README – PUSH NOTIFICATION FCM

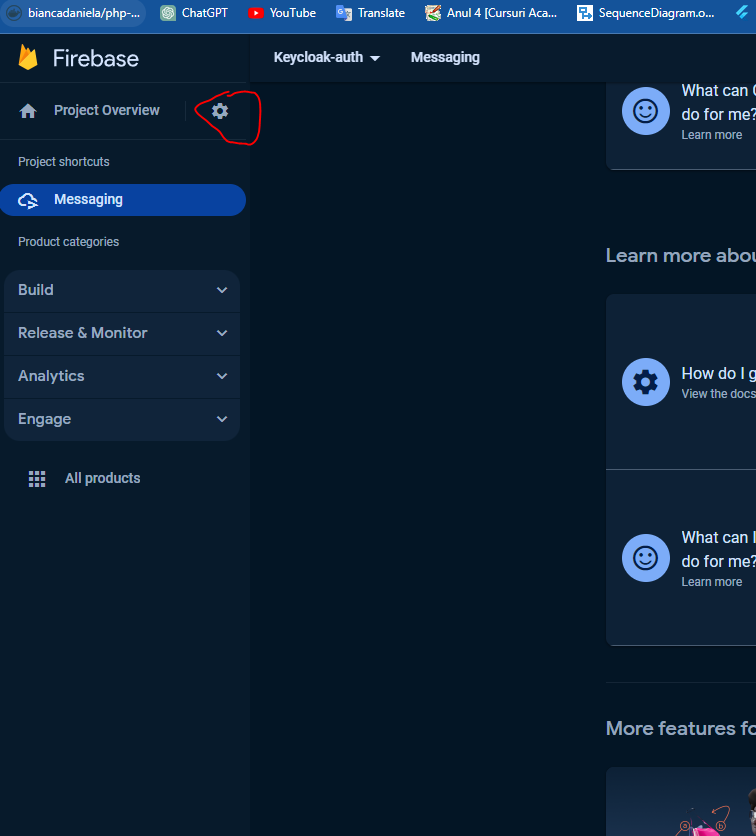
1. Pentru a implementa un SPI avem nevoie sa implementam interfetele: **ProviderFactory** si **Provider.** Si mai trebuie sa implementan un service configuration file (vezi exemplu cu theme selector SPI)
2. Descoperirea extensiilor la momentul pornirii serverului: In Java, ‘java.util.ServiceLoader’ este un mechanism standard folosit pentru a incarca servicii sau extensii in mod dynamic. Pentru ca, Keycloak sa poata descoperi si incarca automat o extensie personalizata trebuie furnizat un indiciu unde se poate gasi aceasta implementare. Acest lucru se face prin crearea unui fisier special in directorul ‘META-INF/services’ al arhivei JAR a extensiei.

**Numele Fișierului**: Fișierul trebuie să fie numit după interfața serviciului pe care îl oferi. De exemplu, dacă implementezi EventListenerProvider, numele fișierului ar trebui să fie org.keycloak.events.EventListenerProvider.

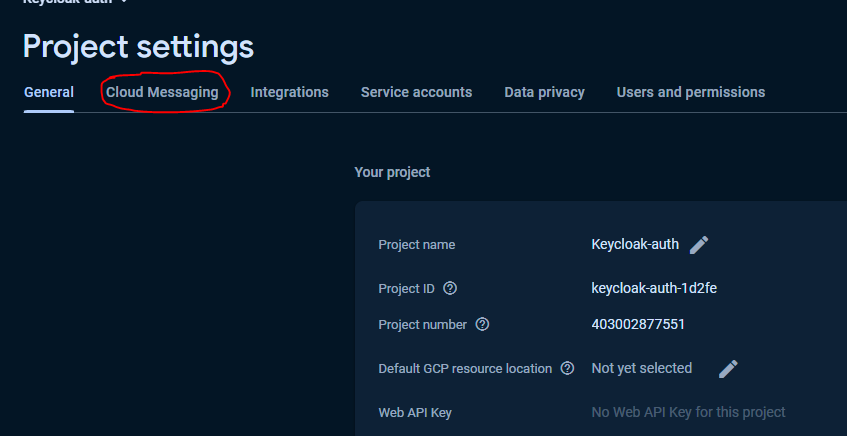
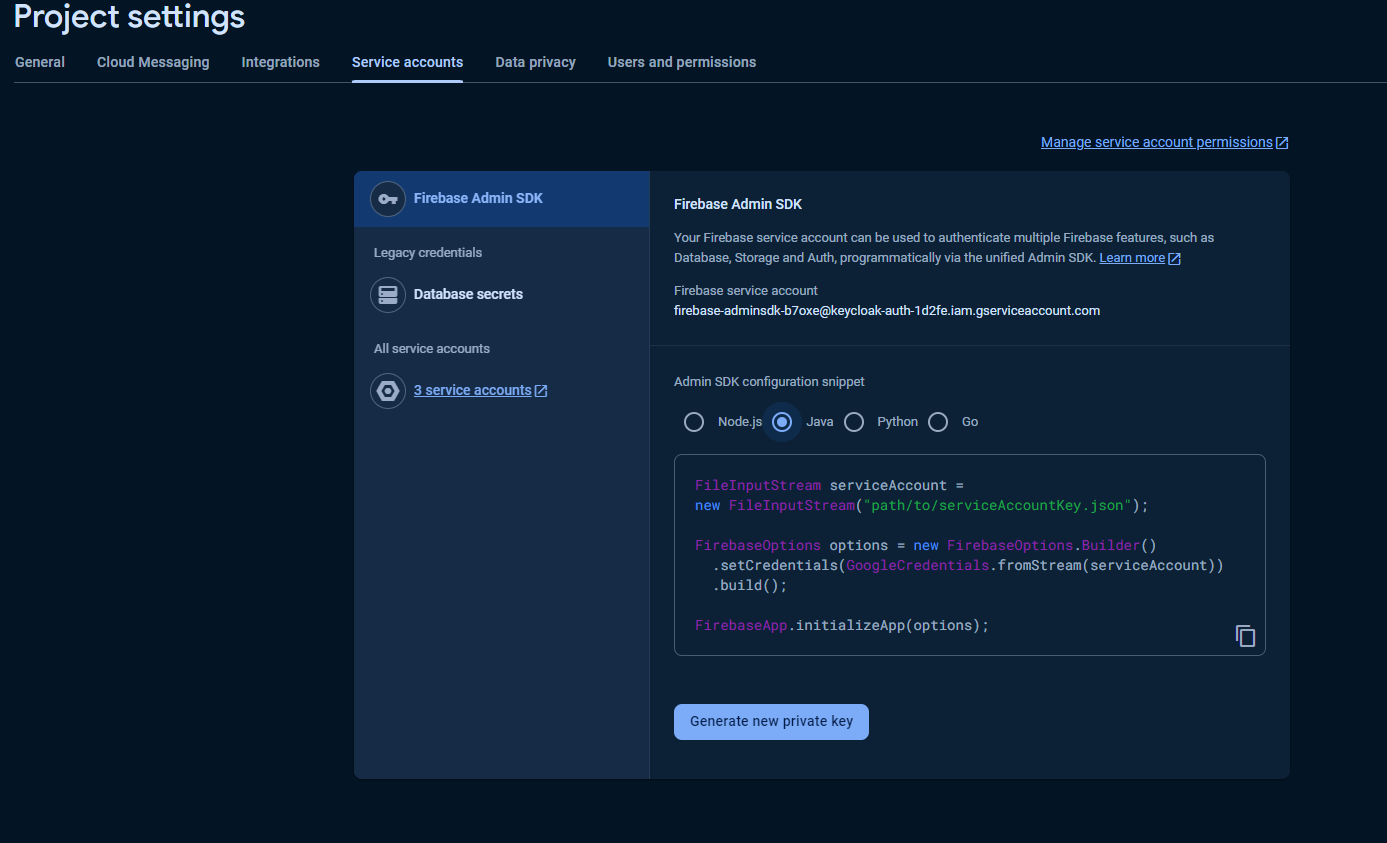
**Conținutul Fișierului:** În acest fișier, pui numele complet (fully qualified name) al clasei de implementare. De exemplu, dacă clasa ta se numește com.example.MyEventListener, aceasta este informația pe care o vei scrie în fișier.

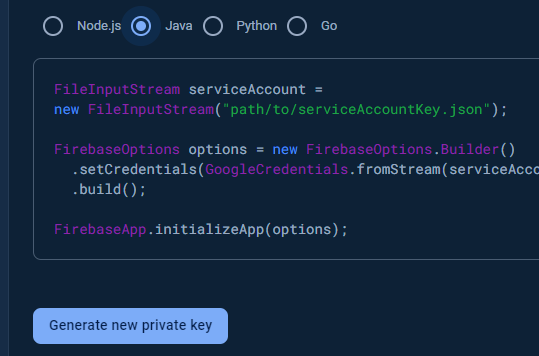
1. Pentru a adauga extensia in keycloak pot sa pun intr-un volum, volum unde se afla si directorul de extensii al Keycloak.
2. SPI (Service Provider Interface) => left column, providers in right column

Mai intai testez serviciul de fcm cu postman:

1. 

Si apasam Project Settings

1. 
2. 
3. Si generam new private key:

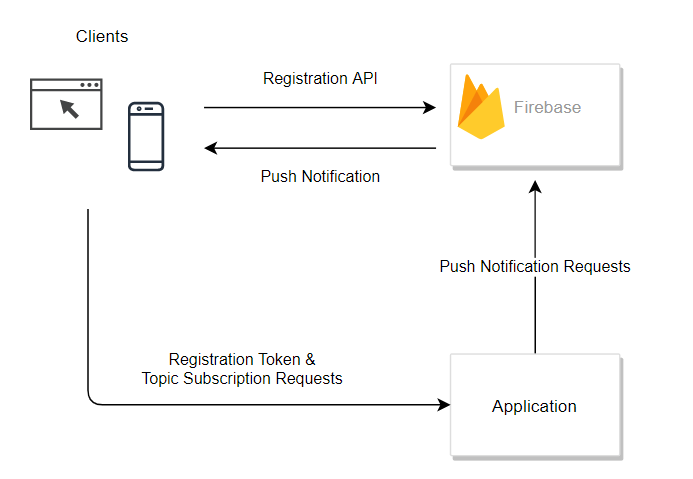


<https://firebase.google.com/docs/admin/setup>

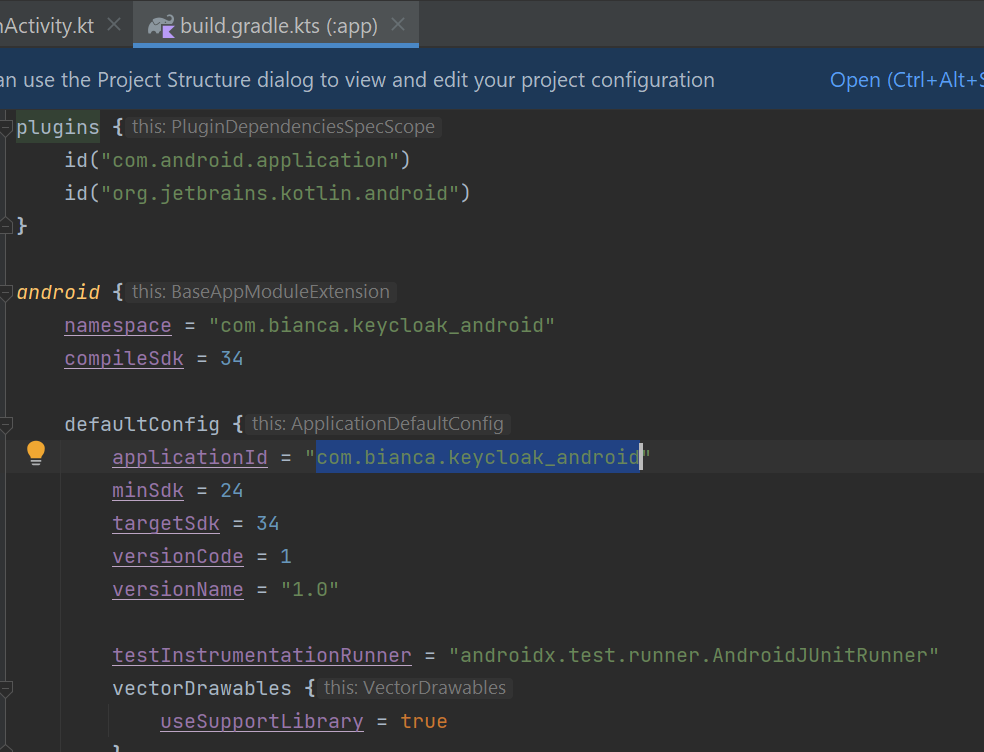
<https://firebase.google.com/docs/cloud-messaging/server>

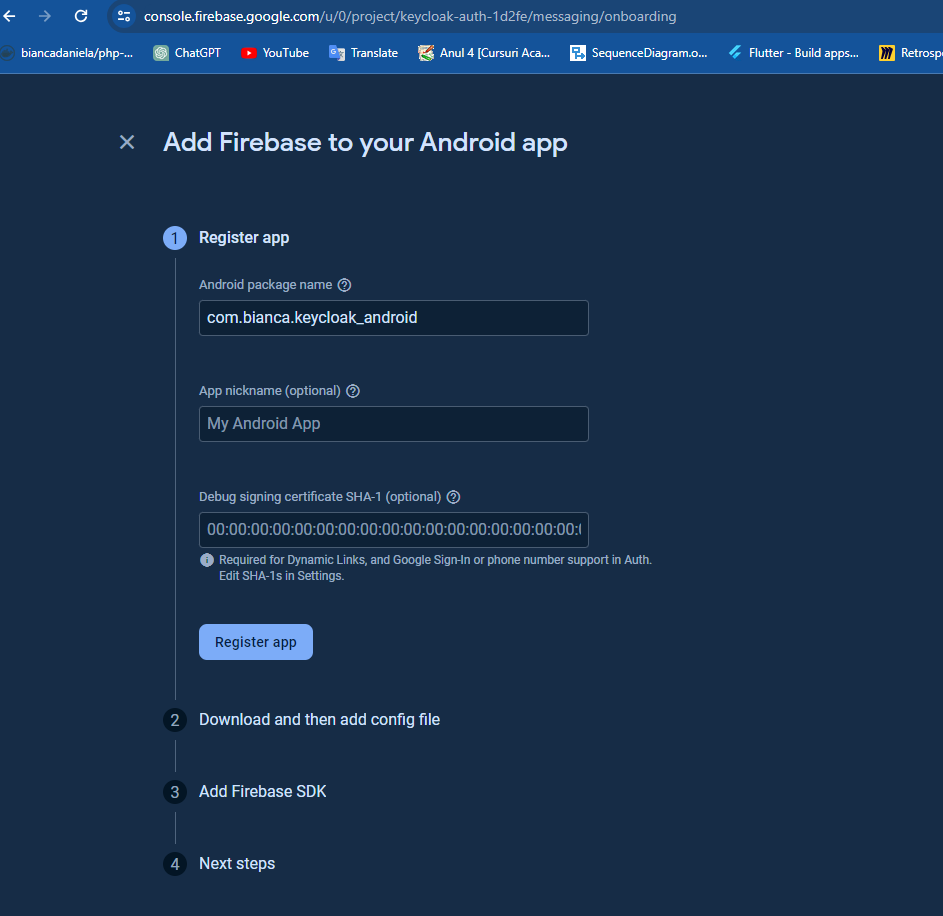
When authorizing via a service account, you have two choices for providing the credentials to your application. You can either set the GOOGLE\_APPLICATION\_CREDENTIALS environment variable, or you can explicitly pass the path to the service account key in code. The first option is more secure and is strongly recommended.

Your app server or trusted server environment sends message requests to the FCM backend, which then routes messages to client apps running on users' devices.

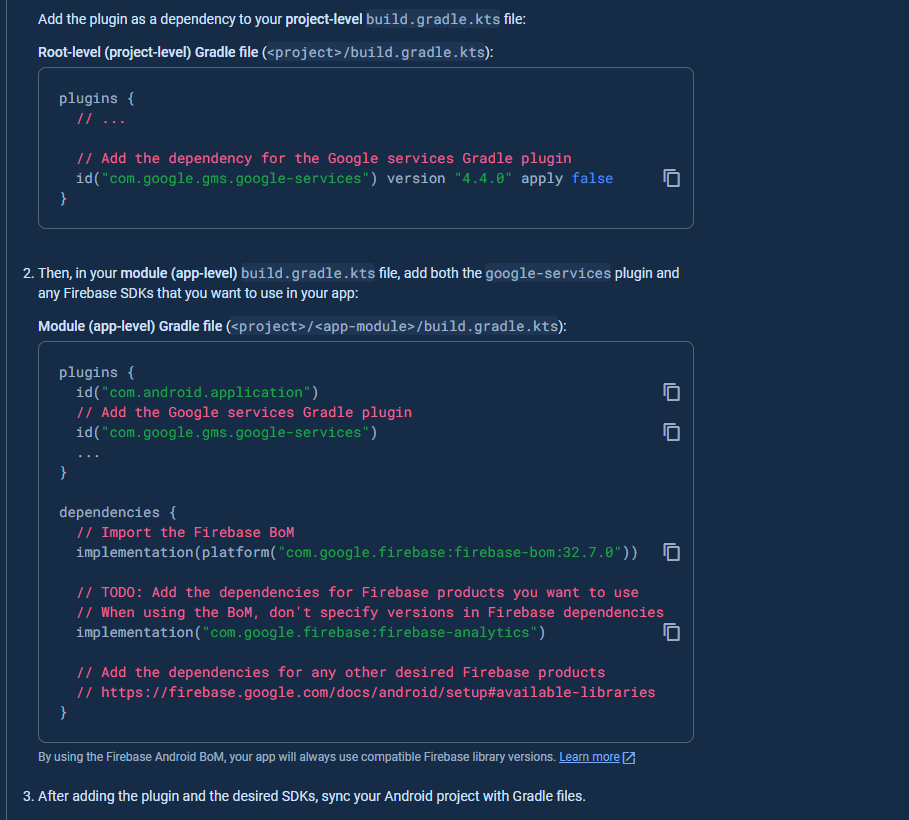


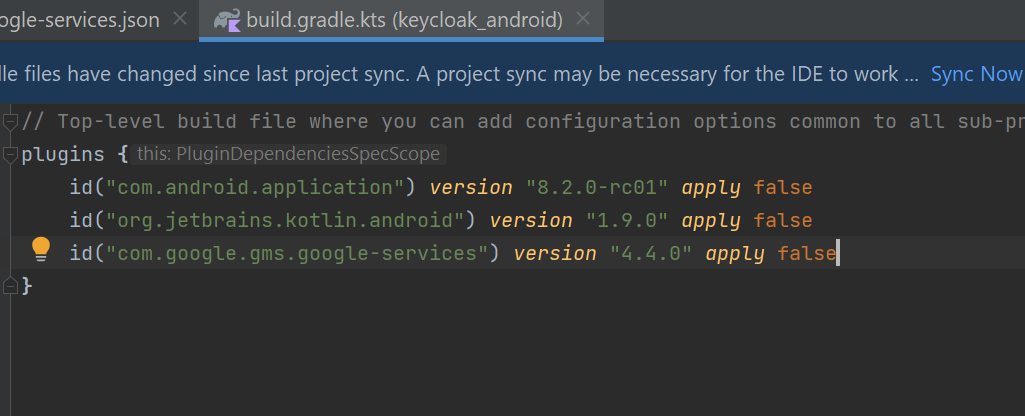
Pentru a adauga aplicatia android in Firebase:



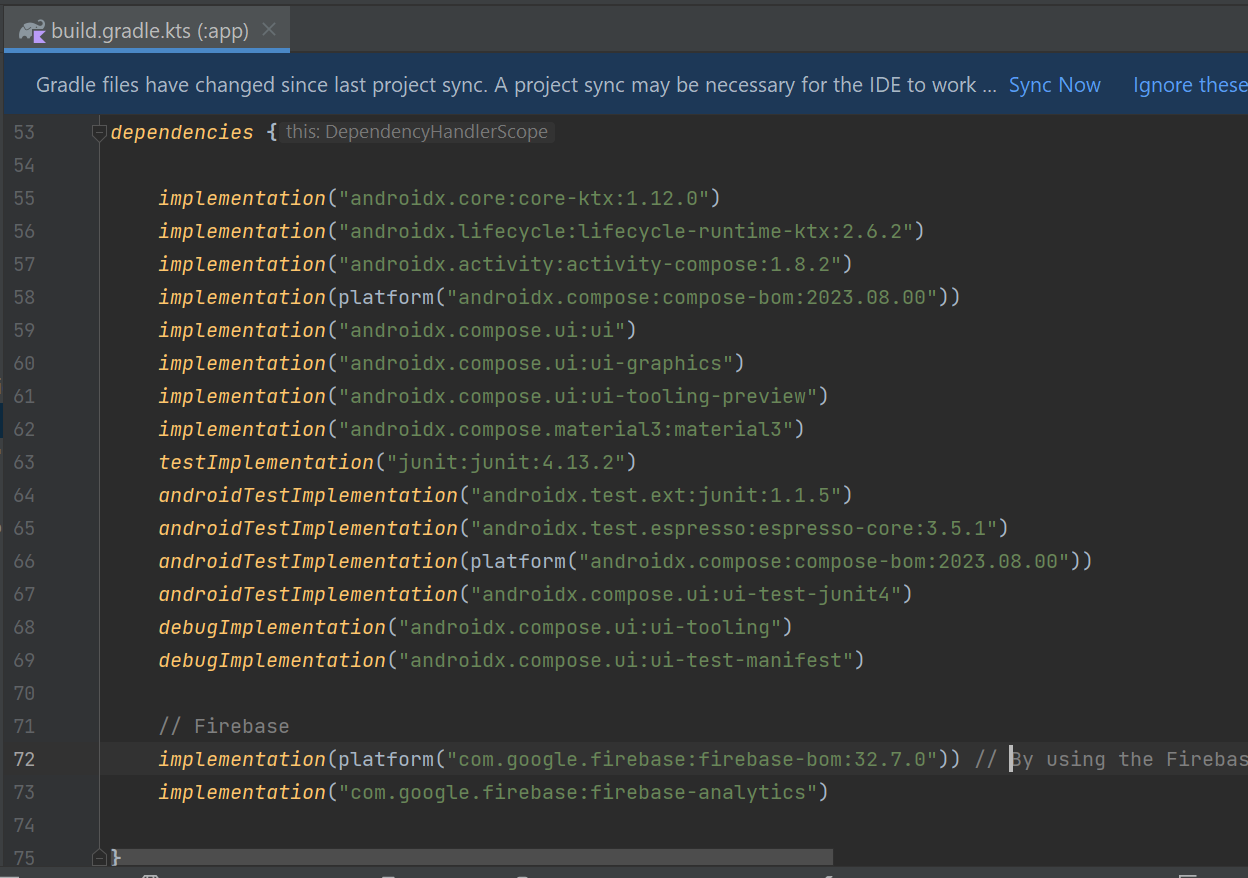


Dupa am putut descarca google-services.json in proiect. Am adaugat dupa plugin-ul in build-gradle(project level):





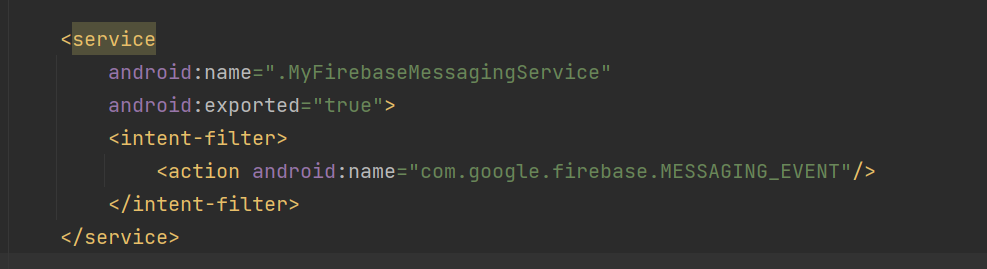
Dupa am adaugat in gradle(module:app) plugin-ul pentru google-services si dependintele pentru Firebase SDK.



Deoarece folosim Firebase BoM (Bill of Materials), nu trebuie să specificam o versiune pentru Firebase Messaging. Firebase BoM va gestiona versiunile tuturor produselor Firebase pe care le utilizam.

Dupa ce am facut toate aceste configurari am creat clasa MyFirebaseMessagingService care extinde FirebaseMessagingService si am suprascris metodele: onMessageReceived si onNewToken.

Dupa a trebuit sa inregistrez serviciul creat de mine in AndroidManifest.xml: (prin declararea in AndroidManifest.xml a serviciului meu fac constient SO Android de existenta si comportamentul serviciului meu si in cadrul serviciului o sa avem un intent-filter care specific ace tipuri de intent-uri sau mesaje pot fi gestionate de catre componenta Android)



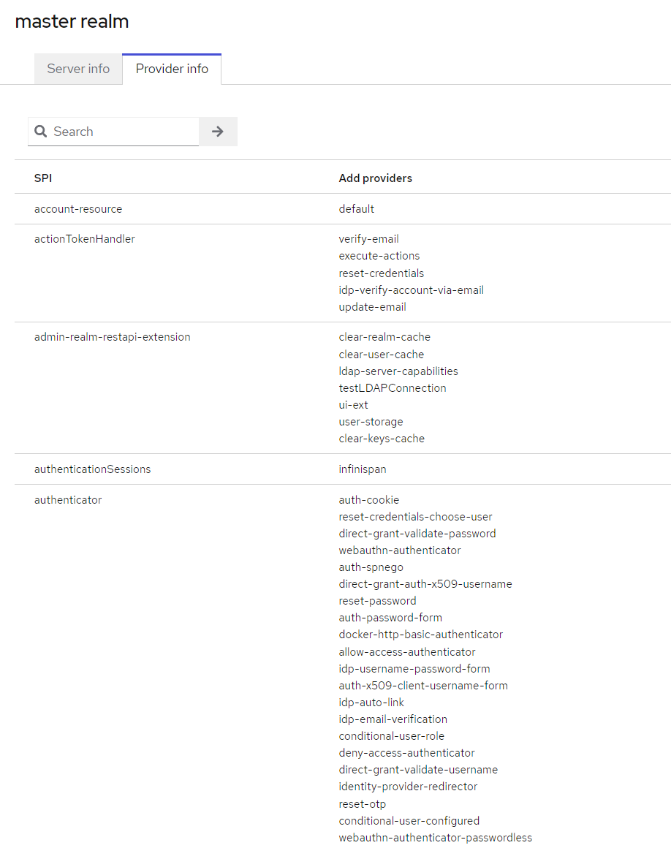
Am setat android:exported=”true” => este adesea setat pe "true" pentru a permite sistemului Android să trimită mesajele FCM către serviciul dvs. chiar și atunci când aplicația dvs. nu rulează activ sau nu este deschisă în mod explicit.

***Cum ar trebui sa arate flow-ul aplicatiei:***

* Aplicația Android obține un token FCM și îl trimite la serverul Keycloak. => la prima rulare a aplicatiei android se va obtine tokenul FCM si se va trimite catre keycloak pentru a il retine in baza de date.
* Când este necesară o autentificare prin notificare push, serverul (SPI-ul Keycloak) va folosi acest token FCM pentru a trimite o solicitare către serverul FCM, cerând ca o notificare push să fie livrată dispozitivului utilizatorului.
* FCM gestionează livrarea notificării către dispozitiv.
* Aplicația Android procesează notificarea și comunică rezultatul (de exemplu, aprobarea sau respingerea autentificării) înapoi la serverul Keycloak.

<https://www.baeldung.com/java-keycloak-custom-user-providers> - custom provider - SPI

* La inceput Keycloak va scana classpath-ul si va alege toti providerii disponibili folosind mecanismul standard: java.util.ServiceLoader.
* Aceasta inseamna ca tot ce trebuie sa facem este sa cream un fisier numit dupa un specific service interface pe care dorrim sa o implementam in forlderul META-INF/services al jarului nostru si sa punem numele complet calificat al implementarii noastre in el.
* Serviciile pe care le putem adauga in keycloak se afla pe pagina de informatii despre server, disponibila la consola de management a Keycloak:



* Documentatia Keycloak listeaza urmatoarele SPI-uri de interes pentru proiect:

1. ***org.keycloak.authentication.AuthenticatorFactory:*** defineste actiunile si flowurile de interactiune cerute pentru atentificarea unui user/client application
2. ***org.keycloak.authentication.actiontoken.ActionTokenHandlerFactory:*** ne permite sa crem actiuni custom pe care keycloak le va efectua la cererea /auth/realms/master/login-actions/action-token. De ex, acest mechanism se afla in spatele fluxului standard de resetare a parolei. Link-ul include in e-mail un astfel de action token.
3. ***org.keycloak.events.EventListenerProviderFactory:*** creeaza un provider pe care keycloak va ascult evenitmentele. **The EventType Javadoc page** contains a list of the available events custom a provider can handle. A typical use for using this SPI would be creating an audit database.
4. ***org.keycloak.storage.UserStorageProviderFactory:*** Allows Keycloak to access custom user stores
5. ***org.keycloak.vault.VaultProviderFactory:*** Allows us to use a custom vault to store Realm-specific secrets. Those can include information like encryption keys, database credentials, etc. => **as putea sa folosesc asta pentru tokenii FCM????**

Pentru a evita time-consuming prin compile-deploy-restart cycle pentru SPI-ul nostru intr-o instanta de Keycloak. O sa folosesc: embed KEycloak in proiect ca test-time dependency.

<https://www.baeldung.com/keycloak-embedded-in-spring-boot-app> - embed Keycloak => mai rapid pentru test

<https://stackoverflow.com/questions/62672377/implement-custom-spi-in-keycloak>

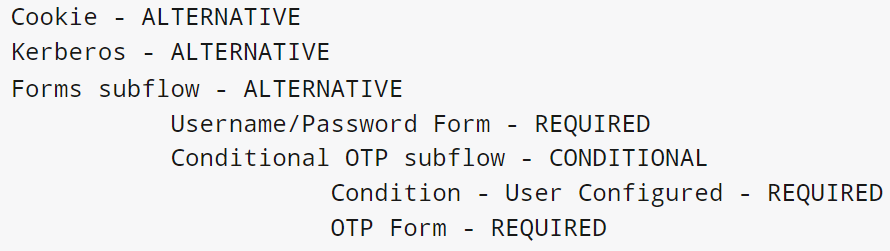
<https://github.com/adviqo-GmbH/keycloak/compare/master...adviqo-GmbH:feature/http-storage>

Sursa de inspiratie pentru partea de push notification: <https://www.keycloak.org/docs/latest/server_development/#terms> => Authentication SPI

Authenticator => reprezinta logica pentru autentificare sau actiunea dintr-un flow. De obicei este un singleton.

Execution => este un obiect care leaga autentificatorul de un flow si de configurarea acestuia. Flow-urile contin execution entries. Fiecare execution defineste cum un authentificator se poarta intr-un flow.

Alternative requirements => care pot fi validate intr-un flow dar nu sunt necesare.



ALTERNATIVE => daca oricare din cele 3 actiuni este indeplinita, atunci restul nu mai este necesar sa fie indeplinite pentru a trece la pasul urmator.

Pasii din spate:

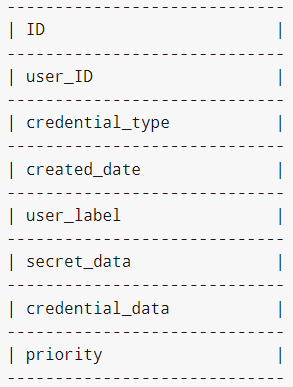
1. OpenID Connect despacheteaza data, verifica clientul si orice semnatura. Creeaza AuhtenticationSessionModel. Observa care browser flow ar putea fi si incepe executarea flow-ului.
2. Fluxul analizeaza cookie execution si vede ca este ALTERNATIVE. Incarca cookie provider. Verifica daca cookie provider solocita ca un utilizator sa fie deja asociat cu sesiunea de autentificare. Cookie provider nu necesita un user. Daca ar face acest lucru, fluxul s-ar anula si ar da eroare. Scopul sau este de a vedea daca exista un set de cookie-uri SSO. Daca exista un set, acesta este validat si UserSessionModel este verificat si asociat cu AuthentificationSessionModel. Daca cookie-ul SSO exista si este valdat => success(). => conectare cu succes fara a fi nevoie si de restul care sunt puse pe acest nivel ca ALTERNATIVE. Daca nu exista un cookie SSO => cookie provider returneaza => attempted(). (nici succes dar nici eroare.). Furnizorul a încercat, dar cererea pur și simplu nu a fost configurată pentru a gestiona acest autentificator.
3. Daca nu e ok cu cookie trecem de ex la urmatorul de pe acest nivel: Kerberos. Kerberos provider de asemenea nu necesita ca un utilizator sa fie deja configurar si asociat cu AuthentificationSessionModel astfel incat acest furnizor este executat. Utilizeaza protocolul browser SPNEGO. Acest lucru necesita o serie de provocari/raspunsuri intre server si client care fac schimb de anteturi de negociere. Furnizorul Kerberos nu vede nuciun antet de negociere asa ca presupune ca aceasta este prima interactiune intre server si client. Asadar, creeaza un raspuns de provocare HTTP pentru client si seteaza o stare forceChallenge(). Un forceChallenge() inseamna ca acest raspuns HTTP nu poate fi ignorat de flux si trebuie returnat clientului. Daca in schimb providerul a returnat o stare challenge() fluxul ar retine raspunsul la provocare pana cand sunt incercate toate celelalte alternative. Deci, în această fază inițială, fluxul s-ar opri și răspunsul la provocare ar fi trimis înapoi către browser. Dacă browserul răspunde apoi cu un antet de negociere reușit, furnizorul asociază utilizatorul cu AuthenticationSession și fluxul se termină deoarece restul execuțiilor la acest nivel al fluxului sunt toate alternative. În caz contrar, din nou, furnizorul kerberos setează o stare de attempted() și fluxul continuă.
4. Urmatorul flux de pe acest nivel este Forms. Aici se solicita Username si Parola. Acest furnizor nu necesita de asemenea ca un utilizator sa fie asociat cu fluxul. Acest furnizor creeaza un raspuns HTTP de provocare si isi seteaza starea la challenge(). Aceasta executie este necesara, astfel incat fluxul onoreaza aceasta provocare si trimite raspunsul HTTP inapoi catre browser. Acest raspuns este o redare a paginii HTML cu nume/parola. In caz de esec => failureChallenge(). Dacă numele de utilizator și parola sunt valide, furnizorul a asociat UserModel cu AuthenticationSessionModel și returnează un status de succes().
5. Urm executie este metoda de 2FA: de exemplu: OTP. Acest furnizor necesita, de asemenea, ca un utilizator sa fi fost asociat fluxul. Aceasta cerinta este indeplinita deoarece una din metodele de mai sus au asocita userul cu fluxul. Furnizorul esre intrebatdaca utilizatorul este configurat sa utilizeze acets furnizor. Daca utilizatorul nu este configurat, atunci fluxul va configura o actiune necesara pe care utilizatorul trebuie sa o efectueze dupa finalizarea autentificarii. Pentru OTP, aceasta inseamna pagina de configurare OTP. Daca utilizatorul este configuratm I se va cere sa introduca codul sau OTP. În scenariul nostru, din cauza fluxului secundar condiționat, utilizatorul nu va vedea niciodată pagina de conectare OTP, cu excepția cazului în care fluxul secundar OTP condiționat este setat la Necesar.
6. Dupa ce fluxul este complet, se creeaza UserSessionModel si se asociaza cu AuthenticationSessionModel. Apoi verifica daca utilizatorului I se cere sa eefctueze actiunile necesare inainte de a se conecta.
7. In primul rand este apelata metoda ***evaluateTriggers()*** a fiecarei required actions. Acest lucru permite furnizorului de actiuni necesare sa-si dea seama daca exista o stare care ar putea declansa actiunea. De ex: daca relm-ul are o politica de expirare a parolei => aceasta ar putea fi declansata prin aceasta metoda.
8. Required action => ***requiredActionChallege().*** Aici furnizorul seteaza un raspuns HTTP care reda pagina pentru actiunea necesara. Acest lucru se face prin setarea unui statut de provocare.
9. Daca required action este in cele din urma cu succes, atunci actiunea necesara este eliminate din lista de actiuni necesare a utilizatorului.
10. Dupa ce toate actiunile necesare au fost rezolvate, utilizatorul este in sfarsit conectat.

***EXEMPLU de SPI implementare intrebare secreta (vezi cod in examples la providers in codul sursa)***

* Ne intereseaza interfata ***Authenticator***
* Trebuie implementate interfetele: org.keycloak.authentication.AuthenticatorFactory si Authenticator (aceasta interfata defineste logica)
* Cum ar fi OTPFormAuthenticator si PasswordAuthentication => sunt autentificatori **CredentialValidators** => ca si push notification de altfel. Trebuie implmentate alte cateva clase:
  + Org.keycloak.credential.CredentialModel => genereaza formatul corect al acreditarii in baza de date
  + Org.keycloak.credential.CredentialProvider => si clasa si interfata si CredentialProviderFactory

SecretQuesionAuthenticator pe care il avem ca exemplu in aceasta parcurgere este in CredentialValidator.

1. Toate clasele vor fi impachetate intr-un singur jar. Acest jar trebuie sa contina un fisier numit ***org.keycloak.authentication.AuthenticatorFactory*** si trebuie sa fie continut in **META-INF/services/ .** Acest fisier continut de acest director ar trebui sa listeze numele de clasa complet al fiecarei implementari AuthenticatorFactory pe care o avem in acest jar. (ex: org.keycloak.examples.authenticator.SecretQuestionAuthenticatorFactory). Acest fisier/serviciu este folosit de Keycloak pentru a scana furnizorii pe care trebuie sa ii incarce in sistem.
2. La final acest jar trebuie copiat in directorul providers.
3. In Keycloak, credentialele sunt salvate in baza de date in tabelul **Credentials** si are urmatoarea structura:



Credentiala <=> acreditare

* ID => cheia primara
* user\_ID => este cheia externă care leagă credentialele la un utilizator.
* credential\_type => este un șir setat în timpul creării care trebuie să facă referire la un tip de autentificare existent. (ex: SECRET\_QUESTION)
* user\_label => este numele editabil al credentialului de către utilizator
* secret\_data => conține un json static cu informații care nu pot fi transmise în afara Keycloak
* credential\_data => conține un json cu informațiile statice ale acreditării care pot fi partajate în Consola de administrare sau prin API-ul REST.
* priority => definește cât de „preferată” este o acreditare pentru un utilizator, pentru a determina ce acreditare să prezinte atunci când un utilizator are mai multe opțiuni.

1. Clasa ***CredentialModel:***
   * Avem campurile de credential\_data si secret\_data => care sunt sub forma de json-uri.

De ex: aici credential\_data va fi reprezentat de question (un singur camp in json) si secret\_data va fi reprezentat de answer.

* + Este bine sa avem si un salt si un algoritm cu nr de iteratii in aceste json-uri pentru a cripta datele in baza de date. => pentru secret\_data. (ex: parolele au un hash sarat: org.keycloak.models.credential.PasswordCredentialModel)
  + In exemplul nostru este clasa **SecretQuestionCredentialModel. =>**  avem metode pentru a prelua datele din baza de date dar si de a crea noi obiecte
  + Cand vrem sa obtinem tipul unei acreditari, trebuie sa ne asiguram ca este cel din credential\_type.

1. Implementare ***CredentialProvider:***
   * Avem nevoie si de un **CredentialProviderFactory** pentru a implementa un CredentialProvider
   * Interfata CredentialProvider preia un parametru generic care extinde un CredentialModel => ex: SecretQuestionCredentialModel.
   * ***CredentialInputValidator*** =>permite Keycloak sa stie ca acest furnizor poate fi folosit si pentru a valida o autentificare pentru un Authenticator.
   * Prima metoda care trebuie implementata este **getType()** => sirul TYPE.
   * A doua metoda ce trebuie implementata este de a crea un **SecretQuestionCredentialModel** dintr-un **CredentialModel** => avem o metoda speciala facuta in SecretQuestionCredentialModel.
   * Metode de: a crea o autentificare si de a sterge o autentificare. Aceste metode apeleaza UserModel’s credential manager, care este responsabil sa stie unde sa citeasca/sa scrie credentialele, de ex local storage sau federated storage.
   * Pentru interfata CredentialInputValidator avem metoda principala **isValid() =>** care testeaza daca o acreditare este valida pentru un anumit utilizator intr-un anumit domeniu. Aceasta este metoda apelata de Authenticator atunci cand incearca sa valideze intrarea utilizatorului. (ex: pentru exemplul nostru trebuie sa verificam daca sirul de intrare este cel inregistrat in Credential la secret\_data)
   * Mai avem doua metode prin care se verifica daca CredentialProvider accepta tipul de autentificare dat si un test pentru a verifica daca tipul de autentificare este configurat pentru un anumit utilizator. Ultimul test inseamna verificarea daca utilizatorul are o acreditare de tip SECRET\_QUESTION.
2. Implementarea unui autentificator
   * Un autentificator utilizeaza acreditarile pentru a autentifica un utilizator => trebuie implementata interfata **CredentialValidator**
   * Interfata de CredentialValidator => ia o clasa CredentialProvider ca parametru si ii permite sa apeleze functiile implmentate in aceasta si singura metoda care trebuie implementata esre **getCredentialProvider()**.
   * La implementarea interfetei Autheticator prima metoda ce trebuie implementata este **requiresUser()** => aceasta metoda trebuie sa returneze true deoarece trebuie sa validam intrebarea secreta asociata utilizatorului. (ex: un provider de Kerberos ar returna false din aceasta metoda, deoarece poate rezolva un utilizator din antetul de negociere , dar in ex prezentat mai sus valideaza o anumita autentiifcare a unui anumit utilizator.)
   * Metoda **configuredFor()** => metoda care este responsabila pentru a determina daca utilizatorul este configurat pentru acest autentificator particular.
   * Metoda **setRequiredActions**() => daca configuredFor() returneaza false si exemplul nostru de autentifcare este necesar in flux, aceasta metoda va fi apelata, dar numai daca **isUserSetupAllowed** returneaza true. Aceasta functie este deci responsabila pentru inregistrarea oricaror actiuni necesare care trebuie effectuate de un utilizator. (ex: inregistram o actiune necesara care va forta utilizatorul sa configureze raspunsul la intrebarea secreta.)
   * Metoda **authenticate() =>** metoda initiala pe care fluxul o invoca atunci cand executie este vizitata pentru prima data. (ceea ce se doresti de aici este ca atunci cand un utilizator a raspuns la intrebarea secreta deja pe computeruk browserului sau, atunci utilizatorul sa nu mai fie nevoit sa raspunda la intrebare => masina este de incredere). Aceasta metoda nu este responsabila pentru a procesa raspunsul la intrebarea secreta ci doar sa redea pagina corespunzatoare sau de a continua fluxul.
   * Metoda **hasCookie**() => verifica daca exista deja un cookie setat in browser (la intrebare s-a raspuns deja).
   * Exista o sectiune a codului referitoare la obtinerea ID-ului acreditarii. Acest lucru se datoareaza faptului ca daca Keycloak este configurat sa permita mai multe tipuri de autentificatoare alternative sau daca utilizatorul ar putea inregistra mai multe acreditari de tip SECRET\_QUESTION (de ex: daca am permis sa alegem dintre mai multe intrebari si am permis utilizatorului sa aiba raspunsuri pentru mai mult de una dintre aceste intrebari), atunci Keycloak trebuie sa stie ce acreditare a fost folosita pentru a inregistra utilizatorul.
   * In cazul in care exista mai multe acreditari, Keycloak permite utilizatorului sa aleaga in timpul autentificarii care credentiale vor fi folosite (adica ce tip de acreditare este folosita), iar informatiile sunt transmise mai departe prin formular catre Autentificator.
   * În cazul în care formularul nu prezintă aceste informații, id-ul de acreditare utilizat este dat de metoda CredentialProvider default getDefaultCredential, care va returna acreditările „cel mai preferate”.
   * Metoda **getCredentialTypeMetadata(CredentialTypeMetadataContext metadataContext)** este o metoda abstracta a interfetei CredentialProvider. Fiecare furnizor de acreditari trebuie sa furnizeze si sa implementeze aceasta metoda. Metoda returneaza o instanta de CredentialTypeMetadata => care include cel putin tipul si categoria de autentificare. => vezi exemplu de la Secret\_Question
   * createAction(ID-ul furnizorului actiunii necesare) sau updateAction (actualizeaza acreditare)
   * **AutenticatorFactory** => ofera metodate de implementare si configurare despre Autenticator
   * **CONDITIONAL =>** ar trebui sa fie intotdeauna utilizat numai pentru fluxuri secundare si, cu exceptia cazului in care exista un motiv intemeiat pentru a proceda altfel, cerinta privind un autentificator ar trebui sa fie **NECESARA, ALTERNATIVA si DEZACTIVATA.**
   * **AuthenticatorFactory.isUserSetupAllowed**() => spune managerului de flux daca va fi apelata sau nu metoda Authenticator.setRequiredActions(). Dacă un Autentificator nu este configurat pentru un utilizator, managerul de flux verifică isUserSetupAllowed(). Dacă este fals, atunci fluxul se întrerupe cu o eroare. Dacă returnează adevărat, atunci managerul de flux va invoca Authenticator.setRequiredActions().
   * Funcția **Authenticator.setRequiredActions()** din Keycloak este folosită în cadrul procesului de autentificare pentru a adăuga acțiuni necesare (required actions) care trebuie să fie efectuate de utilizator înainte de a finaliza procesul de autentificare. Scopul principal al acestei funcții este de a asigura că anumite condiții sau cerințe de securitate sunt îndeplinite înainte ca utilizatorul să fie considerat autentificat complet. Acest lucru poate include resetarea parolei, verificarea adresei de email, configurarea autentificării cu două factori, sau orice altă acțiune personalizată definită în sistem. (De exemplu, în contextul implementării unui sistem de autentificare cu două factori, după prima autentificare cu succes folosind numele de utilizator și parola, funcția **setRequiredActions()** poate fi utilizată pentru a solicita utilizatorului să configureze un al doilea factor de autentificare (cum ar fi un token mobil sau o aplicație de generare de coduri OTP) înainte de a putea accesa complet serviciile.)
   * **isConfigurable()** => este un indicator care specifica Consolei de administrare daca Autentificatorul poate fi configurat intr-un flux.
   * **Fiecare ProviderConfigProperty defineste numele proprietatii de configurare => aceasta este cheia folosita in harta de configurare stocata in AuthenticatorConfigModel.** Eticheta defineste modul in care va fi afisata optiunea de configurare in Consola de admin.
3. Adaugarea unui formular de autentificare
   * Metoda createForm() pe care trebuie sa o apelam in authenticate() pentru a construe o pagina HTML dintr-un fisier din tema de conectare: “secret-question.ftl” => acest fisier ar trebui adaugat in **theme-resources/templates** in jar.
4. Adaugare Required Action
   * Pentru a implementa actiunea de raspuns de exemplu la intrebare trebuie sa cream interfata RequiredActionFactory: deci adaugam META-INF/services/ un fisier numit: org.keycloak.authentication.RequiredActionFactory => in el punem toate implementarile RequiredActionFactory

La final pentru a adauga jar-ul trebuie sa il copiem in providers/ si sa executam bin/kc.[sh | bat] build .

8.1. RequiredActionProvider

- avem functia de **requiredActionChallenge()** => metoda este responsabila pentru redarea formularului HTML care va conduce actiunea necesara (metoda challenge() informeaza managerul de flux ca o actiune necesara trebuie executata)

- metoda **processAction()** => procesarea intrarii din forma HTML a actiunii necesare. Răspunsul este scos din postarea formularului. Este creat un UserCredentialValueModel și sunt setate tipul și valoarea acreditării. Apoi este invocat UserModel.updateCredentialDirectly(). În cele din urmă, RequiredActionContext.success() notifică containerul că acțiunea necesară a avut succes.

***Activați acțiunea necesară***

Ultimul lucru pe care trebuie să-l faci este să intri în Consola de administrare. Faceți clic pe meniul din stânga Autentificare. Faceți clic pe fila Acțiuni necesare. Faceți clic pe butonul Înregistrare și alegeți noua acțiune necesară. Noua dvs. acțiune necesară ar trebui acum să fie afișată și activată în lista de acțiuni necesare.

**Ce ar trebui eu sa fac???**

1. Trebuie sa implementez interfata **Authenticator** pentru a crea propriul provider de autentificare. Acest provider va gestiona logica specifica pentru etapa de autentificare cu notificare push
2. Ma intereseaza metodele din aceasta interfata: authenticate, action, requiresUser. In authenticate trebuie implementata logica pentru a trimite o notificare push catre dispozitivul utilizatorului.
3. Trebuie sa creez sau sa modific un flux de autentificare existent pentru a include etapa de autentificare prin notificare push ca un pas suplimentar dupa autentificarea initiala.
4. Folosirea interfetei VaultProviderFactory pentru stocarea tokenurilor FCM => utilizarea unui vault ofera un nivel suplimentar de Securitate prin criptarea datelor sensibile si prin centralizarea gestionarii acestora.

**Infinispan + Cookies**

Cookies:

* Sunt folosite pentru a gestiona sesiuni utilizator in aplicatia web.
* Un cookie de sesiune este setat in browseul utilizatorului cand acesta se autentifica si este trimis inapoi la server la fiecare cerere, permitand serverului sa recunoasca sesiunea active a utilizatorului
* Pe partea de Securitate: Trebuie luate masuri de Securitate, cum ar fi utilizarea flag-urilor Secure si HttpOnly => pentru a reduce interceptarea sesiunilor (session hijacking) si atacurile cross-site scripting (XSS)

Infinispan:

* Este o solutie de caching distribuit si o solutie de sotcare a datelor in memorie care poate fi folosita pentru a gestiona sesiuni active intr-o maniera scalabila
* In Keycloak, Infinispan este adesea utilizat pentru a stoca sesiuni si alte date la nivel de cluster, permițând o gestionare eficientă a sesiunilor utilizatorilor și replicarea/stocarea distribuită a datelor de sesiune.

Keycloak:

* Keycloak gestionează sesiunile utilizatorilor folosind **un amestec de cookies (pentru a identifica sesiunile la nivelul clientului) și Infinispan (pentru a stoca datele de sesiune și a le gestiona într-un mediu distribuit).** Când un utilizator se autentifică, Keycloak creează o sesiune de utilizator care este reținută în Infinispan și utilizează un cookie de sesiune pentru a asocia cererile clientului cu sesiunea respectivă.
* **Cookies:** Sunt folosite pentru a menține starea sesiunii între client (browser) și server, facilitând recunoașterea utilizatorului la fiecare cerere fără a fi necesară reautentificarea.
* **Infinispan**: Folosit de Keycloak pentru a stoca detalii ale sesiunii și a permite gestionarea eficientă a sesiunilor într-un mediu distribuit sau clusterizat, asigurând scalabilitatea și performanța sistemului de autentificare.

Keycloak folosește Infinispan ca sistem de caching la nivel intern pentru a gestiona datele de **sesiune, tokenurile**, și **alte informații temporare**. Aceasta înseamnă că, chiar și într-o instanță singură (non-clusterizată) a Keycloak, Infinispan este folosit pentru a îmbunătăți performanța și eficiența prin reducerea accesului la baza de date și prin stocarea datelor în memorie.

Cache-ul Implicit: Keycloak configurează și utilizează Infinispan pentru cache-ul său implicit. Acest lucru include cache pentru sesiuni, utilizatori, autorizații și altele. Nu este necesară o configurare suplimentară pentru a beneficia de aceste optimizări de performanță.

**Avantaje ale Folosirii Infinispan pentru Codurile de Acces 2FA:**

- Dacă doar aplicația Keycloak și extensiile sale (de exemplu, un custom Authenticator sau un Credential Provider) au acces la cache-ul Infinispan, riscul de expunere externă este minimizat. Accesul restricționat asigură că doar componentele autorizate pot accesa sau modifica datele stocate.

- Infinispan oferă performanțe înalte pentru operațiunile de citire și scriere, ceea ce este esențial pentru un proces de autentificare rapid și eficient. De asemenea, suportă expirarea automată a datelor, ceea ce este ideal pentru codurile de acces temporare folosite în 2FA.

- Utilizând Infinispan, beneficiezi de scalabilitatea și gestionarea eficientă a sesiunilor și datelor temporare pe care Keycloak le implementează nativ. Acest lucru te ajută să menții performanța sistemului chiar și pe măsură ce numărul de utilizatori crește.