Spring 5

Spring Core

- Spring uses java config to configure POJOs.
- @Configuration, @Bean, new AnnotationConfigApplicationContext(Configuration.class), applicationcontext.getBean(), @Component, @Service, @Repository, @Controller, @Autowired on field, getter or constructor, @Primary/@Qualifier to resolve ambiguities, @Import to import bean definitions from other configuration files, @Scope, @PropertySource to read data from different source files which are on the classpath or file or url, @ComponentScan, MessageSource to read i18n data, @PostConstruct and

@PreDestroy, @Lazy(), @DependsOn(), @Profile() to activate certain beans only in certain spring profile

- ○ POJOs can aware of any of the spring IoC containers resources by implementing certain resource aware interfaces like BeanNameAware, EnvironmentAware, MessageSourceAware etc
- ○ Aspect oriented annotations like @Aspect, @Before("execution(* Calculator.add(..))"),pointcuts, @After, @AfterReturning, @AfterThrowing, @Around, @EnableAspectJAutoProxy, @Order(0), @Order(1)
- ○ The execution points matched by a pointcut are called join points. A pointcut is an expression to match a set of join points while an advice is the action to take at

a particular join point.

```
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
@Component
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }
}
```

- ≡ ○ @Pointcut("execution(* *.*(..))")
  Private void logOperation() {}.
  @AfterThrowing(pointcut= "logOperation()",throwing="e") public void logAfterThrowing(Join Point joinPoint, IllegalArgumentException e) {}
- ≡ ○ Use AOP for introductions for POJOs: with an AOP introductions we can make a class to dynamically implement two different interfaces by using two different already implemented abstract classes.

Introductions are the way to implement multiple inheritance in java.
@DeclareParents(value="com .apress.ArithmeticCalculaterImpl" defaultImpl=MinCalculatorImpl.cl ass) public MinCalculator minCalculator;
@DeclareParents(value="com .apress.ArithmeticCalculaterImpl" defaultImpl=MaxCalculatorImpl.c lass) public MaxCalculator maxCalculator; similarly we can introduce some states like counter to POJOs

- ≡ ○ Multithreading and concurrent programming through spring's TaskExecutor
- ≡ ○ POJOs can communicate between each other by using spring's application event. This will reduce the tight coupling. CheckoutEvent {}. @Autowired

ApplicationEventPublisher aep; aep.publishEvent(checkoutEvent) .@EventListener public void onApplicationEvent(CheckoutEve nt ce) {}.

Spring MVC

- ≡ ○ Dispatcher Servlet, Handler Mapping, Model, View Resolver, View
- ≡ ○ @Controller, @RestController, @RequestMapping, @PathVariable, @RequestBody, @RequestParam
- ≡ ○ Intercept request and response with spring's HandlerInterceptor. This interceptor can be applied only to set of urls.
- ≡ ○ Public class A implements HandlerInterceptor { public boolean preHandle(...) { return true; } public void postHandle(...) {} public void afterCompletion(...) {}. Before servlet is invoked, after

servlet is returnes and just before view is rendered, and finally after view rendered completely.

- ≡ ○ Instead of imolementing HandlerInterceptor it is better to extend HandlerInterceptorAdaptor class as it already provides all default implementation and we can only override the required one.
- ≡ ○ To register interceptor we need to implement WebMvcConfigurer. Public class config implements WebMvcConfigurer { @Override public void addInterceptor(InterceptorRegistry ir) { ir.addInterceptor(measurementbean());
- ≡ ○ We can also apply these interceptor to some set of urls by .addPathPatterns("/some url");
- ≡ ○ Changing users Locale.

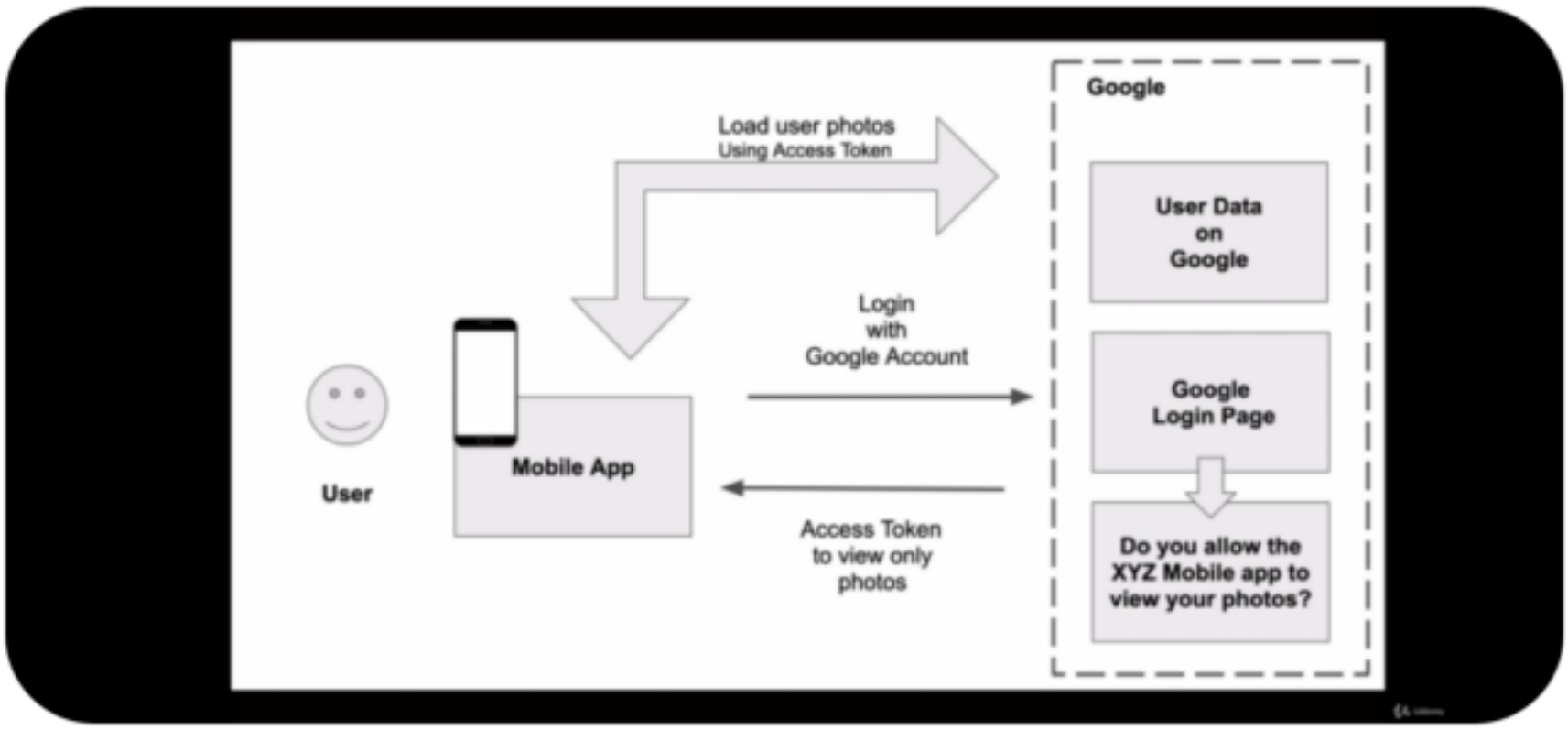
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
localeChangeInterceptor.setParamName("language");
return localeChangeInterceptor;
}. Value of the language can be send from ui

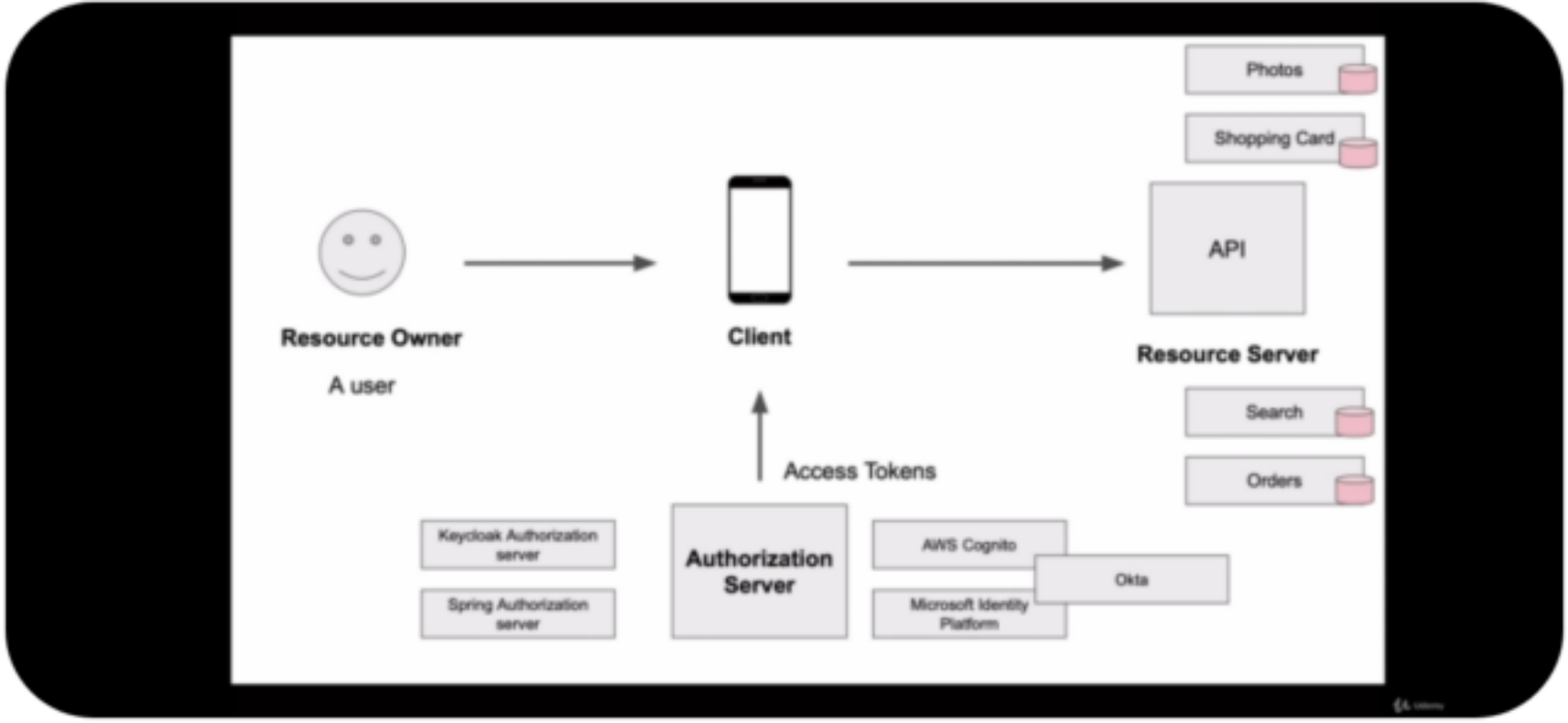
- Also we can externalize locale sensitive text message to different text files. @Bean public MessageSource messageSource() {
ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();messageSource.setBasename("messages");
return messageSource;
}. message_properties,

message_de.properties.

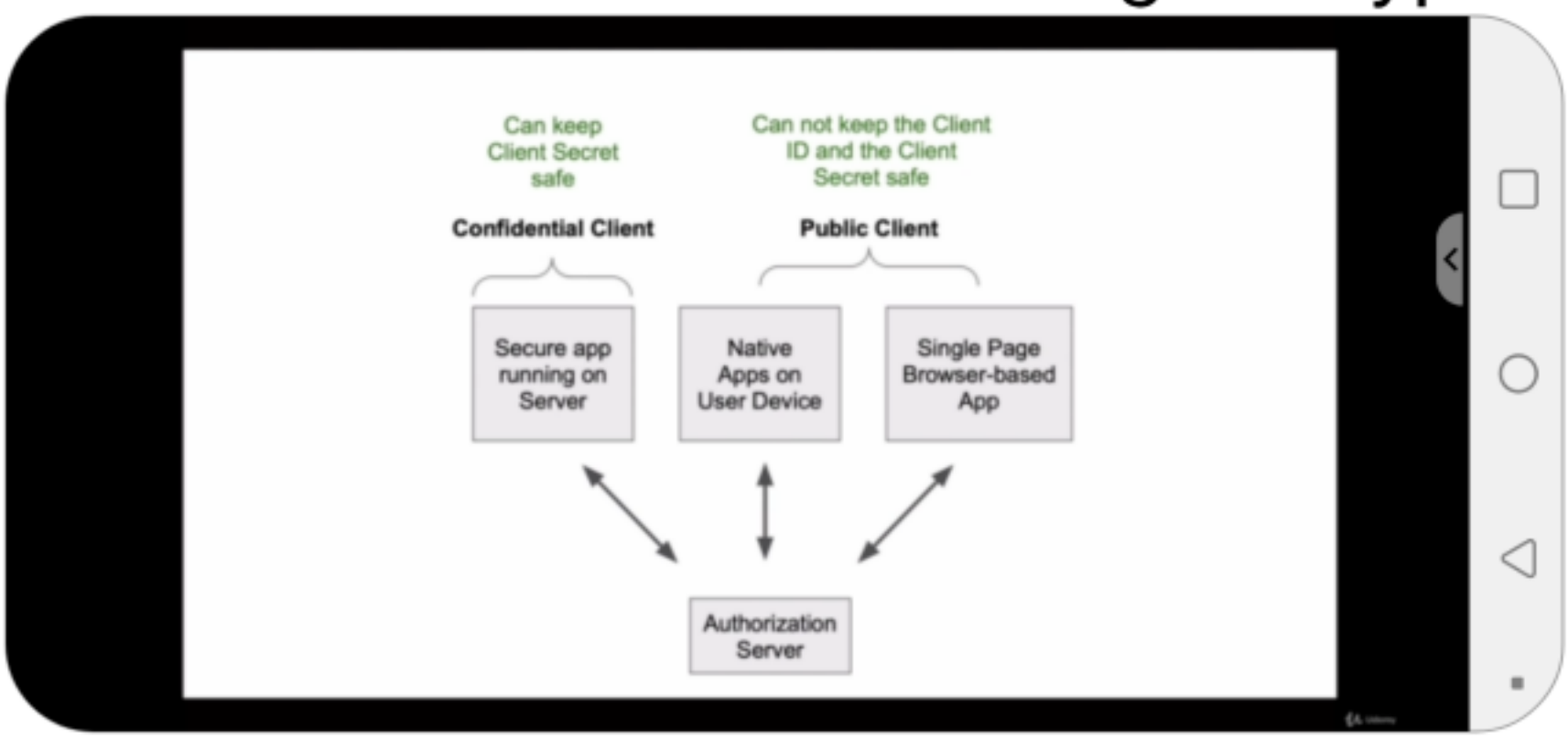
Spring Security:

- Authentication: is the process of verifying an identity.
- Authorization: is the process of verifying what someone is allowed to do. It allows an application to access users data or perform some operations on behalf of them.
- OAuth2: is an authorization or delegated authorization framework or standards



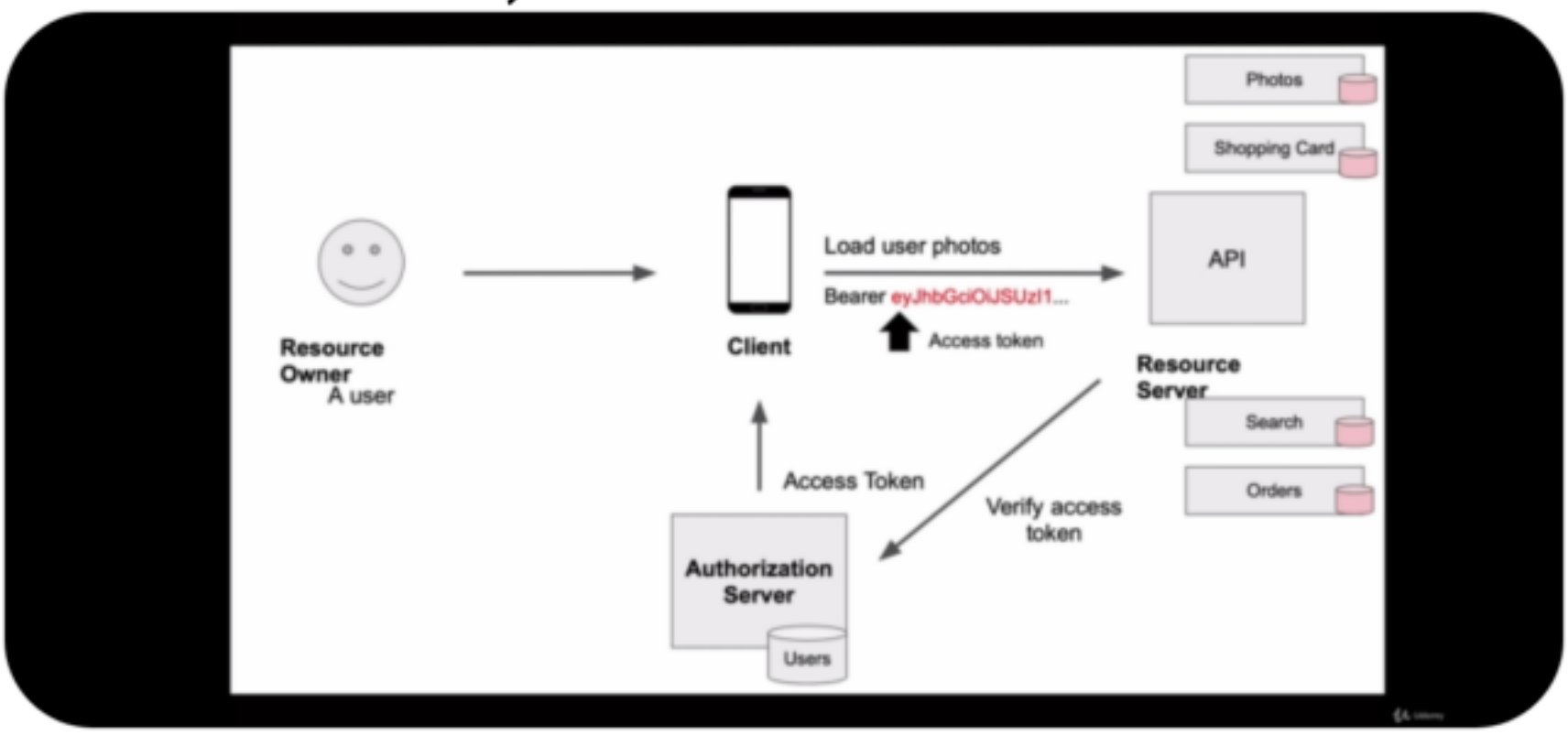


- Client application types: there are mainly 2 different types. Depending on these 2 types of client application authorization server provides different authorization flows or grant types

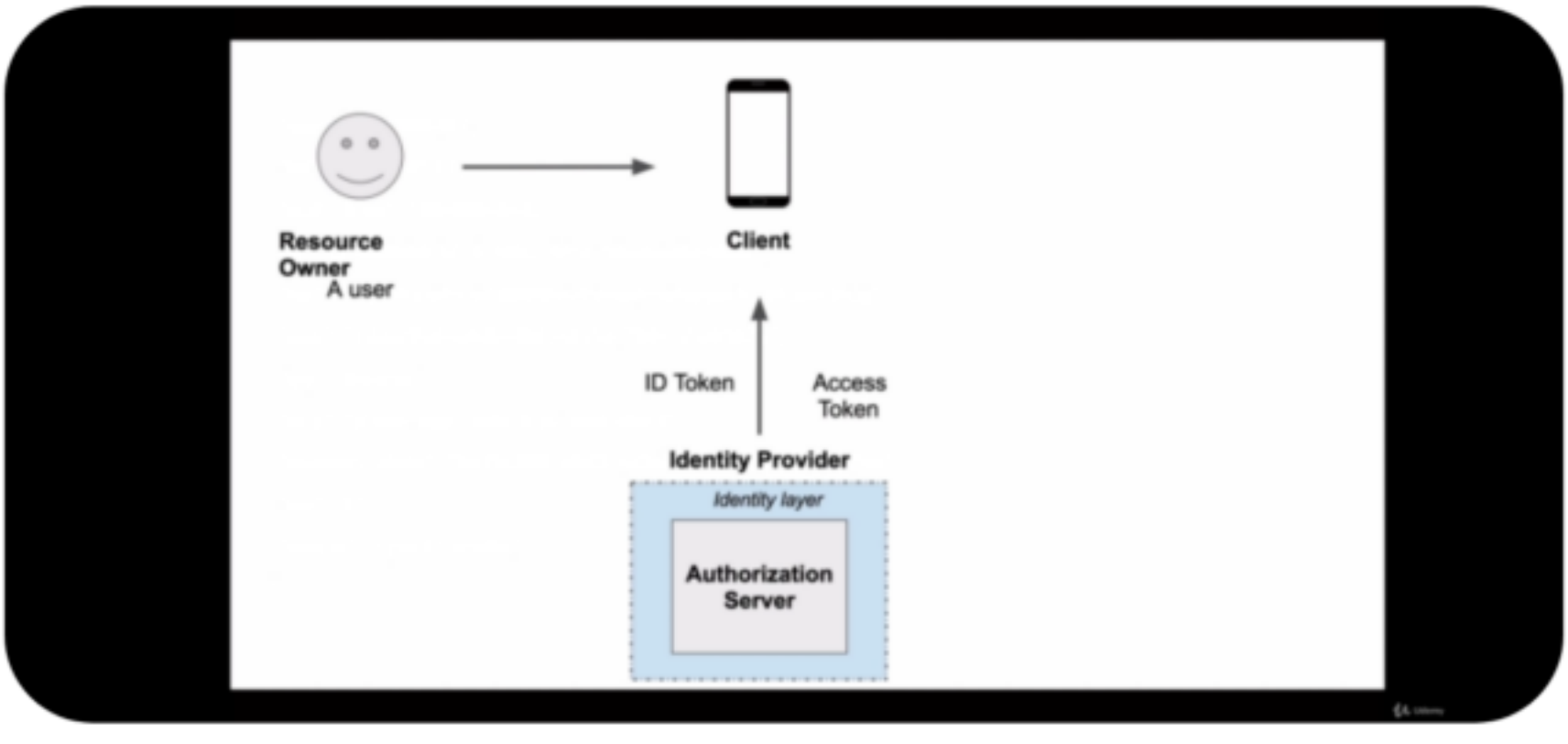


- Access tokens: required to access resource servers. Access tokens can be either by

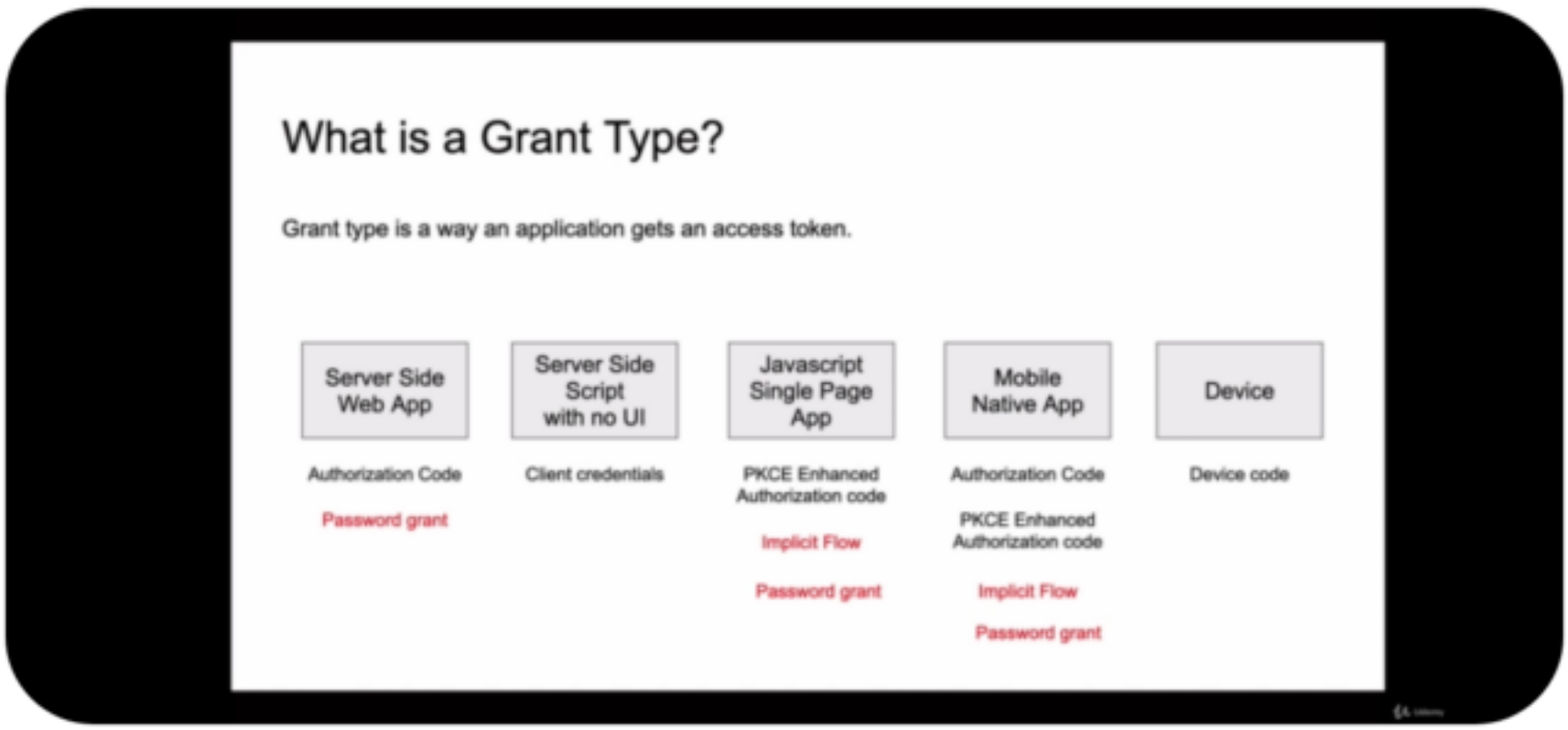
reference(contains reference id which refer to a specific token in authorization server) or by value(which actually contains entire authorized details in a jwt format)



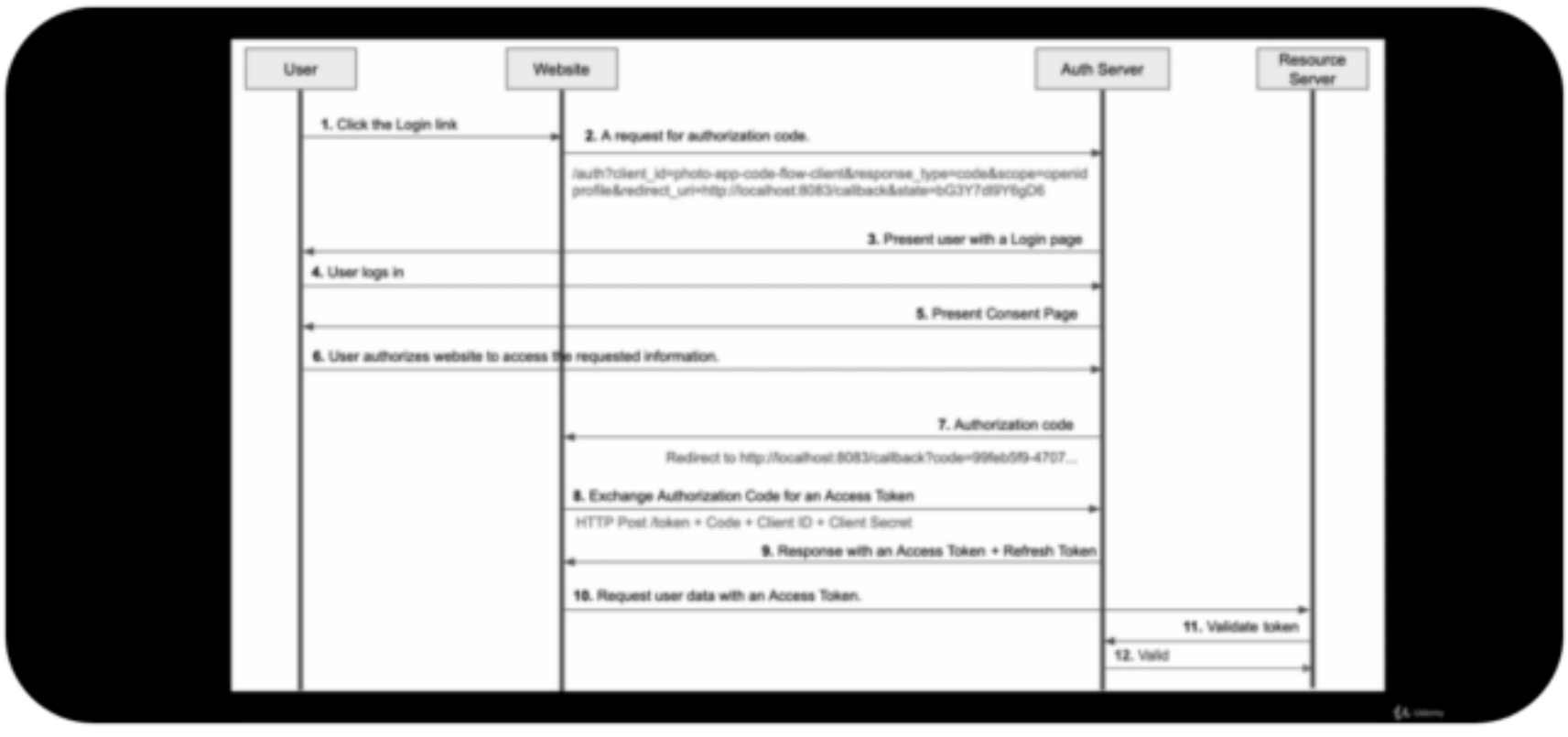
- Openid Connect: is an authentication protocol or identity management protocol.

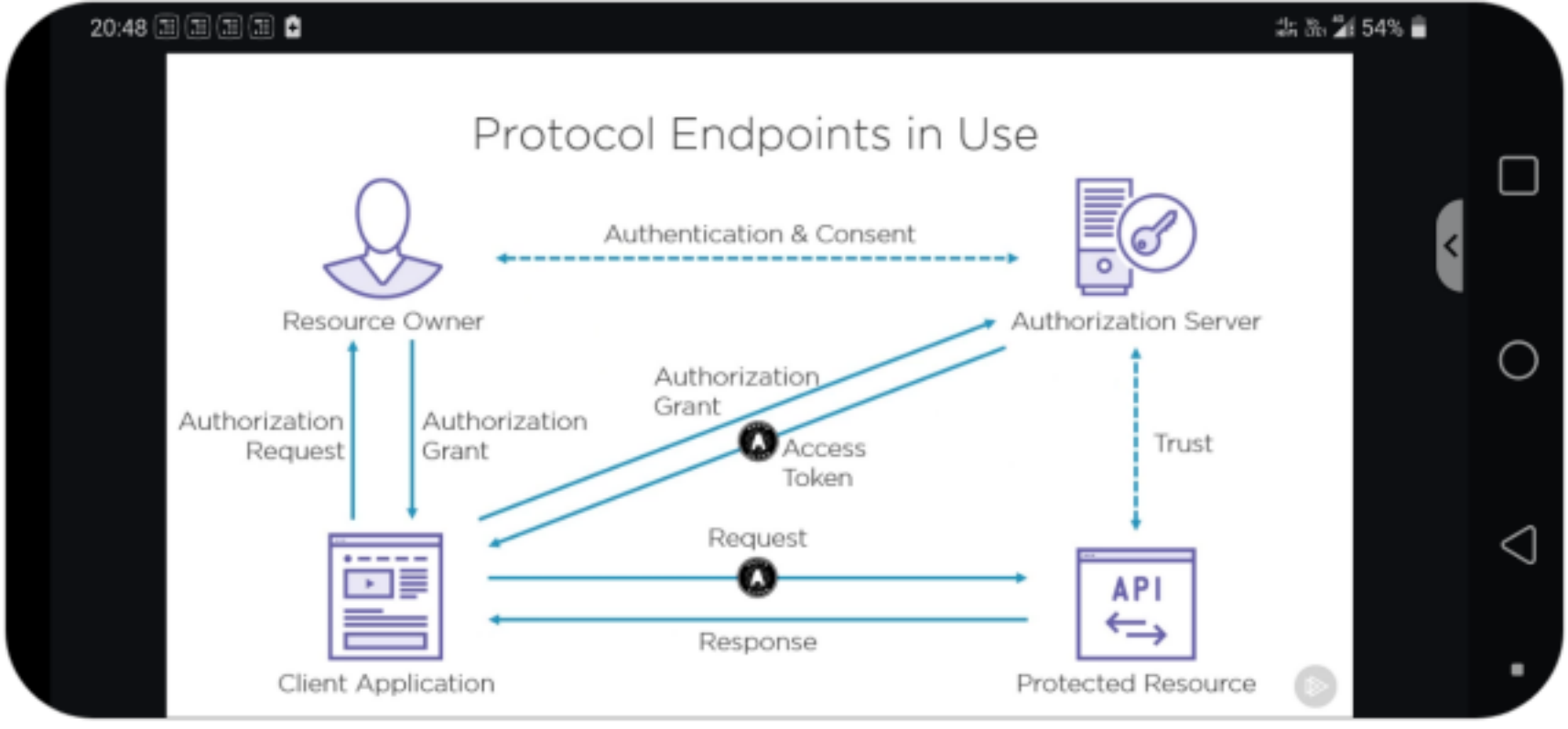


- Grant types: is a way an application or client gets an access token from authorization server.



- Authorization code grant types: this grant type ll be used in web apps which contact to backend server for data.

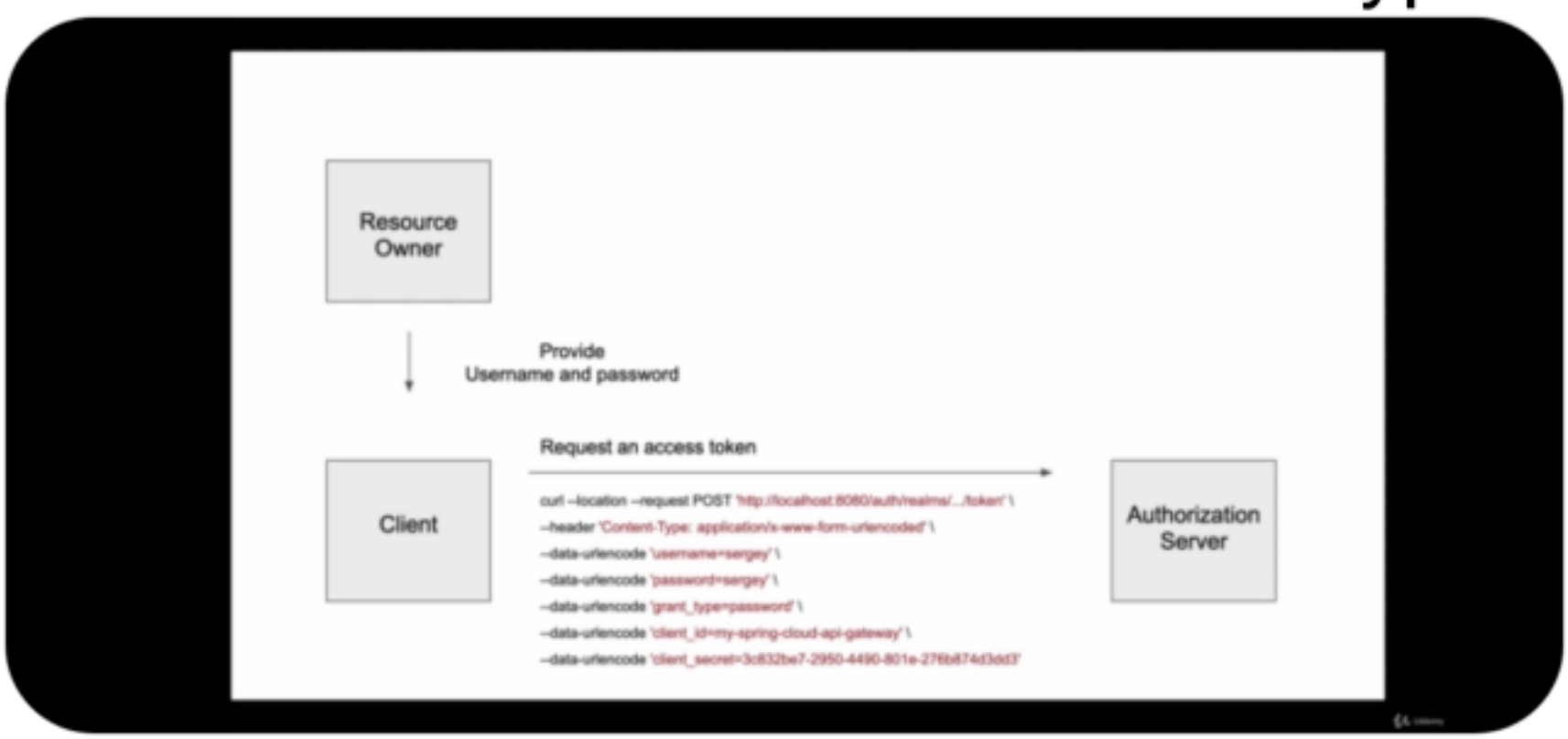




- We will hit /auth(get) endpoint by providing required query parameters which will give us the authorization code. We use this code and then hit the / token(post) endpoint by providing the code and other details to receive the actual access token and id token also if in case the authorization server supports the identity management or oidc
- Client credentials grant type: used in machine to machine communications between one microservice to another one. Microservices ll directly send

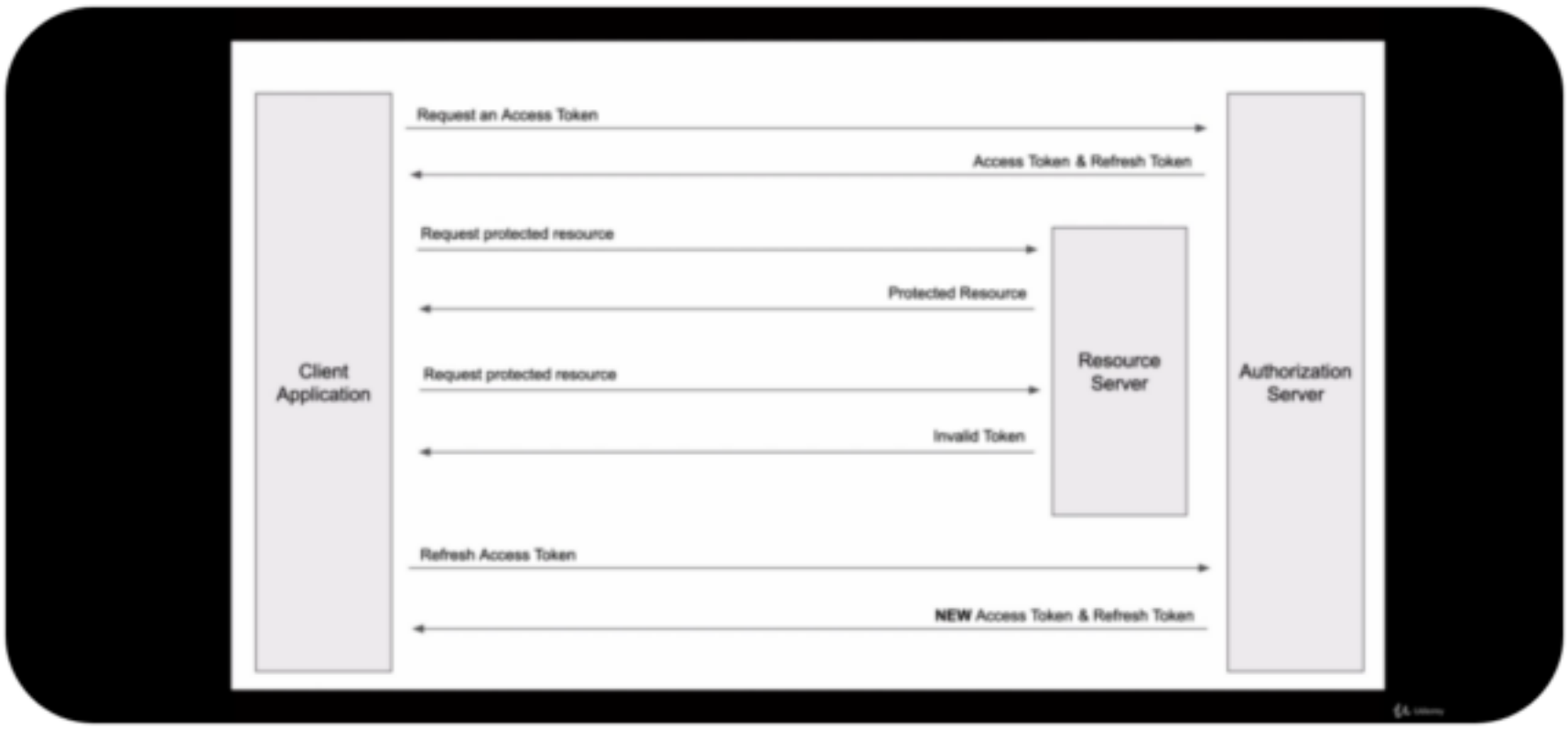
post request to /token endpoint with required query parms

≡ ○ Password grant types: if client apps does not support redirect, then this grant types can be used. Else it is better to use other types



≡ ○ Refreshing an access token or validity of an access token can be extended once its expired by

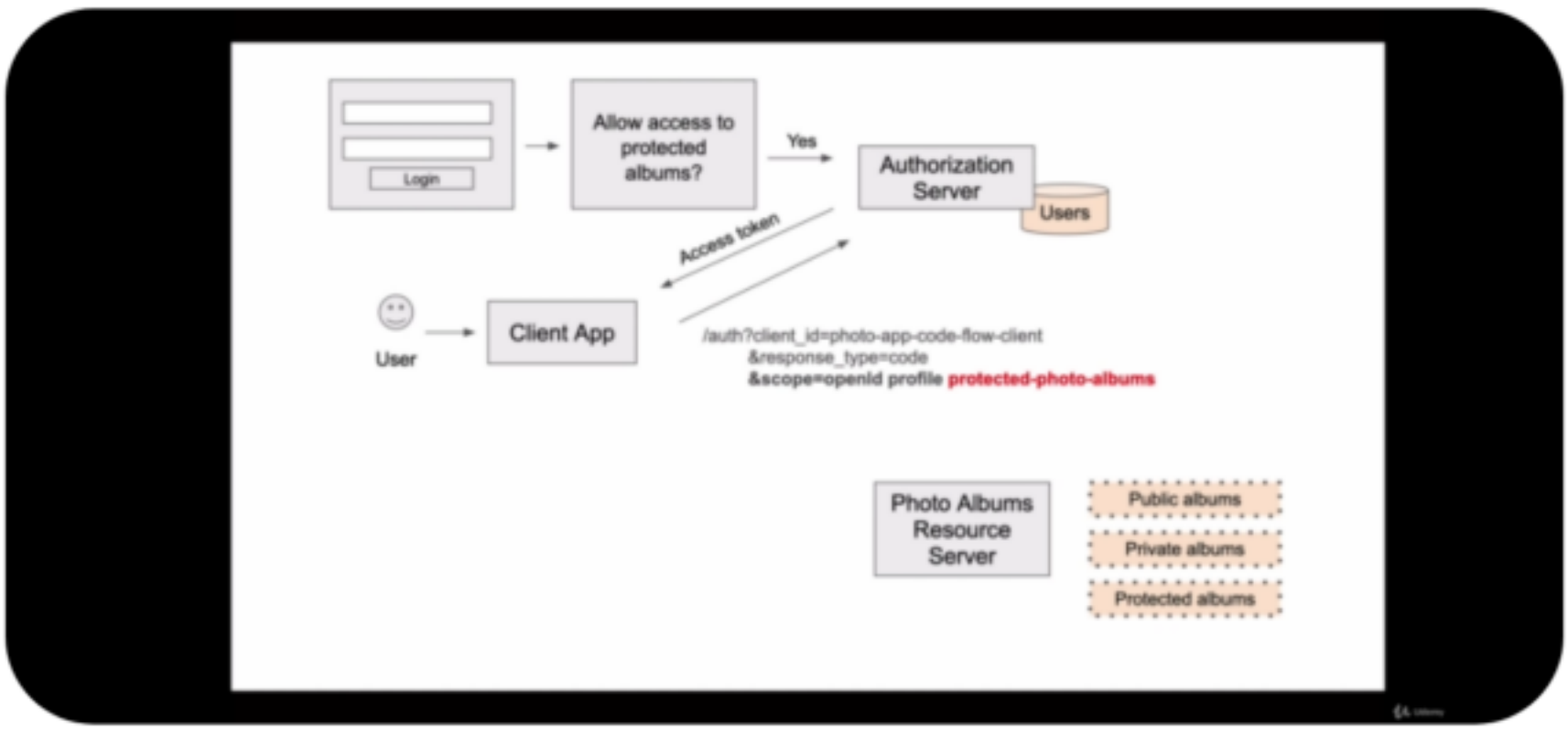
sending refresh token to authorization server.



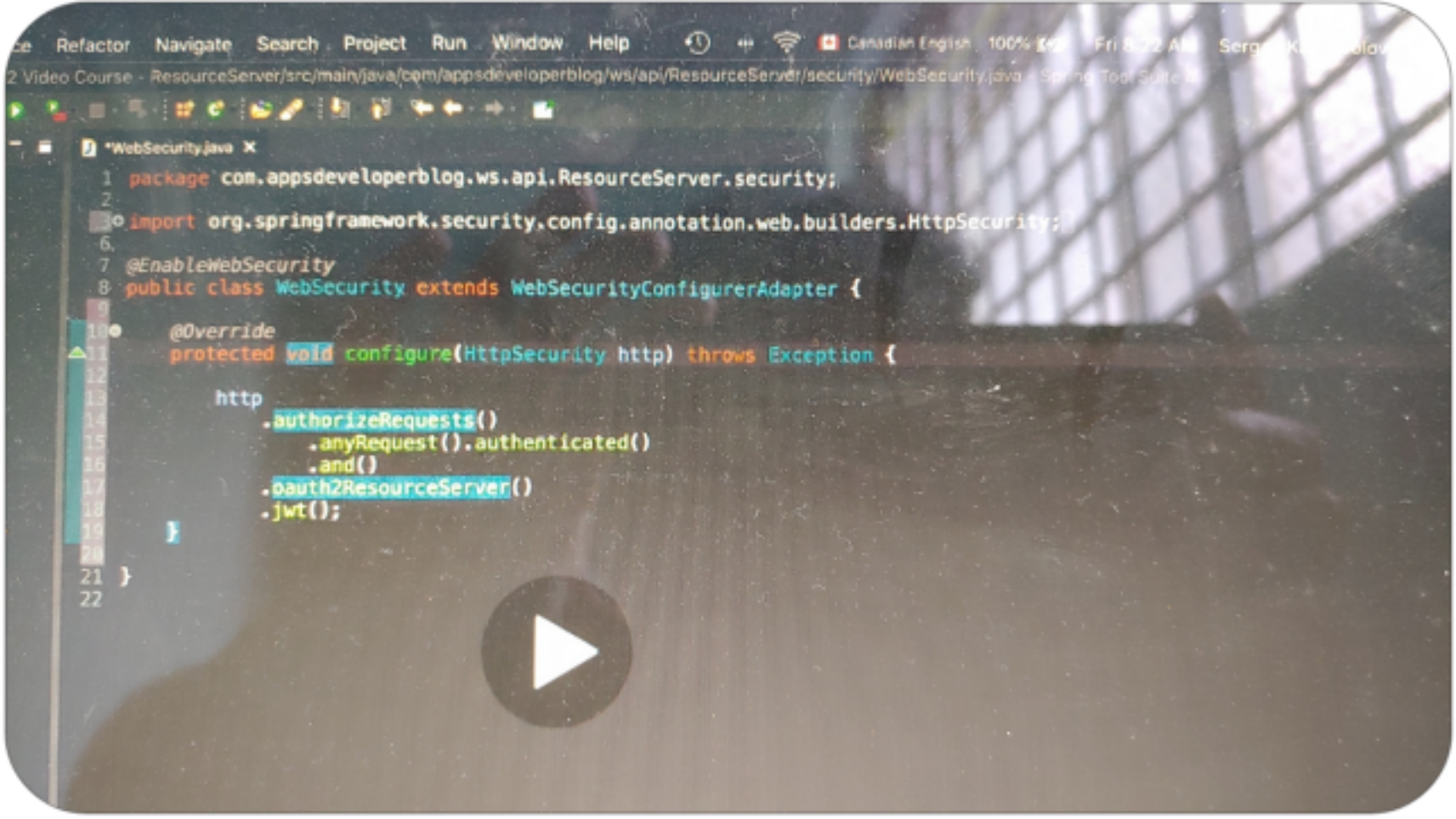
- In case of authorization server which supports oidc we can request for refresh tokens which never expires by using the scope offline_access.
- Scopes: is a mechanism in Oauth2 to limit an applications access to a users account. An application/client can request one or more scopes, this information then presented to the user in the consent form, and the access token issued to the application will be limited to the

scopes granted.

- Later client applications can access only those permitted resources as signed in the access token from a resource server. This is called scope based access control.



- To configure the scope based access control we need to configure the WebSecurityConfigurerAdapter.

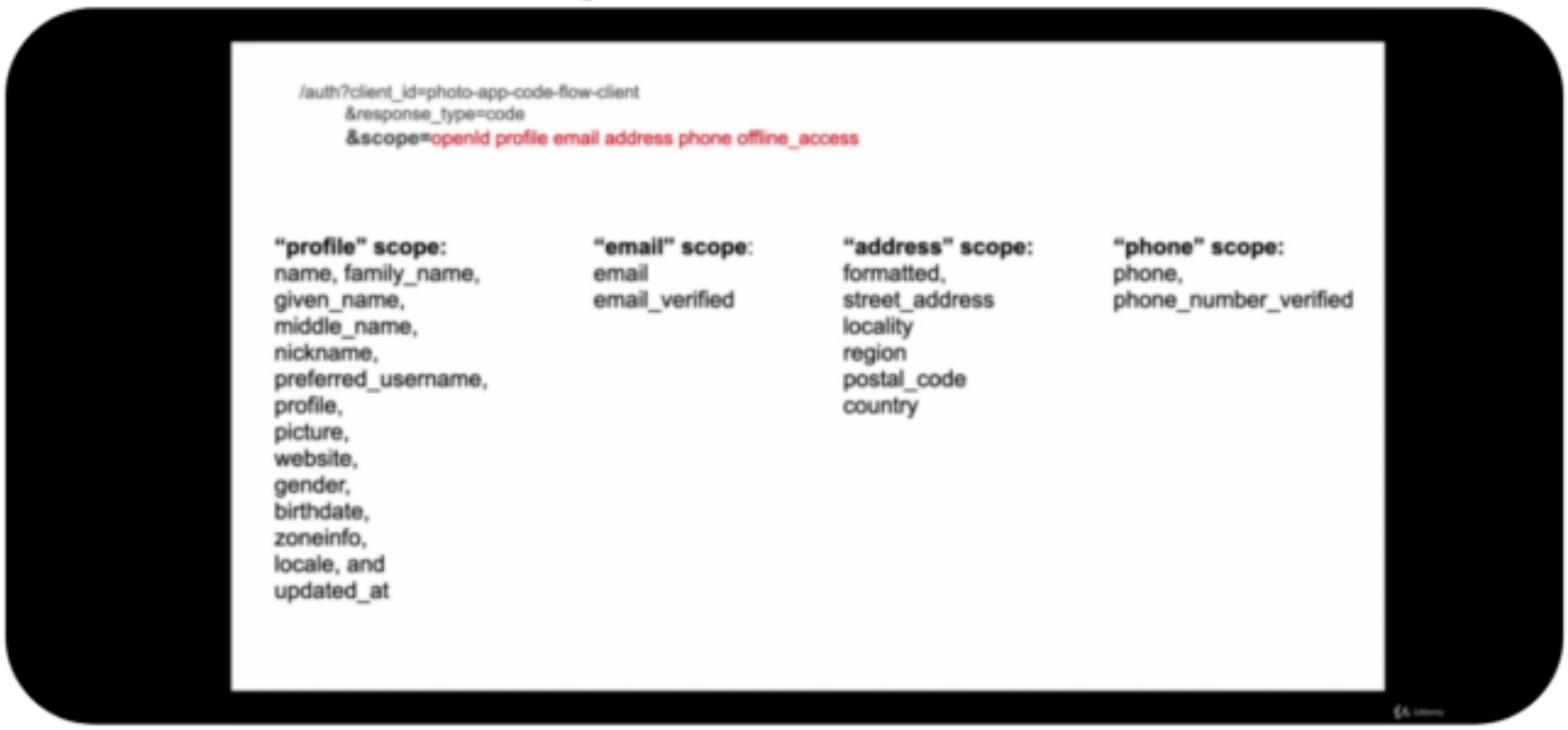


≡ ○ Here oauth2ResourceServer() will create the BearerTokenAutyenticationFilter to intercept the http request and it ll extract the authorization bearer token to expect the token to be jwt token and then pass it to jwt() to verify its validity.

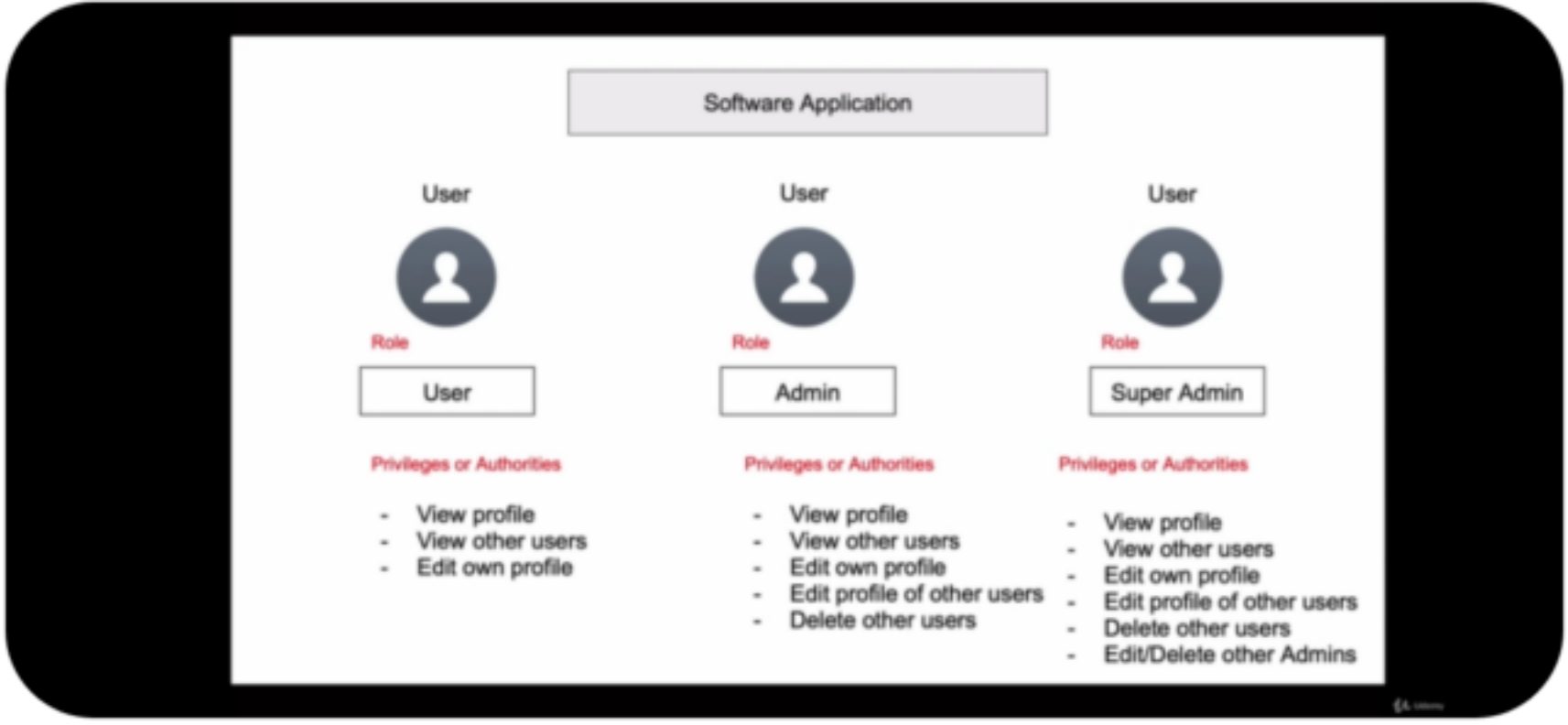
≡ ○ By default openid connect protocol supports many scopes which can be requested by clients. If the scope is openid then it tells the authorization server that, the client will be making an openid connect

request so authorization server must return an id token alonflg with access token.

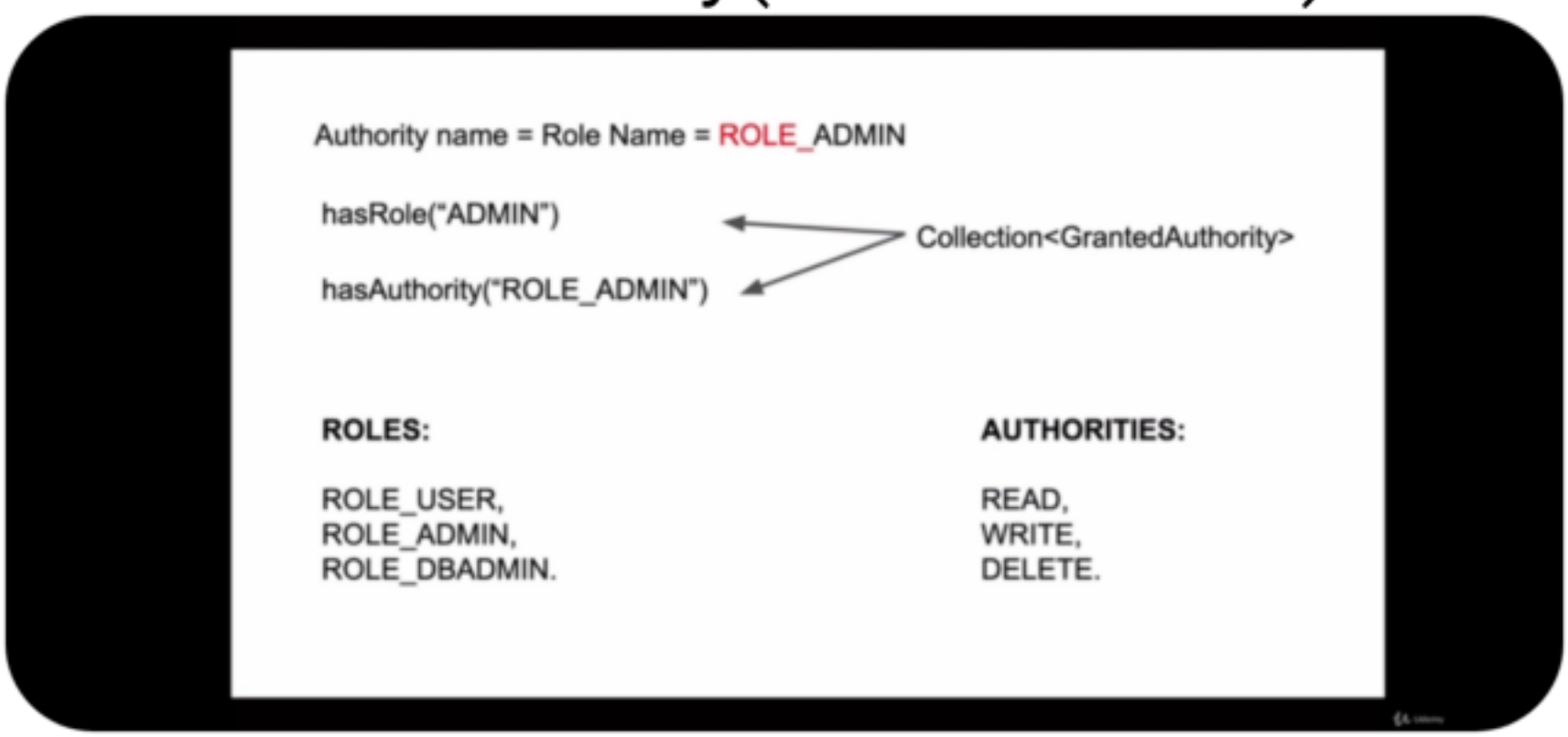
- Other scopes are like email profile address phone etc



- Roles and Authorities: role is collection of authorities. Authority is a privilege assigned to a user in the system like view profile, edit profile, delete profile

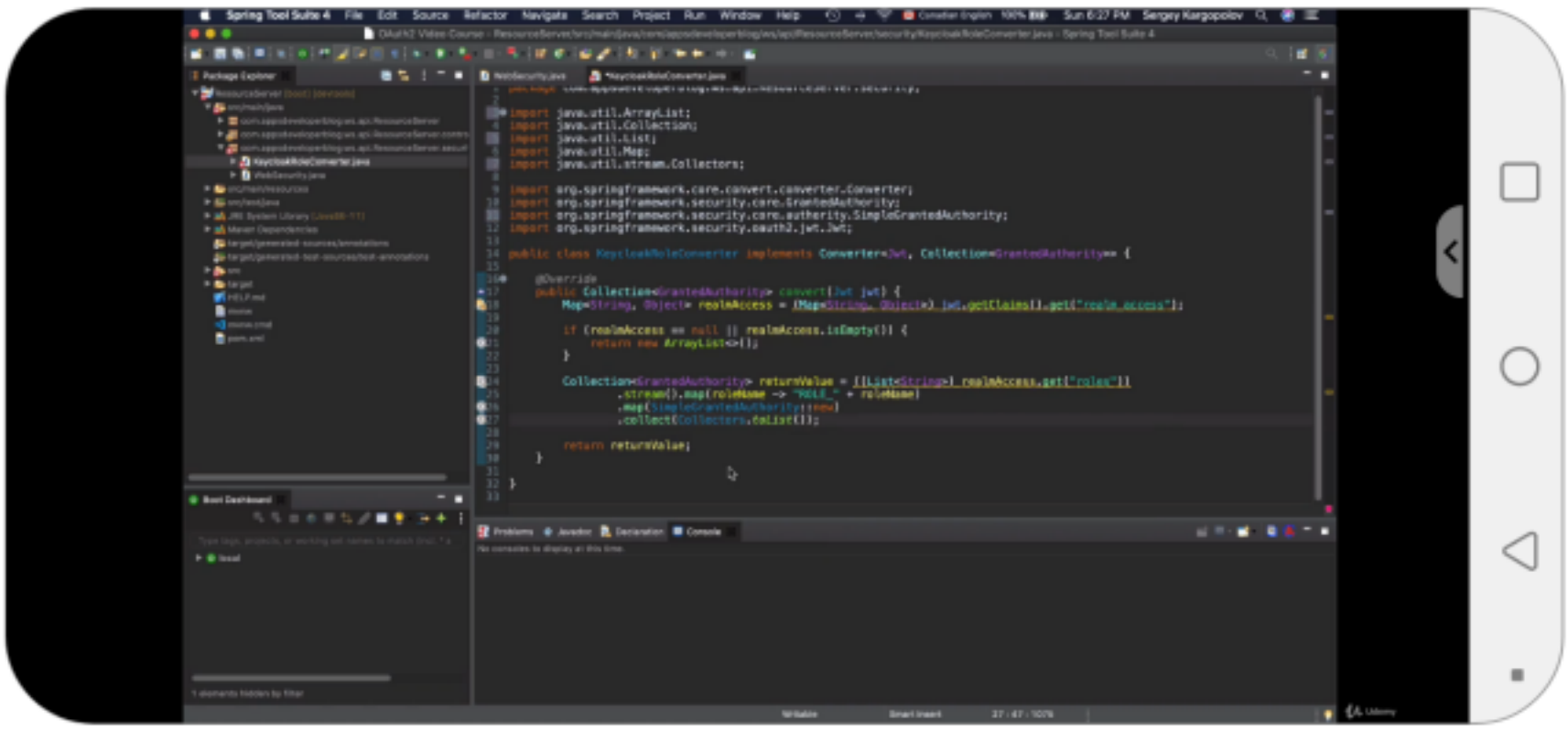


≡ ○ In spring security there is no difference between role and authority. Both ll be returned in the single list of Collection<GrantedAuthority>. Authority name = Role name = ROLE_ADMIN. hasRole(ADMIN). hasAuthority(ROLE_ADMIN).

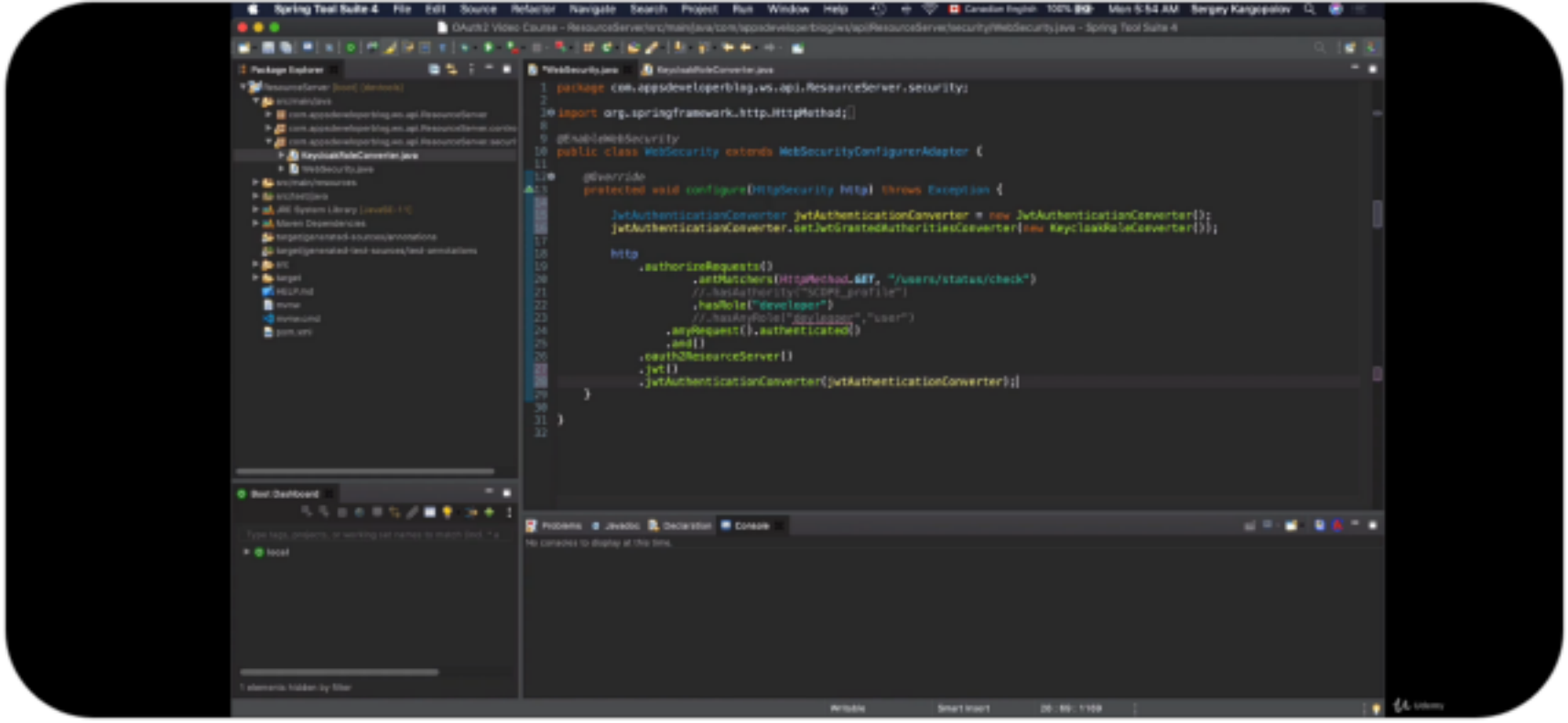


≡ ○ To have the role based access

control we need to add hasRole("developer") in the security configurer and then need to have a role converter class which can read the list of assigned roles(a specific claim) from access token and convert it to spring security list of granted authorities.



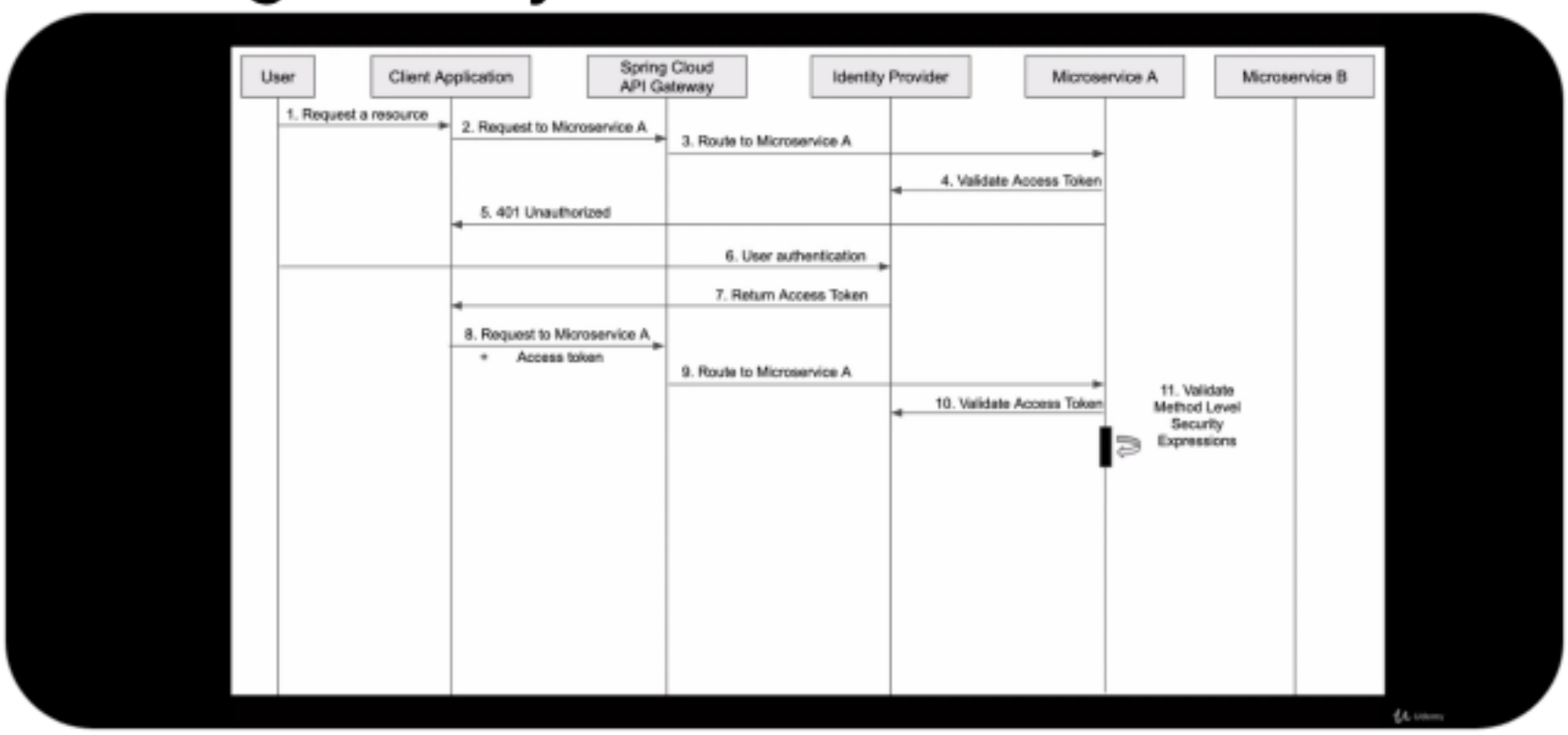
- Finally register this new class in the security configurer

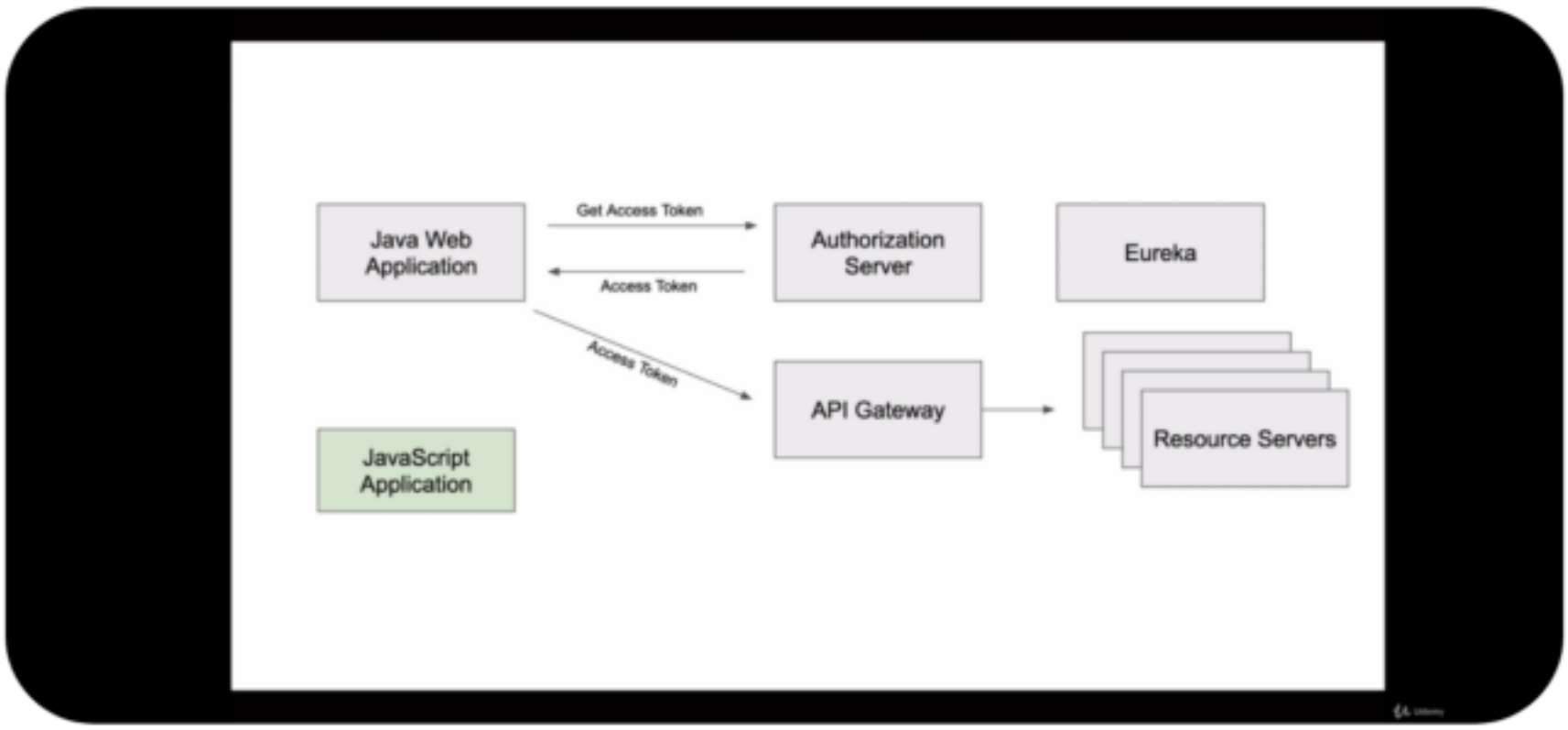


- Method level access security: spring security provides various annotations like @Secured, @preAuthorize(), @postAuthorize() where it support expressions which can be evaluated. These annotations can be used in services controllers method or at class levels. If the expressions returns true then the method ll be allowed to execute for that user else it returns access denied error
- To use these annotations we must use the annotation @EnableGlobalMethodSecurity(s

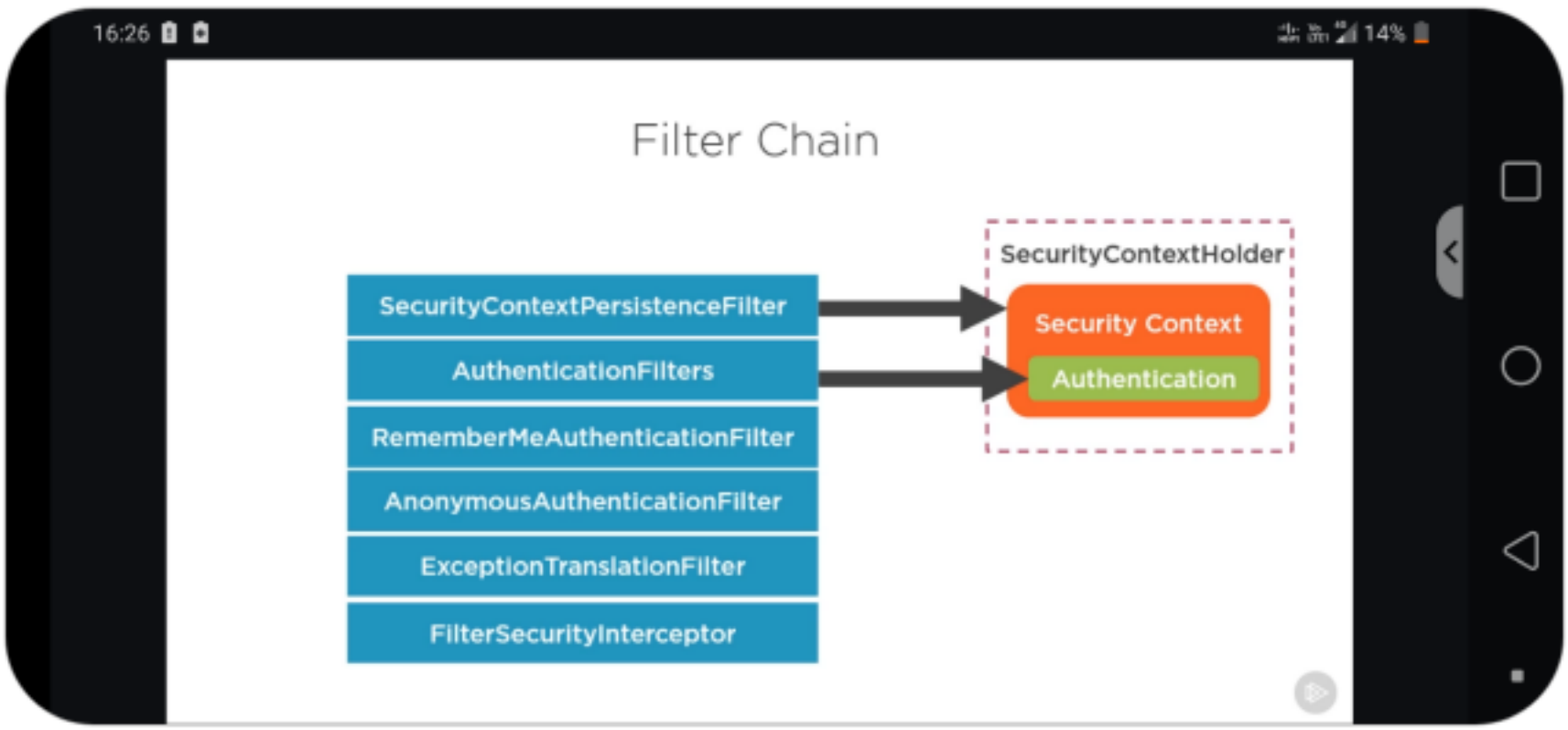
ecuredEnabled=true, prePostEnabled=true) in the WebSecurityConfigurereAdapter class

- Resource server behind api gateway:

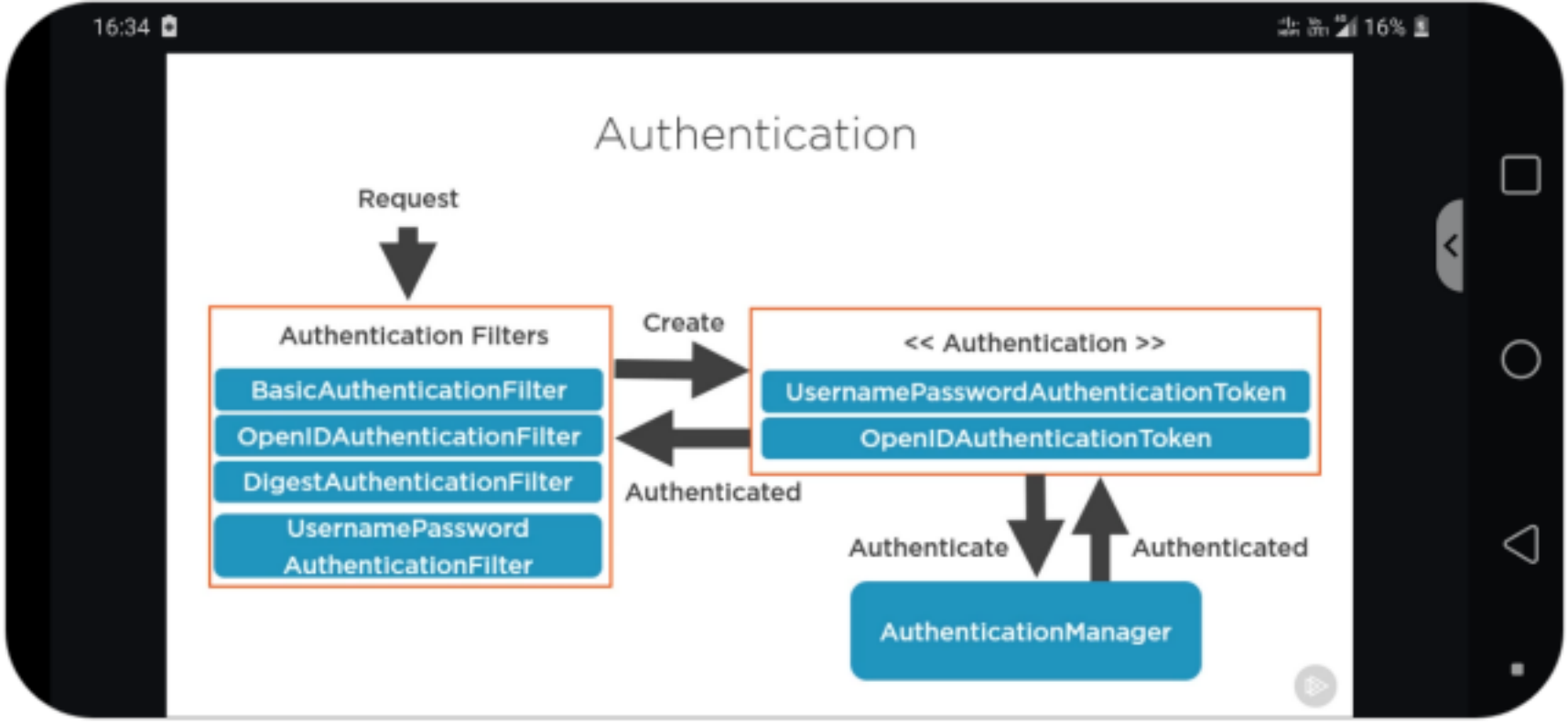




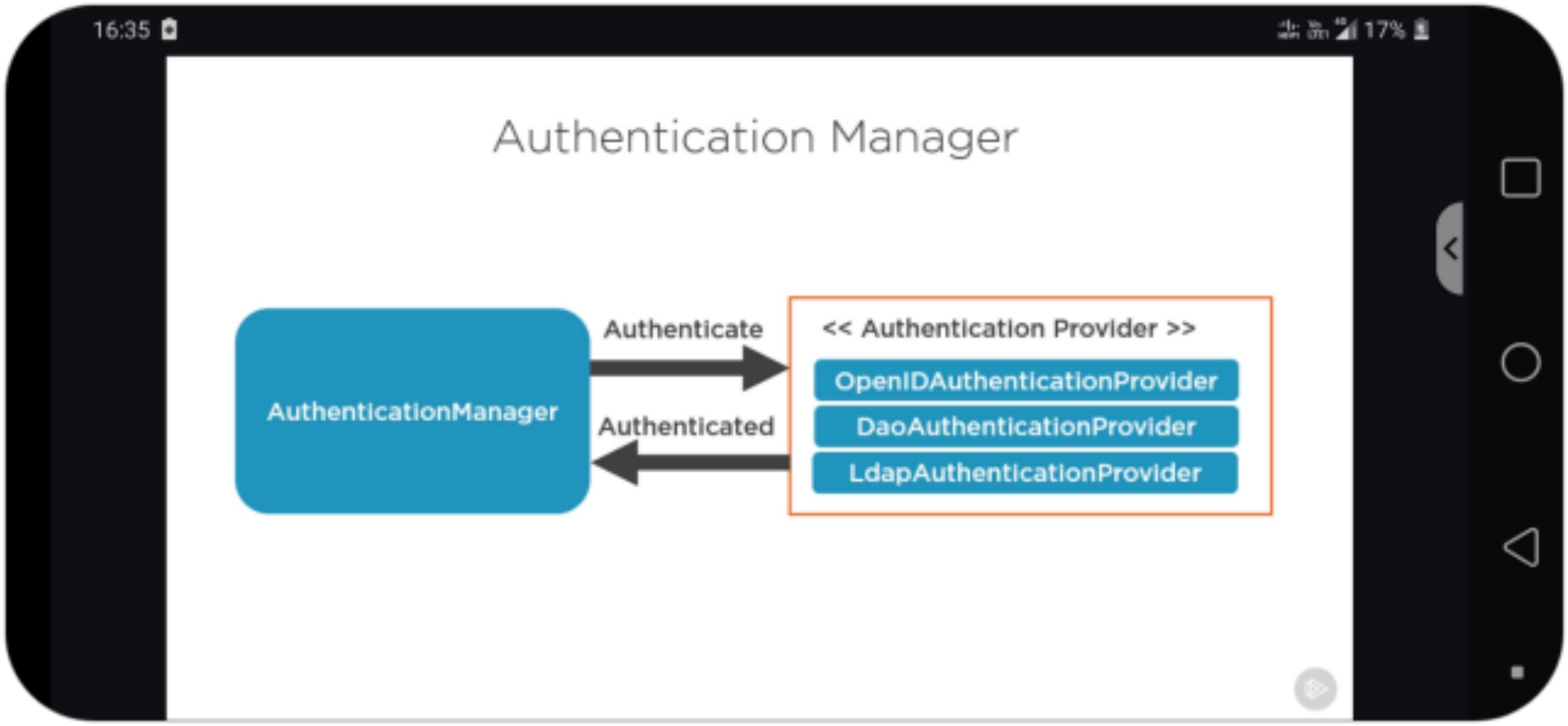
- Security filters in spring security: spring security has chains of filter which does the authentication and authorization. At root level it has security context which holds Authentication principle object. FilterSevirityInterceptor does the authorization work.



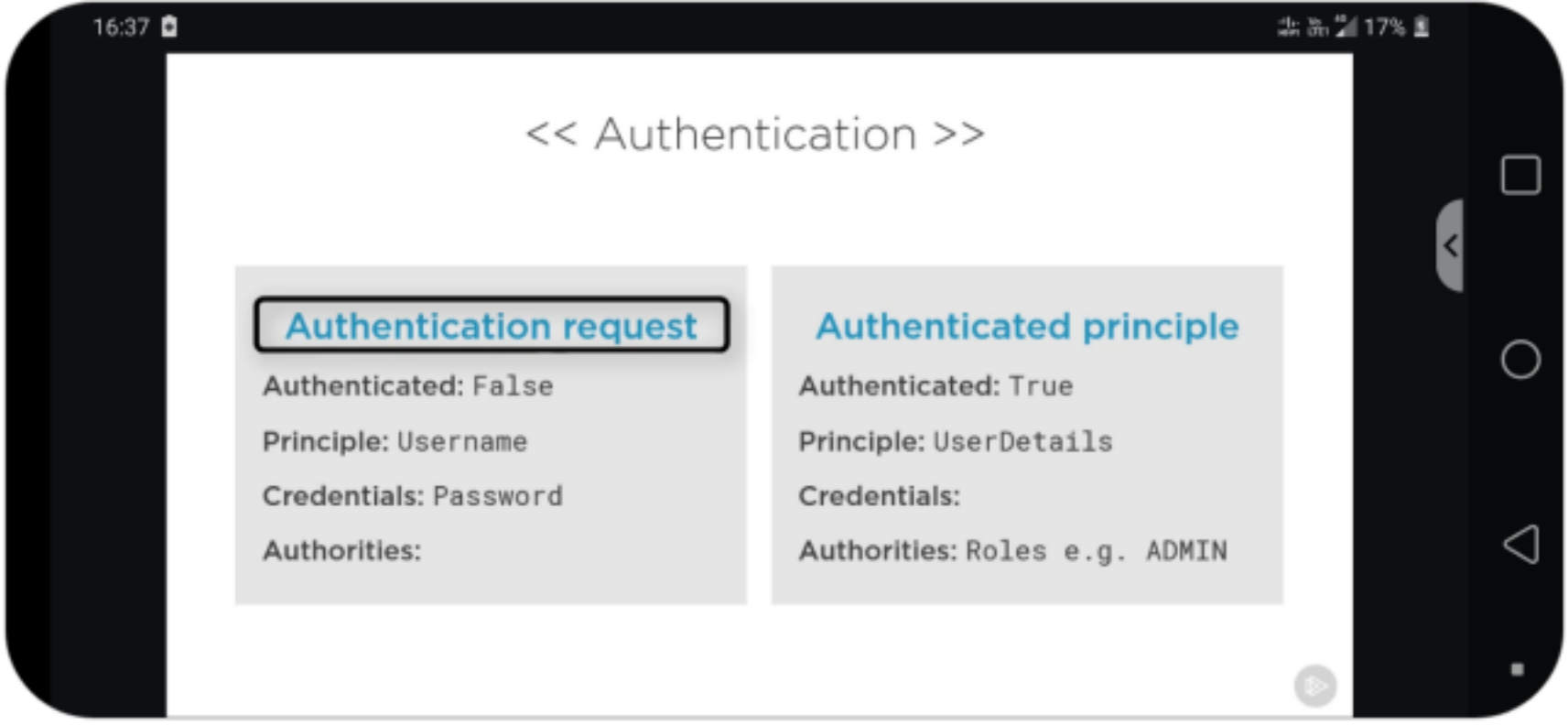
- Authentication Filter: which handles the authentication process.



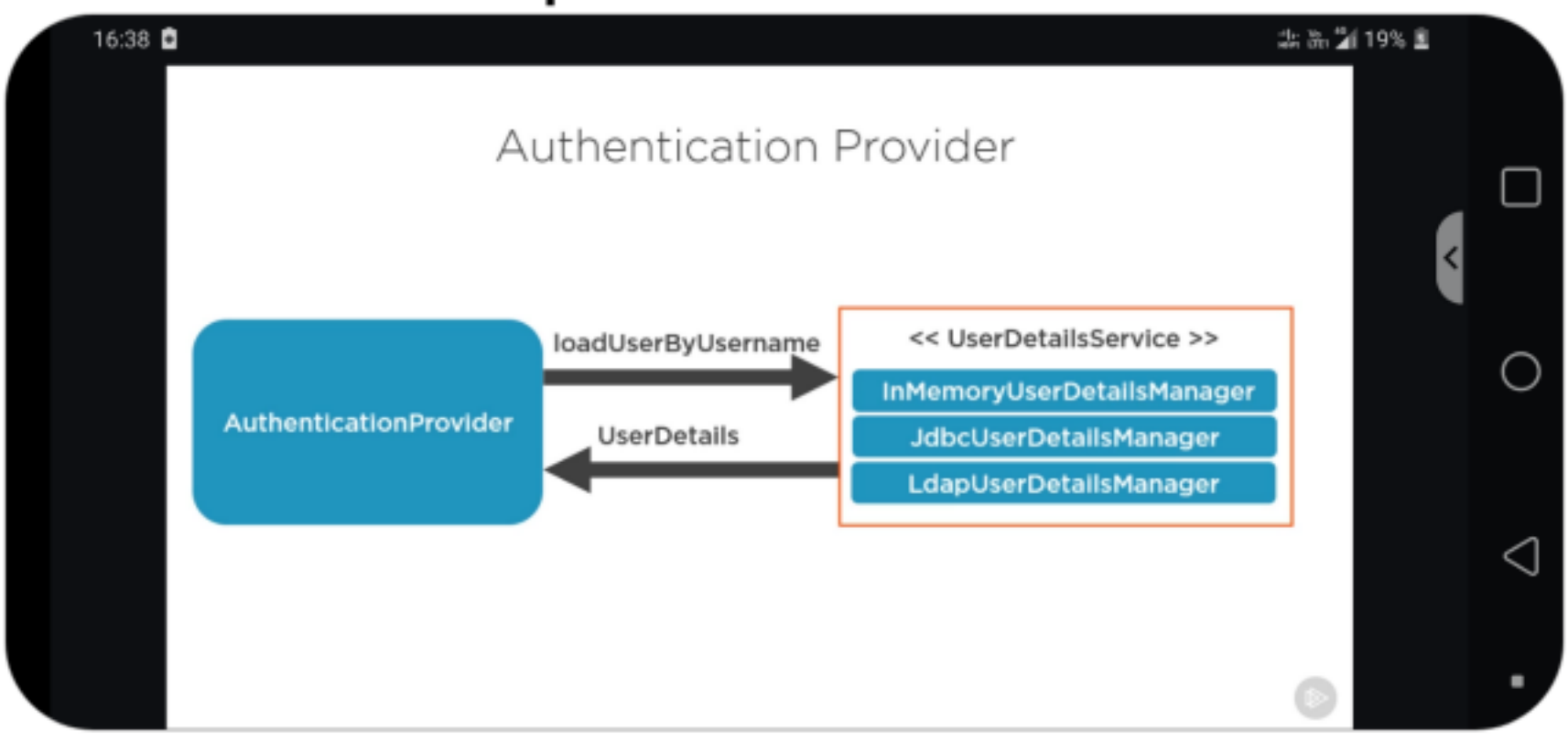
It creates Authentication Token and pass it to authentication manager which then pass it to Authentication provider



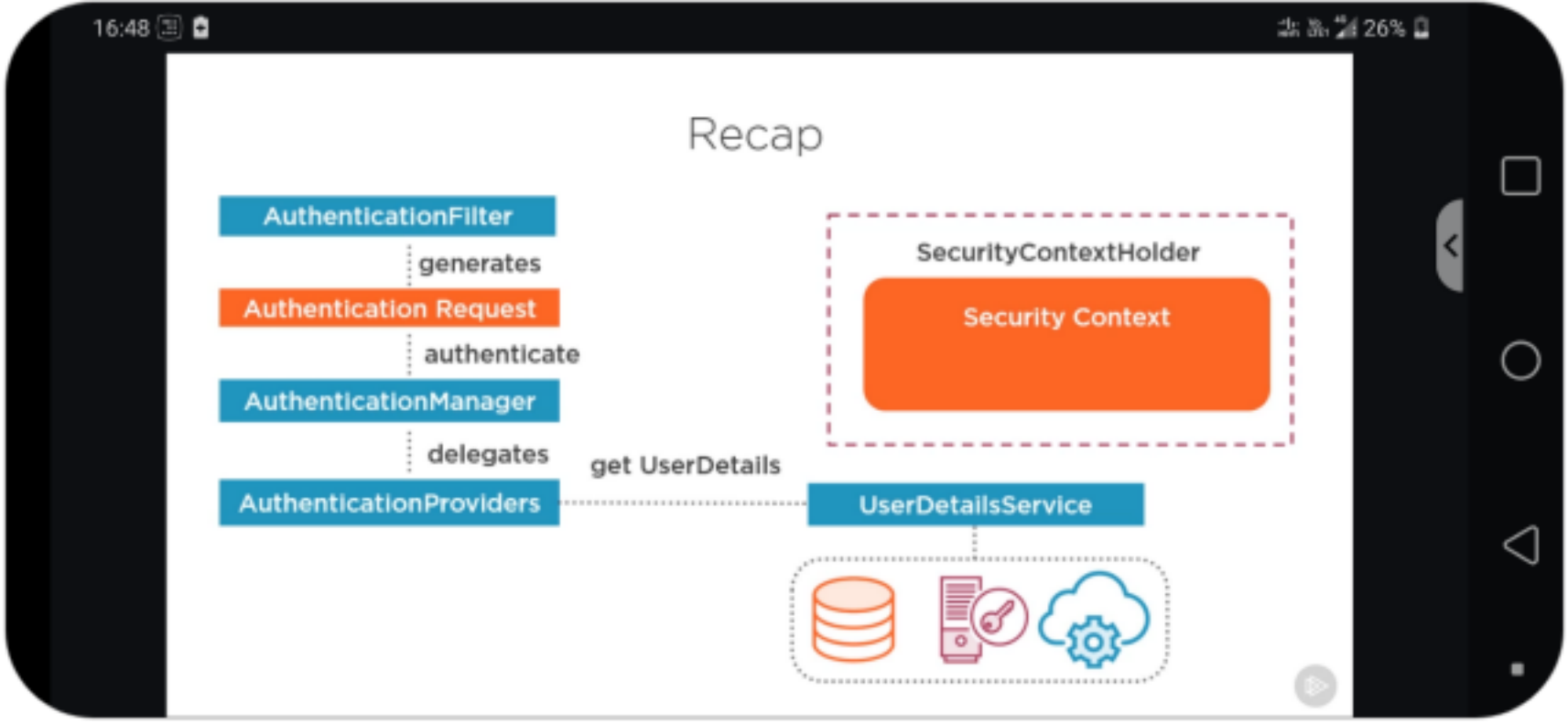
Content of authentication before and after the request

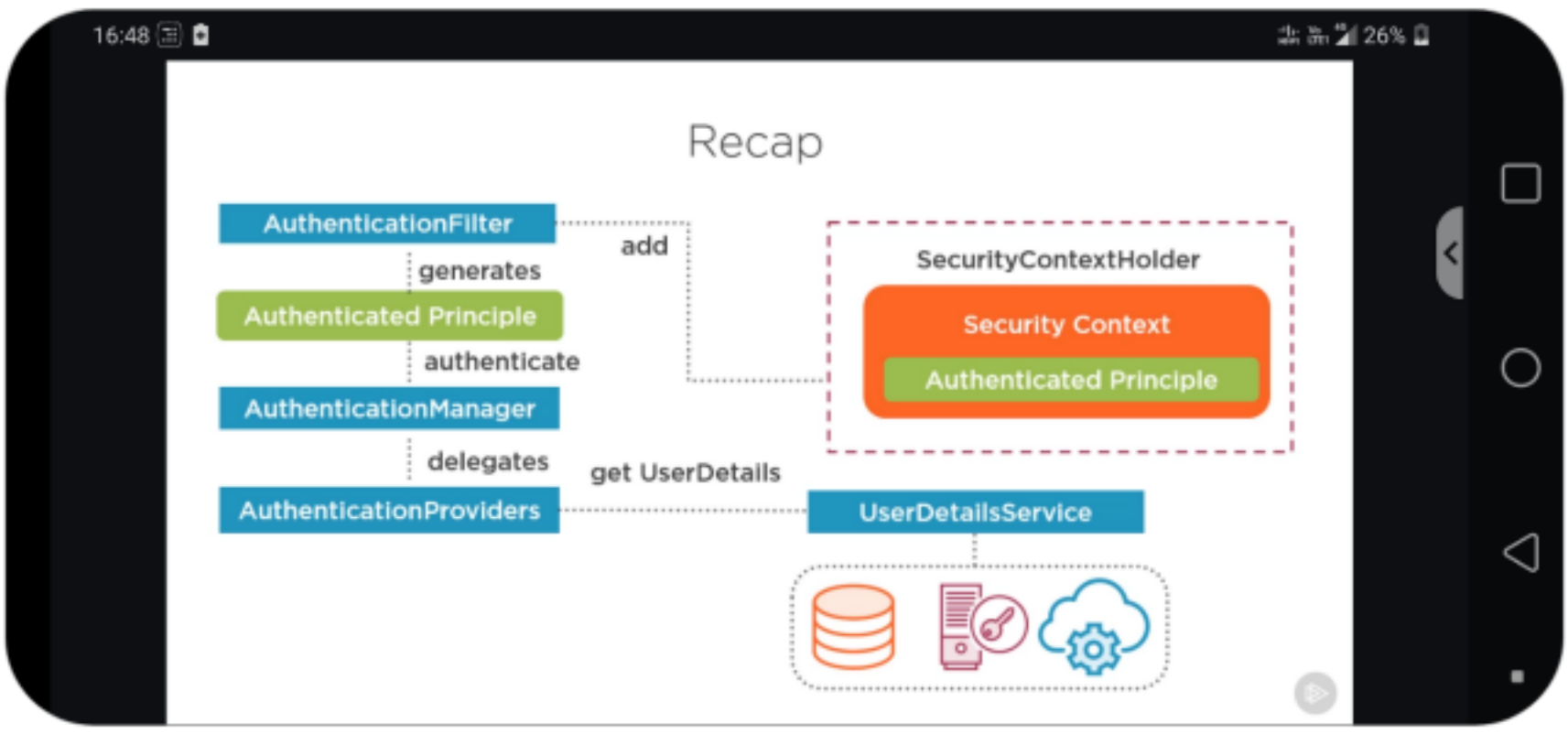


Authentication provider loads the user details from specific userdetailsservice



Recap





Spring Security login flow By Username and Password:

- AuthenticationFilter: For example we can create a class which extends usernamepasswordauthenticationfilter and overrides the attemptAuthentication method to

convert the incoming username and password as UsernamePasswordAuthenticationToken and pass it to AuthenticationManager. Later we need to register this class with HttpSecurity in the WebSecurityConfigurereAdapter with the addFilter method.

- ≡ ○ Then in the WebSecurityConfigurerAdapter, pass the custom UserDetailsService and BcryptPasswordEncoder to AuthenticationManagerBuilder
- ≡ ○ In the custom UserDetailsService implement the loadUserByUsername which returns the entire UserDetails object including the encrypted password by using spring data repository
- ≡ ○ Create a new UserDetails by

using spring security User object along with list of user authorities

- Once the authentication is successful, in the successfullAuthentication method of AuthenticationFilter, we use the UserDetails object and create the JWT token and send it back to the client

Spring Security in API Gateway:

- We can add spring boot starter security and jwt dependencies in API gateway and then implement the WebSecurityConfigurerAdapter which will allow certain url to be accesses without any token pr authentication and restrict access for other set of urls without an access token.
- It can also verify access tokens sent by a client before forwarding the request to next server by

using kind of BasicAuthenticationFilter

- ○ Creating a new class which extends BasicAuthenticationFilter and overriding the doFilterInternal method where we can have logic to check whether incoming request has the Authorization Header or not. If it has by using JWT plugin we can check whether the token has proper userid/subject and other required data or not. If it passes the verification we can forward the request else we can return the filter with validation error.

Method level security in a microservice:

- ○ AuthorizationFilter: Move the BasicAuthenticationFilter class which is implemented in API gateway into a particular microservice. So when user sent any http request the filter ll be

called which will check and verify the jwt token and then extract the user id, role and authority details into Authentication object of Security Context, so that this Autyentication object can be used from any method of any class.

Overall flows:

- Registration flow: create a rest api which will accept user details and save it in the DB. While saving the password it will encrypt it by using Bcrypt password. Along with this spring security provides support for forgot/reset/update password, password strength and rules.
- Email verification flow: once the user is registered, by default it will be disabled and an email verification link is send to user. Once the user click and activate

this link, user will be enabled in the DB

- Login flow: it can be either form based login or basic authentication login. In both the cases we can use in-memory username password for the authentication manager or we can configure authentication manager to connect with custom UserDetailsService which will connect to user and roles table of the db and creates a User object with user and authority details. Then spring security will automatically verify the encrypted password against the given password. Once it is successfully logged in, we can create a jwt token with user details and send it back to client.
- Logout and remember me flow

==========

Kafka Consumer in Spring Kafka

- ≡ ○ Consumer group id is required
- ≡ ○ Consumer auto commit by default false
- ≡ ○ Consumer auto.offset.reset by default latest so only latest message ll be consumed by consumer. If it is earlier then all message ll be consumer
- ≡ ○ ConsumerRecord is the spring representation of lafka message. This provides informations abt key partition or message itself
- ≡ ○ @KafkaListener has option to send concurrency value. If we set it as 2 or 3, spring creates those many consumers per that group to listen from.
- ≡ ○ Producer and consumer configurations can be override also kafkatemplate can also be configured.
- ≡ ○ Can filter any message per

consumer group through some kafka container factory bean configurations, and then the same container factory can be passed as a parameter to @KafkaListener containerFactory argument

≡ ○ KafkaListener and KafkaHandler to handle different messages from the same topic

≡ ○ KafkaListener with options to set group id, how many consumers per group and exception handler bean name which usually implements ConsumerAwareListenerErrorHandler interface where we can log the exception and send to elk.

≡ ○ Configurations can be added in yml file for consumer to read the message from begining or from latest whenver it restarts and also auto commit of offset to true or

false.

≡ ○ If we add new partitions in a topic while consumers are running kafka does rebalancing and assigns tye new partition to some consumer. By defaukt it is for 5 min which can be configured

≡ ○ We can use @SendTo annotation to directly send method return value to other topic

≡ ○ One partition should always consumed by one consumer. But one consumer can consume message from many partitions

≡ ○ We can also write global error handler instead of writing error handler for each listener. Write a class which implements ConsumerAwareErrorHandler and overrides the handle method to log the error or send it to elk. Then register this class to kafka container factory

KafkaListenerContainerFactory bean.

- Dead letter topic: Also all the failed messages can be send to kafka´s dead letter topic available for each topic with partition count same as its original topic. Then another listener can consume these unprocessed messages and do some functionalities on them

Spring Retry

- Use Spring RetryTemplate to retry message processing in consumer for the configured number of times before sending the actual error and then handling it at global error handler