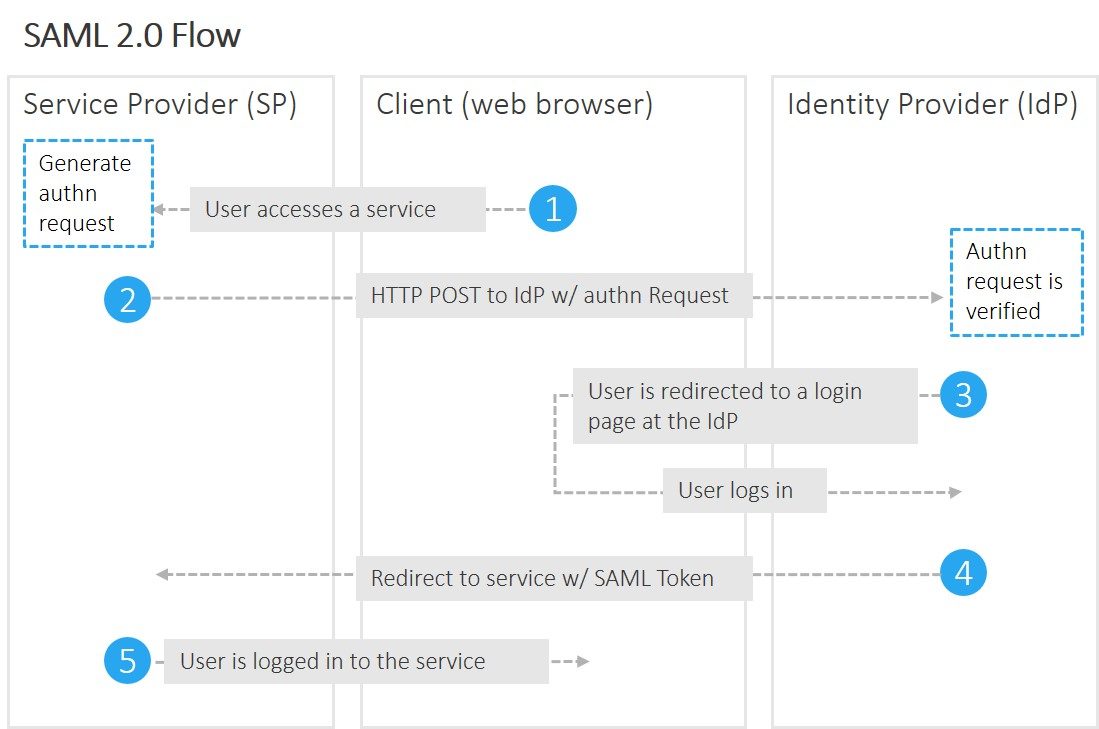
**Single Sign On**

The SAML workflow

1. An end user clicks on the “Login” button on a file sharing service at [example.com](http://www.example.com/). The file sharing service at example.com is the Service Provider, and the end user is the Client.
2. To authenticate the user, example.com constructs a SAML Authentication Request, signs and optionally encrypts it, and sends it directly to the IdP.
3. The Service Provider redirects the Client’s browser to the IdP for authentication.
4. The IdP verifies the received SAML Authentication Request and if valid, presents a login form for the end user to enter his username and password.
5. Once the Client has successfully logged in, the IdP generates a SAML Assertion (also known as a SAML Token), which includes the user identity (such as the username entered before), and sends it directly to the Service Provider.
6. The IdP redirects the Client back to the Service Provider
7. The Service Provider verifies the SAML Assertion, extracts the user identity from it, assigns correct permissions for the Client and then logs him in to the service



All done. Note that the SP never processed or even saw the Client’s credentials. Here we succeeded logging in with two redirects. The rude awakening comes when we want to move from a web application to a native one, such as a mobile app.

The devil lies within the step 6. There are two ways the IdP can redirect the Client back to the SP: HTTP Redirect and HTTP POST. The first one is not recommended, since the length of the HTTP Redirect URL is limited, and there is no standard telling what exactly is the maximum length. The second way avoids data size issues, but is very archaic to use. Either the user has to click a button to submit the POST form, or it has to be automated using JavaScript. This isn’t that big of a problem when dealing with web applications, but on a mobile application the authentication would simply fail, as the applications do not have access to the POST data and for a good reason.

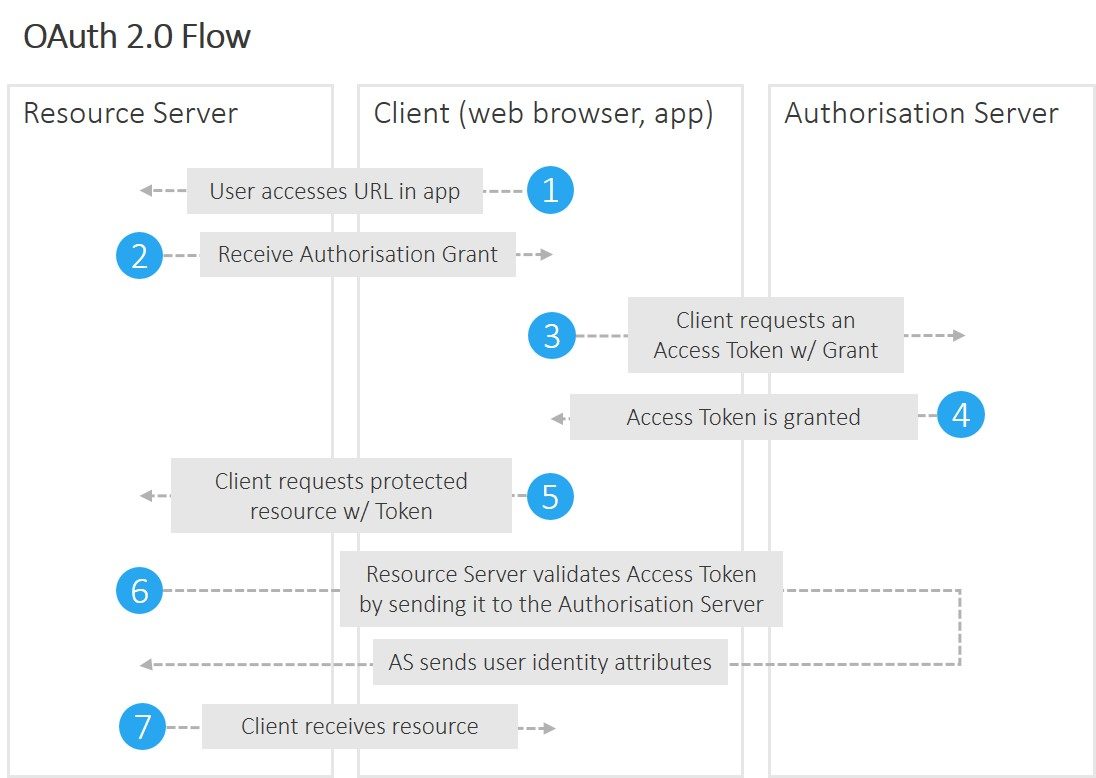
There are possible (complex) workarounds, but the best solution might be to use OAuth instead.

**The OAuth workflow**

Critically, OAuth doesn’t assume that the Client is a web browser.

The example workflow proceeds now as follows:

1. An end user clicks on the “Login” button on a file sharing service at [example.com](http://www.example.com/). The file sharing service at example.com is the Resource Server, and the end user is the Client.
2. The Resource Server presents the Client with an Authorisation Grant, and redirects the Client to the Authorisation Server
3. The Client requests an Access Token from the Authorisation Server using the Authorisation Grant Code
4. The Client logs in to the Authorisation Server, and if the code is valid, the Client gets an Access Token that can be used request a protected resource from the Resource Server
5. After receiving a request for a protected resource with an accompanying Access Token, the Resource Server verifies the validity of the token directly with the Authorisation Server
6. If the token was valid, the Authorisation Server sends information about the Client to the Resource Server

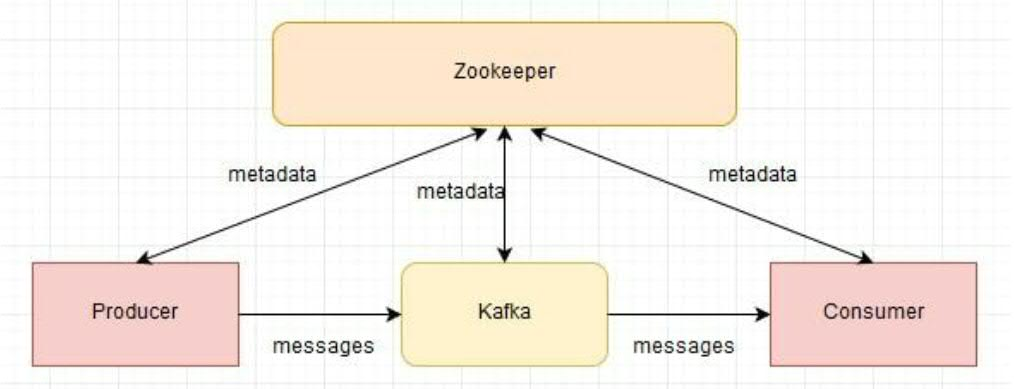


We can now skip the awkward HTTP POST dance, but only at the price of an additional round trip to the Authorisation Server. This is caused by the different underlying trust models and their embodiments, the tokens. SAML Assertions or “SAML tokens” contain the user identification information (which can be trusted because it is signed), while with OAuth the Resource Server needs to make additional round trip in order to authenticate the Client with the Authorisation Server.

What if you can’t choose between SAML and OAuth? Good news, one can always use **both**. In this scenario, the SAML Assertion can be used as an *OAuth Bearer Token* to access the protected resource. In addition, if the lack of authorisation is the only thing holding back on your OAuth implementation, be sure to check out *OpenID and OpenID Connect*, open standards that builds upon OAuth in order to provide just that.

My next blog is about how OpenID builds upon OAuth 2.0. Subscribe to our newsletter (on the top right) to get the latest news, blog releases, whitepaper publications etc.

**Kafka**



<https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html>

Kafka is distributed publish subscribe messaging system which is more fast,durable,fault tolerance,scalable and distributed.

**Why docker is required ?**

The Docker**Containers** allow the developer/sysadmin to bundle an application with all needed components (libraries and other resources) and to deliver it as an independent and single package.

**From a certain point of view** a container is not so much different from a **virtual machine**. But, instead of creating a full operating system, a Docker Container has just the minimum set of operating system software needed for the application to run and rely on the host Linux Kernel itself.

This allow a huge boost on performance (compared to a virtualized system you can save up to 8% processing power) and a reduced memory and disk footprint for your application.

**Docker is Open Source**: everybody can contribute, customize and extend Docker.

**What is the advantage to move application to cloud?**

Your application is experiencing increased traffic and it’s becoming difficult to scale resources on the fly to meet the increasing demand.

You need to reduce operational costs while increasing the effectiveness of IT processes.

Your clients require fast application implementation and deployment and thus want to focus more on development while reducing infrastructure overhead.

You’d like to build a widely distributed development team. Cloud computing environments allow remotely located employees to access applications and work via the Internet.

Capex to Opex: Cloud computing shifts IT expenditure to a pay-as-you-go model, which is an attractive benefit, especially for startups.

**What will you think to split monolithic to microservice?**

**Scaling**: In general we create Microservices to build a system. It is easier to scale up the Microservice that is being used more.

**Resilience**: In Microservice architecture, if one service goes down, it may not affect the rest of the system.

**Technology Mix**: These days technology is changing everyday. With Microservices approach, you can keep using the latest technology for your new Microservices.

**Easy Deployment**: Microservices architecture, if done correctly, helps in making the deployment process smooth.

**Independent Development** – All microservices can be easily developed based on their individual functionality

**Independent Deployment** – Based on their services, they can be individually deployed in any application

**12 Factor app?**

**1. Code base**  
Use one [codebase](https://whatis.techtarget.com/definition/codebase-code-base), even when building cross-platform apps. Address the needs of specific devices with [version control](https://whatis.techtarget.com/definition/version-control).

**2. Dependencies**  
Explicitly declare and isolate all dependencies.

**3. Configuration**  
Don’t store config as constants in code. Instead, design the app to read its config from the environment.

**4. Backing Services**  
Treat back-end services as attached resources to be accessed with a URL or other locator stored in [config](http://12factor.net/config).

**5. Build, Release, Run**  
Strictly separate build and run stages.

**6. Processes**  
Execute the app as one or more [stateless](https://whatis.techtarget.com/definition/stateless) processes. Data that must be persistent should be stored in a stateful backing service.

**7. Port binding**  
Use port binding to export services.

**8. Concurrency**  
Scale out apps [horizontally](https://searchcio.techtarget.com/definition/horizontal-scalability), not [vertically](https://searchcio.techtarget.com/definition/vertical-scalability).

**9. Disposability**  
Use fast startups and graceful shutdowns to maximize robustness.

**10. Parity**  
Facilitate [continuous deployment](https://searchitoperations.techtarget.com/definition/continuous-deployment) by ensuring that development, staging, and production environments are as similar as possible.

**11. Logs**  
Treat [logs](https://whatis.techtarget.com/definition/log-log-file) as event streams. Logs should not be concerned with routing or storing the app’s output.

**12. Admin processes**  
Run admin tasks as one-off processes from a machine in the production environment that’s running the latest production code.

**Mutual Authentication**

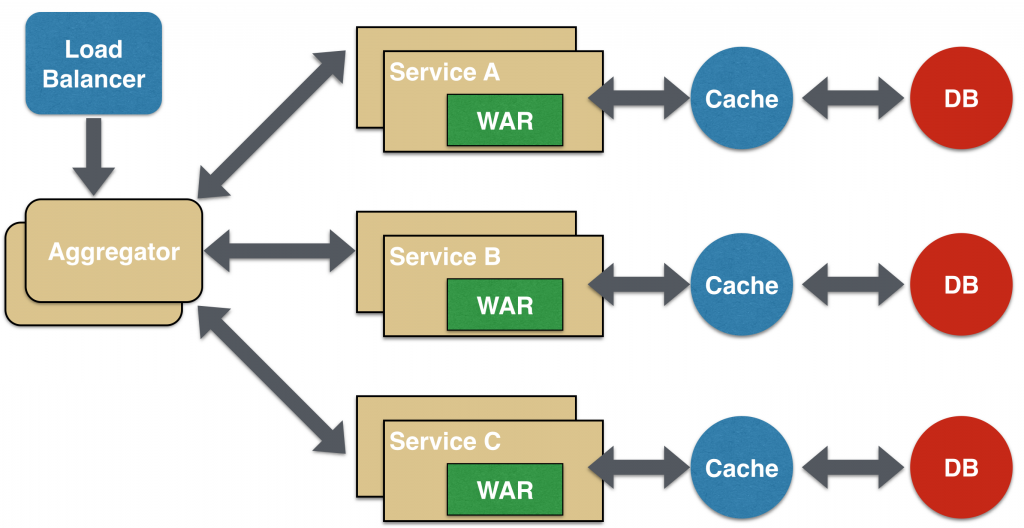
Mutual authentication can be achieved using **client certificate**.

**Microservice Design Pattern**

## Aggregator Microservice Design Pattern

The first, and probably the most common, is the aggregator microservice design pattern.

In its simplest form, Aggregator would be a simple web page that invokes multiple services to achieve the functionality required by the application. Since each service (Service A, Service B, and Service C) is exposed using a lightweight REST mechanism, the web page can retrieve the data and process/display it accordingly. If some sort of processing is required, say applying business logic to the data received from individual services, then you may likely have a CDI bean that would transform the data so that it can be displayed by the web page.



Another option for Aggregator is where no display is required, and instead it is just a higher level composite microservice which can be consumed by other services. In this case, the aggregator would just collect the data from each of the individual microservice, apply business logic to it, and further publish it as a REST endpoint. This can then be consumed by other services that need it.

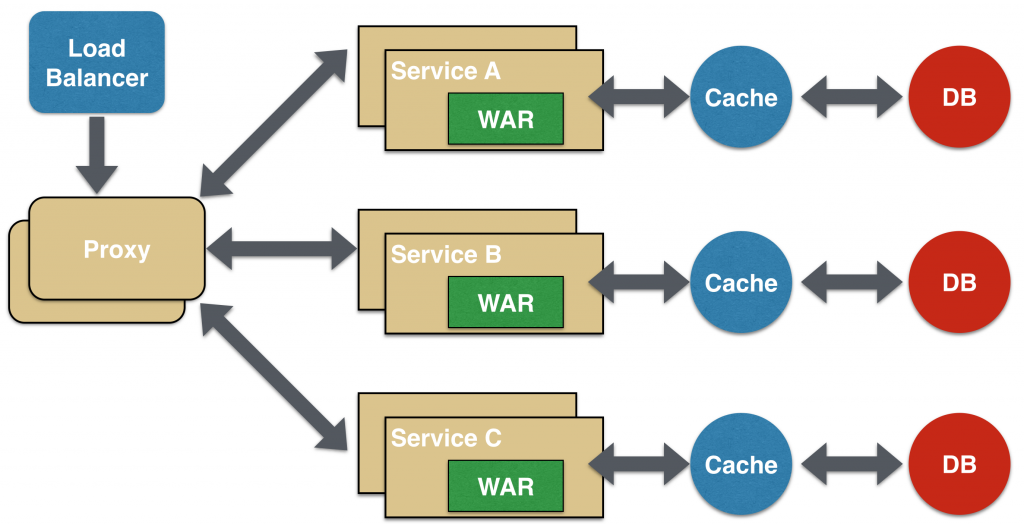
This design pattern follows the DRY principle. If there are multiple services that need to access Service A, B, and C, then its recommended to abstract that logic into a composite microservice and aggregate that logic into one service. An advantage of abstracting at this level is that the individual services, i.e. Service A, B, and C, and can evolve independently and the business need is still provided by the composite microservice.

Note that each individual microservice has its own (optional) caching and database. If Aggregator is a composite microservice, then it may have its own caching and database layer as well.

Aggregator can scale independently on X-axis and Z-axis as well. So if its a web page then you can spin up additional web servers, or if its a composite microservice using Java EE, then you can spin up additional WildFly instances to meet the growing needs.

## Proxy Microservice Design Pattern

Proxy microservice design pattern is a variation of Aggregator. In this case, no aggregation needs to happen on the client but a different microservice may be invoked based upon the business need.

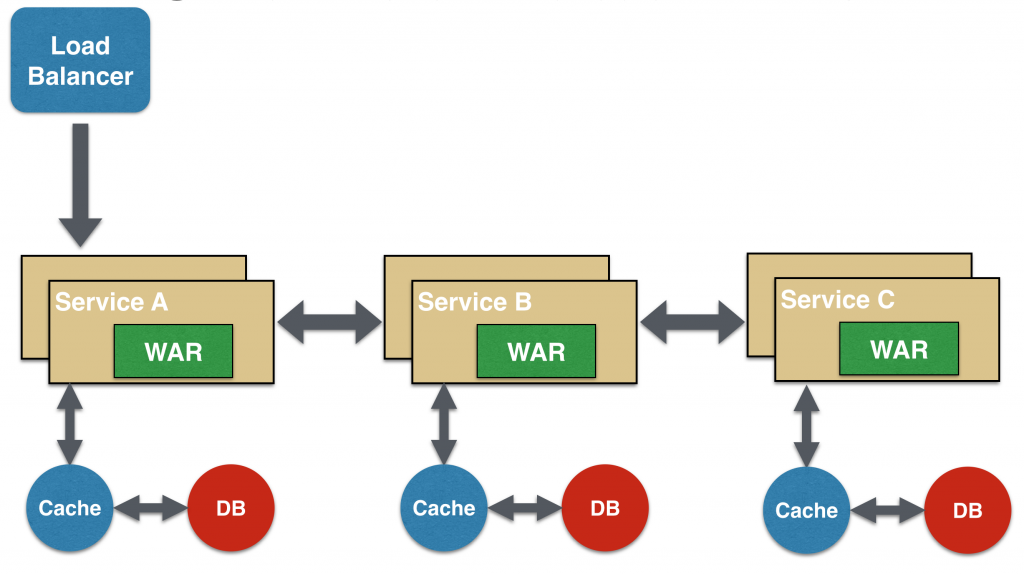
[](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-proxy.png)

Just like Aggregator, Proxy can scale independently on X-axis and Z-axis as well. You may like to do this where each individual service need not be exposed to the consumer and should instead go through an interface.

The proxy may be a dumb proxy in which case it just delegates the request to one of the services. Alternatively, it may be a smart proxy where some data transformation is applied before the response is served to the client. A good example of this would be where the presentation layer to different devices can be encapsulated in the smart proxy.

## Chained Microservice Design Pattern

Chained microservice design pattern produce a single consolidated response to the request. In this case, the request from the client is received by Service A, which is then communicating with Service B, which in turn may be communicating with Service C. All the services are likely using a synchronous HTTP request/response messaging.

[](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-chain.png)

The key part to remember is that the client is blocked until the complete chain of request/response, i.e. Service <-> Service B and Service B <-> Service C, is completed. The request from Service B to Service C may look completely different as the request from Service A to Service B. Similarly, response from Service B to Service A may look completely different from Service C to Service B. And that’s the whole point anyway where different services are adding their business value.

Another important aspect to understand here is to not make the chain too long. This is important because the synchronous nature of the chain will appear like a long wait at the client side, especially if its a web page that is waiting for the response to be shown. There are workarounds to this blocking request/response and are discussed in a subsequent design pattern.

A chain with a single microservice is called singleton chain. This may allow the chain to be expanded at a later point.

## Branch Microservice Design Pattern

Branch microservice design pattern extends Aggregator design pattern and allows simultaneous response processing from two, likely mutually exclusive, chains of microservices. This pattern can also be used to call different chains, or a single chain, based upon the business needs.

## [Microservice Branch Design Pattern](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-branch.png)

Service A, either a web page or a composite microservice, can invoke two different chains concurrently in which case this will resemble the Aggregator design pattern. Alternatively, Service A can invoke only one chain based upon the request received from the client.

This may be configured using routing of JAX-RS or Camel endpoints, and would need to be dynamically configurable.

## Shared Data Microservice Design Pattern

One of the design principles of microservice is autonomy. That means the service is full-stack and has control of all the components – UI, middleware, persistence, transaction. This allows the service to be polyglot, and use the right tool for the right job. For example, if a NoSQL data store can be used if that is more appropriate instead of jamming that data in a SQL database.

However a typical problem, especially when refactoring from an existing monolithic application, is database normalization such that each microservice has the right amount of data – nothing less and nothing more. Even if only a SQL database is used in the monolithic application, denormalizing the database would lead to duplication of data, and possibly inconsistency. In a transition phase, some applications may benefit from a shared data microservice design pattern.

In this design pattern, some microservices, likely in a chain, may share caching and database stores. This would only make sense if there is a strong coupling between the two services. Some might consider this an anti-pattern but business needs might require in some cases to follow this. This would certainly be an anti-pattern for greenfield applications that are design based upon microservices.

## [Microservice Branch Shared Data Design Pattern](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-branch-shared-data.png)

This could also be seen as a transition phase until the microservices are transitioned to be fully autonomous.

## Asynchronous Messaging Microservice Design Pattern

While REST design pattern is quite prevalent, and well understood, but it has the limitation of being synchronous, and thus blocking. Asynchrony can be achieved but that is done in an application specific way. Some microservice architectures may elect to use message queues instead of REST request/response because of that.

## [Microservice Async Messaging Design Pattern](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-async-messaging.png)

In this design pattern, Service A may call Service C synchronously which is then communicating with Service B and D asynchronously using a shared message queue. Service A -> Service C communication may be asynchronous, possibly using WebSockets, to achieve the desired scalability.

**Eureka Server**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

eureka:

instance:

hostname: localhost

client: #Not a client

registerWithEureka: false

fetchRegistry: false

@SpringBootApplication

@EnableEurekaServer

public class DiscoveryMicroserviceServerApplication {

public static void main(String[] args) {

SpringApplication.run(DiscoveryMicroserviceServerApplication.class, args);

}

}

**Eureka Client**

# Discovery Server Access

eureka:

client:

serviceUrl:

defaultZone: http://localhost:1111/eureka/

@SpringBootApplication

@EnableDiscoveryClient

public class AccountsMicroserviceServerApplication {

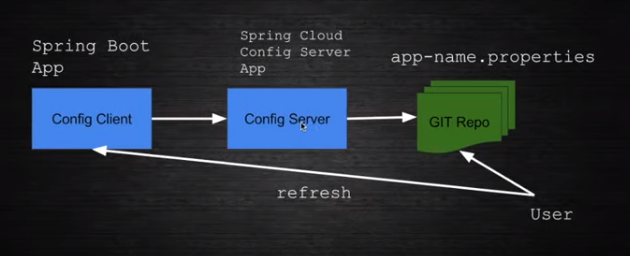
public static void main(String[] args) {

SpringApplication.run(AccountsMicroserviceServerApplication.class, args);

}

}

**Spring Config Server**



*- `config-client` - The actual App which is going to use some property*

*- `config-server` - The Config Server which is pointing to a GIT Repo to pick the property file.*

*- `config-client.properties` - The actual property file which is used by the config-server to push it to config-client. This is present inside the config-server folder (which is created as Git Repo)*

**ConfigClient Application**

@SpringBootApplication

public class ConfigClientApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigClientApplication.class, args);

}

}

**MessageResource.java**

@**RefreshScope**

@RequestMapping("/rest")

@RestController

public class MessageResource {

@Value("${message: Default Hello}")

private String message;

@GetMapping("/message")

public String message() {

return message;

}

}

**bootstrap.properties**

spring.application.name=config-client

spring.cloud.config.uri=http://localhost:8980

management.security.enabled=false

**ConfigServer Application**

@EnableConfigServer

@SpringBootApplication

public class ConfigClientApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigClientApplication.class, args);

}

}

**bootstrap.properties**

spring.cloud.config.server.git.uri={HOME}/Downlaod/ConfGit/Example

**To take latest properties, need to hit post actuator refresh url with blank body**