Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Ghadri Najib

PLATFORMFÜGGETLEN SZOLGÁLTATÁS INTEGRÁLÓ RENDSZER

Szakdolgozat (BSc)

KONZULENS

Dr. Simon Balázs

BUDAPEST, 2017

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Ghadri Najib**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 12. 08.

...…….
Ghadri Najib

Összefoglaló

Napjainkban a webalkalmazások használata egyre nagyobb szerepet játszik feladataink végrehajtásában. Munkánkat egyre több webalkalmazás együttes használatával végezzük a böngészőnkben, egyaránt a mobileszközeinkről és az asztali számítógépjeinkről.

A webes világ mellett a mobil világ is gyorsan fejlődik. Ma már egy okostelefonon képesek vagyunk gyorsabban elvégezni a feladatokat, mint egy számítógépről. Ez nagyban köszönhető annak, hogy a telepített mobilalkalmazásaink képesek egymás segítségével feladatokat elvégezni. Ez pedig annak köszönhető, hogy a mobil alkalmazások egy egységes keretrendszerbe integráltak, ami miatt közös módon tudnak kommunikálni, és feladatokat megosztani egymással, mint például az Androidban.

A webes világból viszont hiányzik egy közös integrációs keretrendszer, emiatt különböző vállalatoktól származó de hasonló szolgáltatások teljesen különböző interfészekkel rendelkeznek, így egy kliens az összes hasonló webszolgáltatás különböző interfészéhez kell külön-külön kapcsolódjon, ami N-szeres fejlesztési időt igényel.

A megoldás az, hogy egy közös webszolgáltatás integráló keretrendszerhez igazítsuk a webalkalmazásokat. A hasonló funkcionalitású webalkalmazásokhoz egységes interfészeket definiálhatunk, amit a vállalatok közösen fejleszthetnek. A webszolgáltatásokat egy közös nyilvántartásban rendeljük interfészekhez. Ennek következtében a szolgáltatás nyilvántartás képes egy kliensnek dinamikusan kiosztani több kompatibilis interfészű szolgáltatást, így elég egy darab klienst készíteni N helyett. A keretrendszerben a webalkalmazások közötti feladatmegosztáshoz az Androidban alkalmazott hatékony utasításküldési architektúra mintájára egy univerzális megoldást tervezek, amivel a felhasználók képesek a csatlakoztatott webalkalmazásaik jogosultságait ugyanabban a központi, felhőalapú szolgáltatás nyilvántartó rendszerben kezelni. A keretrendszer egy hatékony fejlesztői módszert kínál a web alkalmazások és webszolgáltatások integrálásához.

Abstract

Nowadays we use more and more web applications in our daily life for all kinds of tasks. We do our work using many web applications together from our browser on either a mobile device or a personal computer.

Besides the web, the mobile world is also growing and improving fast. Today we can do some tasks faster than on a computer. This is because mobile applications are integrated into a common framework, thus they are able to share data, communicate and command each other in a common way, the way it is in Android for example.

However this is a missing piece from today's web applications. Thus web services that are from different vendors, but share similar functionality still cannot be accessed in a common way from a client, and hence the client needs to implement an interface for each web service.

The solution is to use a common web service integration framework. Web services that share similar functionality should use common interfaces, that vendors should develop together. A central web service registry should map the interface definition to the web service implementations. This way the service registry can delegate implementations to a client that uses one compatible interface, and this way a client needs only to implement one interface for a well-defined functionality. In this framework for the interoperability and command between web applications I got inspired by Android, and I used many of its solution to specify a way to connect web applications simply and effectively. A user of this framework is able to manage all web applications and web services and the connections between them in a common and convenient way. This framework provides and efficient way to integrate web applications and web services.

1 Bevezetés

Napjainkban egyre inkább webes alkalmazások használatával éljük mindennapjainkat, ellentétben az utókorral, amikor is lokálisan, számítógépekre telepített alkalmazásokra és lokális adatátvitelre tudtunk csak hagyatkozni. A webes alkalmazások/szolgáltatások fő tulajdonsága, hogy elosztottak, tehát az internetes hálózaton lévő egymástól független számítógépeken futnak. Ettől függetlenül a mindennapi felhasználók sajátjuknak tudják a Google Drive-jukat, Facebook-, Neptun-, Gmail-üket és még sok, egyébként nem a számítógépünkön futó alkalmazást. Ezért manapság az operációs rendszereinket egyre inkább böngészésre használjuk, ami már nem csak böngészés, hanem webalkalmazás használat is.

Ettől függetlenül még létezik olyan rendszer, ahol a lokális alkalmazások/szolgáltatások fejlettsége is nagyon számít. Ez a mobil világ. Mobiljainkon is használunk webes alkalmazásokat, de megjelent az igény arra, hogy a lokálisan telepített alkalmazások könnyen telepíthetők, törölhetők és kezelhetők legyenek, és hogy egymás funkciót felhasználhassák. A megoldás az lett, hogy a lokális alkalmazások meg kell feleljenek egy keretrendszernek, hogy közös módon tudjanak kommunikálni, adatot cserélni, és egymás szolgáltatásait felhasználni. Erre példa napjainkban az Android vagy az iOS operációs rendszer.

A webes szolgáltatások viszont nehezen képesek más webesalkalmazással együtt működve elvégezni egy feladatot, főleg, ha az a másik szolgáltatás egy másik vállalattól származik. Például, ha szeretnénk egy olyan alkalmazást írni, ami az összes e-mail-fiókunkból kiszűri azokat a beérkező e-maileket, amik számlaértesítők, és megjeleníti egy felületen esetleg értesítéseket ad, akkor ahhoz, hogy a világon a legtöbb felhasználót megcélozzuk, akkor világ összes komolyabb e-mail szolgáltatójának az egymástól teljesen különböző webes API-dokumentációját kell használni. Látjuk, hogy lehetetlen feladat előtt állunk.

Egy másik mindennapi példa a telefonjainkon jól ismert megosztási funkció, amikor is egy képet vagy szöveget vagy bármilyen fájlt/adatot szeretnénk egyik alkalmazásból egy másikba küldeni és ott felhasználni. Mobiltelefonjainkon alkalmazásból alkalmazásba ugrunk, hogy egy bizonyos dolgot végrehajtsunk, tehát valahogyan ez a kommunikáció a mobil alkalmazások között lehetséges. Képzeljük el

hogy a webes Google Drive-ban egy kiválasztott fájlt nem csak a Gmail-el lehetne megosztani, hanem a Facebook Messengerrel, Slack-el, és akármilyen másik megosztási utasítást támogató webalkalmazással. Ez a funkció egy böngészőben a webalkalmazások között egyelőre nem egy kidolgozott probléma.

A megoldás az, hogy ahogyan a lokális alkalmazásokat is egy egységes keretrendszer használatára ösztönözzük, úgy a webes alkalmazásokat is. Dolgozatomban erre adok egy olyan megoldást, amiben megpróbálom a legtöbb webes szolgáltatás megcélozni, úgy, hogy minél kevésbé keljen nagy változtatást végezni rajtuk és a legfelhasználóbarátibb legyen mind a rendszerre való fejlesztés mind a használat.

2 Szolgáltatás integráció napjainkig

Már korábban is felmerült az igény arra, hogy webszolgáltatások összekapcsolására és felderítésére egy standard létezzen, amikor is 2000-ben a Microsoft, IBM és további cégek közösen fejlesztették ki és majd tovább a Web Service Definition Language – (WSDL) és a Universal Description, Discovery and Integration (UDDI) azaz *univerzális leírás, felfedezés és integrálás* specifikációkat azért, hogy a számítógépek közötti (machnine-to-machine) szolgáltatásaikat egységesen leírhassák és összeköthessék [1]. A WSDL egy XML alapú webszolgáltatás interfész leíró nyelv az UDDI pedig a WSDL-ben megírt szolgáltatás leírók alapján tud szolgáltatásokat kiajánlani más szolgáltatásnak, másszóval egy UDDI szerverben lehetne keresni WSDL-ben megírt szolgáltatás API-k közül. A szolgáltatások között az adatközlés az XML alapú Simple Object Access Protocol – SOAP szövegekkel történik.

Az UDDI [2] egy nyílt standard, ami specifikálja a szolgáltatások jegyzésének és keresésének módját. Ez a rendszer egy univerzális és uniform módot biztosított az összes vállalat számára arra, hogy szolgáltatásokat publikáljon egy közös adatbázisba, ahonnan más vállalatok vagy akár hétköznapi felhasználók ezeket a szolgáltatásokat böngészni használni tudják. Úgy képzelték el mintha a szolgáltatások mai Play Store-ja lenne. Hasonlóan is működik, hasonló módon kellett szolgáltatásokat regisztrálni (vállalt információk, WSDL definíció, kulcsok és hasonló lekérdezési módszer). Bár mindennapi szolgáltatások kezelésére találták ki, mint például egy névjegyzék szolgáltatás, a rendszer érdekes módon sosem volt éles két fő probléma miatt: túl nagy komplexitás [3], biztonsági kérdések mellőzése [4].

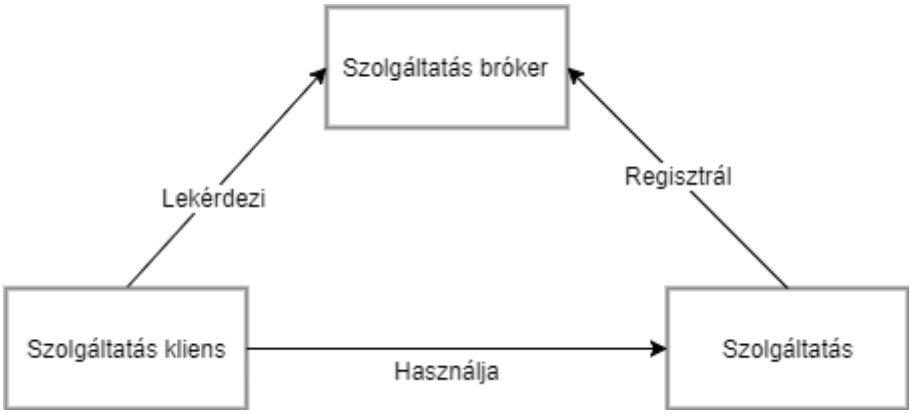
2.1 SOA

Az ilyen szolgáltatás architektúrát, mint a UDDI-t Service-Oriented Architecture-nek azaz SOA-nak hívjuk. A SOA [5] egy szoftver tervezési módszer, aminek lényege, hogy az egyes szolgáltatások egy hálózatban helyezkednek el, és egy protokoll által meghatározott módon kommunikálnak. Alapvető elvei függetlenek gyártóktól, termékektől, terjesztőktől és technológiáktól. A szolgáltatások különálló egységek, amik távolról is elérhetők, egymástól függetlenül elérhetők, használhatók,

frissíthetők, újra kombinálhatók a folyamatok folytonos változásának, megújulásának megfelelően. Egy szolgáltatásnak a következőket kell teljesítenie:

- Logikailag reprezentál egy üzleti aktivitást előre meghatározott kimenettel.
- Önálló, független egység (self-contained).
- Fekete doboz a kliensei számára.
- Tartalmazhat több részszolgáltatást.

Egy SOA alapú rendszerben alapvetően három szereplő van [6]:



1. ábra: Szolgáltatás-orientált architektúra

A szolgáltatás-bróker, szolgáltatás kliens és a szolgáltatás szolgáltató. A szolgáltatás-bróker feladata a szolgáltatások nyilvántartása és biztosítani a szolgáltatások rendszerezett és felügyelt regisztrálását és lekérdezését. Egy bróker nyilvántartást tart a regisztrált szolgáltatásokról és hogy milyen interfészeket implementálnak. A szolgáltatás kliens kérdezi le a számára megfelelő szolgáltatásokat a bróker nyilvántartásából a bróker által támogatott lekérdezési módon. A bróker tehát szétválasztja a klienst és a szolgáltatást, így dinamikusan lehet kapcsolatot teremteni.

A szolgáltatás szolgáltató feladata, hogy szolgáltatását és további kötelező információt beregisztrálja a brókerhez. A szolgáltatás szolgáltatójának kötelessége, hogy a hirdetett végponton a szolgáltatás működő képes legyen. Lehetősége van díjat kiszabni a szolgáltatás használatára, amit a kliens közvetetten a brókeren keresztül (mint például ahogyan a Google Play Store-ban történik) vagy közvetlenül a szolgáltatónak fizeti ki. A rendszerezettség alatt azt értjük, hogy a szolgáltatások egyértelműen azonosíthatók és felügyeltség alatt azt, hogy a szolgáltatók identitása hitelesített.

Mind mai napig nem terjedt el egy publikus SOA rendszer. A UDDI volt a legutóbbi komolyabb kísérlet erre. Ennek ellenére létezik több privát SOA rendszer is, az egyik példaértékű a Google API Discovery Service [7]. Ez a rendszer leginkább a Google saját használatra tervezte, hogy a saját szolgáltatásait kösse össze. Ez a szolgáltatás is egy fajta szolgáltatás-brókert valósít meg. Lehetőség van REST API definíciókat lekérdezi JSON formátumban, regisztrálni viszont nem lehet, hiszen csak a Google API-k közül tudunk lekérdezni, emiatt nem publikus ez a SOA rendszer. A JSON leírók egy API-t írnak le, olyanokat, mint például a Gmail API, Youtube API stb. Ezt a rendszert leginkább kliens könyvtárak a Google szolgáltatások listázására használják, és a Google belső szolgáltatásaik használják. Mivel webszolgáltatásokról beszélünk ezért többnyire REST alapú szolgáltatásokról van szó. Az ilyen API leírások, mint a fenti egy webszolgáltatás végpontjait, használandó üzenettípusait és biztonsági elvárásokat definiál.

2.2 REST

A REST [8] (Representational State Transfer) egy szoftverarchitektúra típus, elosztott kapcsolatú (loose coupling), internet alapú rendszerek számára. Egy REST típusú architektúra kliensekből és szerverekből áll. A kliensek kéréseket indítanak a szerverek felé; a szerverek kéréseket dolgoznak fel és a megfelelő választ küldik vissza. A kérések és a válaszok erőforrás-reprezentációk szállítása köré épülnek. Az erőforrás lényegében bármilyen koherens és értelmesen címezhető koncepció lehet. Ebből következik az URI (Uniform Resource Identificator) és URL (Uniform Resource Locator) koncepciója. Egy erőforrás-reprezentáció általában egy dokumentum egy URI-val ellátva, mely rögzíti az erőforrás jelenlegi vagy kívánt állapotát. REST-kompatibilis webszolgáltatások állapot mentes operációkkal biztosítják a kérést indító klienseknek az erőforrások elérését és manipulálását.

Egy REST-beli azaz „RESTful” webszolgáltatásnál, egy erőforrás URI-jára tett kérések egy választ bocsát ki, ami XML, HTML, JSON vagy más előre definiált formátum lehet. A válasz megerősítheti, hogy az erőforráson kért manipuláció sikeres volt, vagy nem. A kérés elküldése és a válasz megérkezése között a kliens úgynevezett átmeneti állapotban van. A REST eredetileg a HTTP [9] protokoll keretein belül lett leírva, de nem korlátozódik erre a protokollra. A HTTP protokoll alatt, a rendelkezésre

álló erőforrás manipuláló metódusok vagy ún. „igék” többek között a GET, POST, PUT, DELETE metódusok.

A REST tehát mindenütt jelen van. Mai böngészőink HTTP alapú REST kérésekkel dolgoznak a webtartalom kiszolgáló szerverekkel, és a böngészőnk játssza a kliens szerepet. Ugyanígy amikor a mobilunkon egy webalkalmazást használunk a háttérben HTTP alapú REST kérések történnek, amik lekérik az adatokat, megjelenítik, és kérésekkel frissítik az adatokat a szerveren. Amikor a BME weboldalát szeretnénk meglátogatni, akkor a böngészőnk egy GET kérést indít a www.bme.hu IP cím mögötti szerverhez.

A HTTP REST hátrányai közé tartozik, hogy a kérések szöveges alapúak, tehát nem bináris kódokat használ (mint egyébként az OSI modell többi rétege) ami nagy költséget eredményez az adatküldési sebesség rovására, továbbá az, hogy kapcsolatot kezdeményezni csak a kliens képes.

2.3 OpenAPI

Az idő során egyre több webszolgáltatás született a már kiforrott és megbízható HTTP protokollra és REST architektúrára alapozva. A webszolgáltatások elérhető végpontjait, erőforrásait, metódusait, az üzenettípusok és a metódusok másszóval az API specifikáció leírására több standard formátum született.

Az OpenAPI Specifikáció [10] egy standard, nyelv-független interfészt definiál RESTful API-k leírására, ami lehetővé teszi felhasználók és számítógépek számára egy webszolgáltatás tulajdonságainak felfedezését, a szolgáltatás forráskódja nélkül. Egy ilyen leírás segítségével egy kliens könnyedén tud csatlakozni a szolgáltatáshoz.

Egy jól megírt OpenAPI definíció használható dokumentáció generálásra és forráskód generálásra több nyelven. Jelenleg a standardot a Linux Alapítvány felügyeli. Jól látjuk, hogy ma az OpenAPI specifikáció olyan mint 2000-ben volt a WSDL, bár más üzenetküldési protokollban. A továbbiakban látni fogjuk, hogy az OpenAPI egy tökéletes formátum a REST alapú webszolgáltatások leírására, ezért tehát szeretnénk, hogy egy univerzális szolgáltatás nyilvántartásba ilyen szolgáltatásokat is bejegyezhessünk.

2.4 RPC gRPC

A REST alapú architektúra elosztott hálózaton statikus erőforrások manipulálására, lekérésére lett kitalálva. A webszolgáltatások másik formája az RPC szolgáltatások. Az RPC (Remote Procedure Call) azaz távoli eljáráshívás egy elosztott programozási forma, ahol egy RPC szerver interfésze úgy használható a támogatott programozási nyelvekben, mint egy lokális könyvtár függvényei. Egy hálózaton lévő számítógépen futó program egy másik a hálózaton lévő gépen lévő program függvényét hívhatja meg, és úgy viselkedik mintha egy lokális függvény hívás lenne. Az RPC is kliens-szerver architektúrájú, ahol az eljárás azaz függvény hívó fél a kliens és a végrehajtó a szerver.

Attól függően, hogy milyen típusú webszolgáltatást publikálunk, el kell dönteni, hogy RPC alapú vagy REST alapú a szolgáltatásunk. Alapvetően egy RPC protokoll bármilyen adattovábbítási protokoll felett megvalósítható akár HTTP protokoll felett is. Ha böngészőből szeretnénk használni egy RPC protokollt akkor WebSocket alapú implementációt érdemes használni, mint például a WAMP. Az RCP alapú webalkalmazás készítés egyelőre nem elterjedt, ezért az RPC még elterjedésre szoruló technológia.

Az egyik legmodernebb RPC keretrendszer a Google által fejlesztett gRPC [11]. A gRPC segítségével HTTP 2.0 protokoll felett tudunk binárisan kódolt RPC hívásokat kezdeményezni. A gRPC egyik fő tulajdonsága, hogy az úgynevezett „proto” interfész leíró nyelven definiáljuk az RPC szolgáltatások interfészeit, amikből a gRPC fordító képes több nyelvre fordítani kliens és szerver csonkokat, amik interoperábilisak.

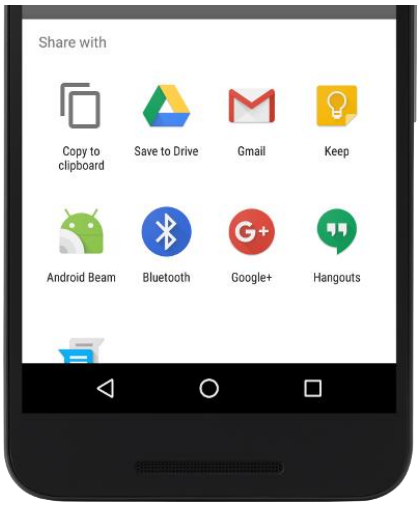
```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

2. ábra: Példa egy egyszerű gRPC szolgáltatás proto definíciójára

A proto definícióban úgy definiálunk RPC függvényeket ahogyan általában deklarálunk függvényeket; a függvény szignatúrája, argumentumok és típusaik és a visszatérési típusok definiálása. A proto definíció egy gRPC szolgáltatás leírására alkalmas, amiből tehát a kliens elegendő információt kap arra, hogy használja a szolgáltatást. Ahhoz, hogy ilyen szolgáltatásokat is be tudjunk regisztrálni egy univerzális szolgáltatás nyilvántartásba szükség van további információra a szolgáltatás elérhetőségét, egyedi nevét, verzióját és a publikáló vállalat információit illetően.

2.5 Android alkalmazások

A mobil eszközök elterjedtével a mobil operációs rendszerek is fejlődtek és velük a mobil alkalmazások is. Napjaink legelterjedtebb operációs rendszere [12], az Android, a szolgáltatás-bróker rendszereket utánozza az alkalmazások közötti feladatmegosztáshoz. Az Androidban jól ismert funkció, hogy alkalmazásból alkalmazásba ugrunk, hogy egy bizonyos dolgot végrehajtsunk. Például, ha egy emailbe egy PDF-fájlt szeretnénk csatolni akkor az email alkalmazás felajánlja a lehetséges PDF fájl kiválasztó alkalmazásokat (lokális fájlböngésző, Google Drive, Dropbox, stb.), majd az operációs rendszer elirányít minket a kiválasztott alkalmazásba, és a kiválasztott fájlal irányít vissza az email alkalmazásba, ahol már csatolva látjuk a fájlt.



3. ábra: Androidban egy megosztás során megjelenő alkalmazás kiválasztó ablak

Ez az Android funkció az úgynevezett Intent resolution, magyarul szándékfeldolgozás. Az alkalmazások szándékokat küldenek az operációs rendszernek

és az operációs rendszer a szándék tartalma alapján választja ki a legmegfelelőbb alkalmazást, ami képes a szándékot végrehajtani.

A natív alkalmazások közötti késői csatolású adatközlési kommunikációs formát IPC-nek hívjuk (Inter Process Communication), és Androidban az Intent Resolution során ilyen kommunikáció történik. Android alkalmazások megírásakor nem tudjuk milyen alkalmazások lesznek elérhetők a rendszeren ezért nem is tudunk egy másik alkalmazást megcímezni. Ehelyett az az ötlet született, hogy szándék szűrőkkel és szándék objektumokkal fogja az Android operációs rendszer futásiidőben összekötni az alkalmazásokat. A kommunikáció csakis az operációs rendszeren keresztül történhet. Ez a módszer nagyon elegánsnak bizonyult, fejlesztése egyszerű és – tapasztalhatjuk – hogy mindennapjainkat elősegíti. Az Android egy Java alapú platform, ezért adott volt, hogy az IPC során küldött objektumok is Java binárisok lesznek, amikhez nem kell különösebb konverzió.

A szolgáltatás integráló rendszerünk készítésekor az Android által használt megoldásból fogunk ötletet meríteni, ezért elemezzük részletesebben az Android alkalmazás összekötő rendszert.

Három része van ennek a kommunikációs rendszernek: az alkalmazások maguk és az általuk regisztrált szándék szűrők, azaz az Intent Filter-ek, a szándék objektum, az Intent, és az operációs rendszerben lévő szoftver modul, ami végrehajtja a szűrést és csatolja az alkalmazásokat. Ez a mechanizmus a routing vagy a publish-subscirbe mintára hasonlít.

2.5.1 Android alkalmazás részei

Androidban a komponensek az alkalmazások építőkockái. A komponensek különböző belépési pontok, ahonnan a felhasználó beléphet az alkalmazásba. Az Android alapokat csak röviden fogjuk átvenni, csak annyira amire szükségünk van a saját rendszerünk megépítéséhez. Androidban négyféle alkalmazás komponens létezik: Actvitiy, Service, Broadcast reviever és a Content Provider.

Az *activiy* egy olyan egy olyan belépési pont, ahol a felhasználó interakcióra képes, azaz van felhasználói felülete, amit a felhasználó lát és kezelhet.

A *service* egy olyan belépési pont, ami az alkalmazást a háttérben futtatja bármilyen ok miatt. Hosszú folyamatok, hálózati kérések, adatbázis műveletek végrehajtását lehet itt végezni. A *service*-nek nincs felhasználói felülete.

A *broadcast reciever,* azaz elosztott üzenet fogadó, egy olyan komponens, ami eseményekre tud feliratkozni amiket más alkalmazás komponensek küldenek vagy az operációs rendszer küld. Ilyen esemény lehet például az akkumulátor merülése egy lokáció megközelítése, időzítő, naptári esemény közeledése, stb. A broadcast reciever komponensek broadcast eseményekre tudnak feliratkozni amiket több ilyen komponens is megkaphat, tehát a ezzel a komponenssel a jól ismert publish-subscribe [13] programozási mintát lehet megvalósítani.

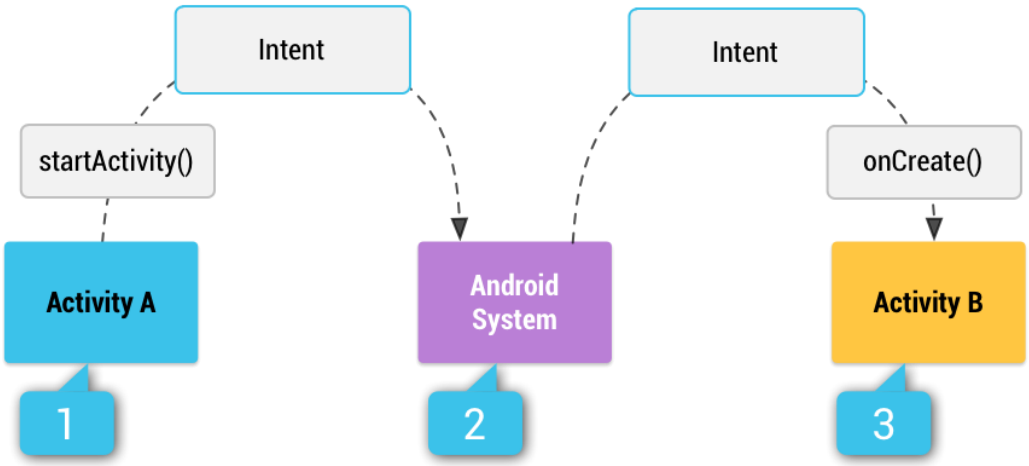
2.5.2 Intent

Az `Intent` [14], azaz „szándék”, egy üzenetküldési típus, amit arra lehet használni, hogy egy *egyszerű* művelet végrehajtását kérjük egy másik alkalmazás komponenstől. Bár az intent-ek az alkalmazás komponensek közötti kommunikáció sok fajtáját segítik elő, három alapvető használati eset van:

- Activity indítása a `startActvity()` rendszerhívás Intent objektum átadásával. Az intent írja le az indítandó activity tulajdonságait és tartalmazza a szükséges adatokat, amiken dolgozni kell. Ha eredményt is várunk az activity-től akkor a `startActivityForResult()` rendszerhívást kell hívni. Az eredmény a hívó activity `onActivityResult()` függvényét fogja meghívni egy új Intent-tel. Az activity felhasználói felülettel rendelkezik.
- Service indítása a `startService()` rendszerhívás Intent objektum átadásával. Az intent írja le az indítandó szolgáltatás tulajdonságait és a szükséges adatokat, amiken dolgozni kell. A service a háttérben fog futni felhasználófelület nélkül.
- Elosztott esemény küldés a `sendBroadcast()` rendszerhívás Intent objektum átadásával. Azok a Broadcast reciever komponensek kapják meg akik feliratkoztak a küldött intent típusára.

Két féle intent típus létezik: **explicit intent** és **implicit intent**. Az explicit intent meghatározza a konkrét célkomponens címét, azaz az osztály nevét, amihez továbbítani kell a szándék végrehajtást.

Az implicit intent nem határoz meg célkomponenst, inkább az operációs rendszerre bízza a megfelelő komponens kiválasztását a rendszeren telepített komponensek közül, amik a megfelelő intent filter-t beregisztrálták. Ha több alkalmazás komponense képes reagálni a szándékra akkor a felhasználó kiválaszthatja, hogy melyik alkalmazást használja. Ez az Intent resolution, azaz szándékfeldolgozás folyamata.



4. ábra: Az implicit intent működése. 1.) Az A activity létrehoz egy Intent-et és meghívja a startActivity()-t. 2.) Az operációs rendszer keres egy egyező intent filter-t, 3.) és az ahhoz tartozó komponenshez küldi a szándék objektumot. [14]

Egy Intent objektum tartalmazza a szükséges információt ahhoz, hogy az Android rendszer meghatározza az indítandó komponenst, plusz az információt, amit a fogadó fog felhasználni, hogy végrehajtsa a kért operációt.

Az Intent a következő adatokat tartalmazza: **Component**, az indítandó komponens egyértelmű azonosítója. **Action**, egy string ami egyértelműen azonosít egy operációt, például: `"android.intent.action.SEND"`. **Data**, adat vagy fájl URI-ja, amin végezni kell az operációt. **Type,** az adat MIME típusa. **Category,** egy string ami további információt ad a célkomponens típusáról. **Extras**, kulcs-érték párok, amiken a célkomponens dolgozhat. **Flags**, az IPC folyamatot befolyásoló jelzők, amik az operációs rendszer számára fontos.

2.5.3 Szándékfeldolgozás folyamata

Ahhoz, hogy egy alkalmazás reagálni tudjon bizonyos szándékokra, szándék szűrőket, azaz Intent filter-eket kell regisztrálni az egyes alkalmazás komponensekhez. Amikor az operációs rendszer egy intent-et kap akkor három tulajdonság alapján keresi az intent-filtereket: Action, Data és Type, Category.

Az intent-filtereket az alkalmazás XML alapú Manifest állományába kell regisztrálni. A Manifest leírja az alkalmazás komponenseit, a komponensek tulajdonságait és az intent-filtereket. Továbbá ebben az állományban kérjük az alap engedélyeket a rendszertől, mint például fájlrendszer olvasása, internet használat, stb. Saját engedélyek deklarálása is itt lehetséges, hogy korlátozzuk egyes komponensek vagy az alkalmazás elérését más harmadik féltől származó alkalmazástól. Az alább látható egy példa egy böngésző activity-re és a szándék szűrőre, ami linkek megnyitását engedi, és a fejlesztői oldalon megtalálható az XML séma definíció [15].

```
<activity android:name=".BrowserActivitiy"
          android:label="@string/app_name">
  <intent-filter>
     <action android:name="android.intent.action.VIEW" />
     <category android:name="android.intent.category.DEFAULT" />
     <data android:scheme="http"/>
  </intent-filter>
</activity>
```

5. ábra: Az activity egy böngésző ablakot jelenít meg, és az intent filter a megnyitási operációk, és a http típusú adatokat szűri, így más alkalmazásból linkeket ezzel a komponenssel lehet megnyitni.

Láthatjuk, hogy az Android egy rugalmas módszert ad az alkalmazások közötti feladatmegosztás megvalósítására. Ez a routing vagy utasító tervezési minta [16], amikor a router az operációs rendszer, és az utasítás az intent objektum tartalma. A szolgáltatás integráló rendszerünk ezt a rendszert fogja használni a webalkalmazások közötti feladatmegosztásra. A következőkben látni fogjuk a webszolgáltatások közötti információ megosztás és hitelesítés ma elterjedt módját.

2.6 OAuth 2.0

Régebben, a webszolgáltatások használatához egy kliens alkalmazásnak a felhasználó azonosítójával és jelszavával kellet hitelesítsék magukat. Ez nagy problémát okozott a felhasználóknak, hiszen amint egy kliens alkalmazásnak megadjuk a jelszavunkat, attól kezdve az alkalmazásnak teljes uralma van a profilunk felett, és a mi nevünkben tesz mindent. Emellett, nem tudjuk másképp visszavonni egy alkalmazás jogosultságát, csak úgy, ha megváltoztatjuk a jelszavunkat, de ekkor minden feljogosított alkalmazástól elvesszük a jogosultságot a profilunk használatához.

A problémára megoldás a Twitter-nél született 2006-ban. Mivel a Twitter akkor már webszolgáltatást kínált harmadikféltől származó webalkalmazásoknak, ezért a

probléma jelen volt. A felhasználók tartottak a kliens alkalmazások használatától, mert nem akarták információbiztonságukat feladni. Viszont emiatt sok hasznos kliens alkalmazást nem használtak.

A Twitter megoldása egy kulcsalapú hitelesítési módszer lett, amit OAuth-nak hívtak. Az OAuth lényege, hogy a jelszó-felhasználónév hitelesítés helyett az alkalmazás egy kulccsal hitelesíti magát. A kulcs megszerzésére viszont a felhasználó ad engedélyt, és a kulcs érvényességét a felhasználó bármikor visszavonhatja. Amíg érvényes a kulcs, addig a szolgáltatásban kezelheti a felhasználó adatai úgy, hogy bemutatja a kulcsot a kérések alkalmával. A felhasználó továbbá korlátozhatja, hogy milyen kérésekhez férhet hozzá a kliens alkalmazás.

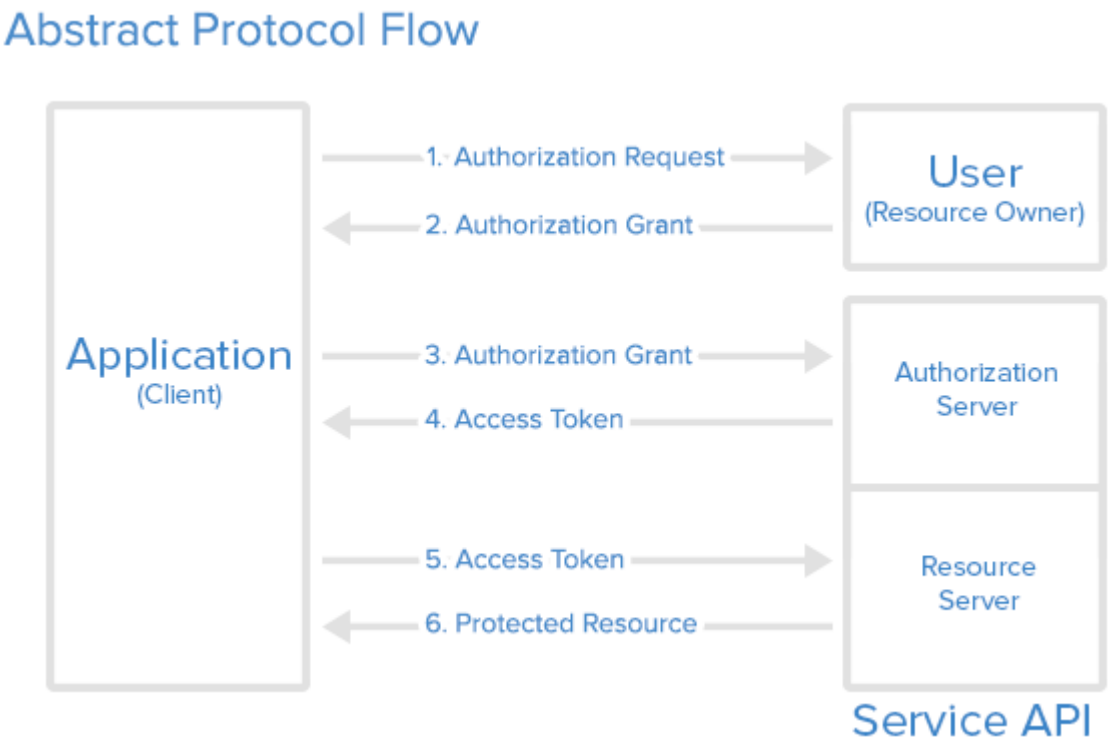
Az OAuth 1.0 sok kétértelműséget okozott, és nem volt teljesen kiforrott. Ma már ipari szinten, de facto használatos az OAuth 2.0 verziója, ami a 6749-es RFC-ben van specifikálva [17], és ez a keretrendszer lett a harmadikfelek hitelesítésének a módszere (harmadik fél alatt olyan személyt vagy alkalmazást értünk, aki a felhasználótól és a szolgáltatástól különbözik).

Az OAuth négy fő szerepet különböztet meg:

- Erőforrás tulajdonos (a felhasználó)
- Erőforrás szerver (az adatokat kínáló szolgáltatás)
- Hitelesítő szerver
- Kliens (egy harmadik féltől származó alkalmazás)

Az erőforrás szerver felel meg a megvédendő szolgáltatás API-jának. Ezen a szerveren érhetők el a felhasználói adatok. A hitelesítési szerver felel a kliensek hitelesítéséért. A gyakorlatban az erőforrás szerver és a hitelesítő szerver egyetlen szerver, és közös API-n érhető el a szolgáltatás és a hitelesítés.

A hitelesítés folyamatát a következő ábra szemlélteti:



6. ábra: Az OAuth 2.0 hitelesítési szekvencia diagramja, forrás: [18]

A hitelesítési folyamat a következőképpen történik:

1. Az alkalmazás (kliens) hitelesítést kér a felhasználótól a szolgáltatás erőforrásainak használatáért.
2. Ha a felhasználó megerősíti a kérést, az alkalmazás egy megerősítési bizonyítványt kap.
3. Az alkalmazás hozzáférési kulcsot kér a hitelesítő szervertől a megerősítési bizonyítvány és az azonosítójának bemutatásával.
4. Ha az azonosítás sikeres és a megerősítési bizonyítvány érvényes, akkor a hitelesítési szerver egy hozzáférési kulcsot ad az alkalmazásnak. Ezzel a meghatalmazás véget ért.
5. Az alkalmazás egy erőforrást kér az erőforrás szervertől a hozzáférési kulcs bemutatásával.
6. Amennyiben a hozzáférési kulcs érvényes, az erőforrás szerver kiszolgálja az alkalmazást az erőforrással.

A gyakorlatban a megerősítés a hitelesítési szerveren keresztül történik. A felhasználó ekkor egy kiugró böngészőablakban erősíti meg hogy a webalkalmazás hozzáférhet az erőforrásokhoz.

Ahhoz, hogy egy alkalmazás azonosítani tudja magát, be kell regisztrálni az a szolgáltatásnál. Ekkor meg kell adni az alkalmazás nevét, weboldalát, és visszahívási URL-t, azaz redirect URL-t, ahova a hitelesítés folyamán a kulcsot megkapjuk. A regisztráció után egy kliens azonosítót és egy titkos kliens kulcsot kapunk.

A hitelesítés során az alkalmazás a szolgáltatás hitelesítési URL-jére küld egy HTTP GET kérést a kliens azonosítójával, egy visszahívási URL-lel, és a kért hatásköröket, azaz scope-okat felsorolva paraméterként. Ekkor egy új ablakban a felhasználó megerősítheti a kérést. A kérés megerősítés után a szolgáltatás a visszahívási URL-t nyitja meg a megerősítési kóddal paraméterként a kérésben. Ekkor a kliens kiveszi a kódot az URL-ből, és így hitelesítve van. Eddig a pontig az úgynevezett implicit hitelesítést történt. Implicit hitelesítéskor a kapott megerősítési kód egyben a hozzáférési kód, és innentől kezdve az alkalmazás jogosult a kapott hatókörben elérhető erőforrásokra.

A másik hitelesítési forma az úgynevezett authorization grant. Ekkor a kliens a megerősítő kóddal és a kliens titokkal kell még egy kódot kérjen a hitelesítési szervertől. Ez egy biztonságosabb hitelesítési mód, hiszen a kulcs nem kerülhet más fél birtokába.

A probléma az OAuth 2.0-val az, hogy nincs egy pontos specifikáció definiálva, ezért különböző vállalatok különböző, nem kompatibilis implementációkat készítenek: *„… this specification is likely to produce a wide range of non-interoperable implementations.„* [19].

Manapság valóban nagy a különbség az implementációk között; más kulcs előállítási algoritmusok, nem egységes hitelesítési folyamat, különböző elnevezésű végpontok, stb. Ennek ellenére az OAuth 2.0 a célját teljesíti, hiszen kiváló hitelesítési módszert kínál a kliens alkalmazásoknak. Ennek köszönhetően az utóbbi években megnőtt a harmadik fél által készített webalkalmazások száma, amik csupán egy másik webalkalmazás szolgáltatásainak bővítésére készült. A mindennap használt Google és Facebook bejelentkezés erre a keretrendszerre épül.

Az egyetlen hátránya az, hogy a különböző hitelesítési implementációk miatt, és a hasonló webszolgáltatások különböző API-ja miatt, egy fejlesztő egy kliens alkalmazást minden célzott webszolgáltatás API-jához kell külön-külön lefejlessze, ahelyett, hogy egy közös OAuth specifikációnak és közös API-knak köszönhetően csak egyszer.

3 AppNet szolgáltatás integráló keretrendszer specifikációja

3.1 Bevezető

A mai webes világból hiányzik egy elterjedt szolgáltatás integráló rendszer, ami webalkalmazásokat és webszolgáltatásokat köt össze dinamikusan. A mai OAuth2 alapú szolgáltatás integrálás csak egy-az-egyhez kapcsolat létrehozására jó, továbbá a webalkalmazások közötti feladatmegosztás, mint ahogyan Androidban működik, egyelőre nem lehetséges. A webszolgáltatásokat tehát két típusra bonthatjuk:

- Gép-gép közötti szolgáltatások a webszolgáltatások.
- Felhasználó-gép közötti szolgáltatások a webalkalmazások.

A célunk az, hogy egy olyan rendszert dolgozzunk ki, ahol a webalkalmazások képesek egy bizonyos feladatot más webalkalmazással végrehajtani és a webszolgáltatások képesek legyenek felderíteni más webszolgáltatásokat különböző típusú szolgáltatás interfész leírók alapján és azokat kapcsolódjanak egymáshoz.

Az alkalmazások feladatmegosztásához az Androidban jól bevált Intent Resolution-hoz hasonló módszert érdemes alkalmazni.

Egy feladat végrehajtását szándékozó alkalmazás nem kell meghatározza melyik alkalmazás hajtsa végre, azt majd az alkalmazás brókerre bízza, ami a felhasználó által engedélyezett/regisztrált alkalmazások közül választja ki a megfelelőt. Ez felel meg az Androidban ismert Implicit/Explicit Intent típusoknak.

Az alkalmazás bróker OAuth2 alapú biztonsági keretrendszert alkalmaz a felhasználó hitelesítéséhez (authentication) és a felhasználó által engedélyezett webalkalmazásoknak a jogosultságaik kezeléséhez (authorization). A felhasználó nem csak telepítéskor hanem futási időben is képes kell legyen jogosultságokat megadni vagy visszavonni.

A szolgáltatás-bróker olyan webszolgáltatásokat kapcsolt össze, amik gép-gép viszonyban vannak. A szolgáltatásokat két komponensre bontjuk: a szolgáltatás interfész leírásra és az interneten elérhető szolgáltatásra, ami egy URL-nek felelhető meg. A szolgáltatás leíró dokumentumok globálisan egyedileg azonosíthatók kell

legyenek. A szolgáltatás leírók egy API leírásnak felelhető meg. Több fajta leíró típust támogatunk köztük REST, gRPC, WSDL, és mások.

A webszolgáltatásokat és webalkalmazásokat ellenőrzött módon lehet egy közös nyilvántartásba regisztrálni, ahonnan a felhasználók csatolni tudják őket a saját alkalmazás hálójukba. Egy alkalmazás háló alatt egy felhasználó által feljogosított olyan webalkalmazások és webszolgáltatások gyűjteményét értjük, amik a hozzáférési jogukkal képesek a többi alkalmazással megosztani a felhasználó általuk képviselt erőforrásait. Emiatt ezt a rendszert Application Network rövidítéseként AppNet-nek hívom.

3.2 Szerepek

Az AppNet négy szerepet különböztet meg:

- Application

Az Application egy webalkalmazás, ami rendelkezik felhasználói felülettel. Egy webalkalmazás egy feladat végrehajtását kérheti egy másik webalkalmazástól a szolgáltatás-bróker segítségével, ha jogosult rá. Egy A webalkalmazás böngészőkből elérhető. A webalkalmazás gondoskodik a felhasználóinak adatairól.

- WebService

A WebService egy webszolgáltatás ami gépi interfésszel rendelkezik. Egy webszolgáltatás egy előre definiált API-n keresztül elérhető. A szolgáltatás hitelesítést követelhet az elérhető API használatához. Egy webszolgáltatás bármilyen protokollt és keretrendszert használhat, nem korlátozott a HTTP protokollra. A webszolgáltatás interfészének leírása kötelező, és az interfész definíció típusa támogatott kell legyen. A webszolgáltatás gondoskodik a felhasználóinak adatairól.

- AppNet Service Broker

Az AppNet szolgáltatás-bróker egy webszerver, ami közösen elérhető minden webszolgáltatásnak és webalkalmazásnak. Annak ellenére, hogy szolgáltatás-bróker a neve webszolgáltatásokat és webalkalmazásokat is kezel egyaránt. Egy alkalmazás vagy szolgáltatás egy felhasználó alkalmazás hálójához való csatlakozást kérheti a szolgálatatás brókertől. A szolgáltatás-bróker nyilvántartja egy felhasználó számára az alkalmazások és webszolgáltatások jogosultságait és engedélyeit.

Egy webalkalmazás egy utasítás elvégzését vagy az utasítás elvégzéséhez alkalmas alkalmazást kérhet a szolgáltatás-brókertől. A webszolgáltatások lekérdezhetnek más webszolgáltatásokat és jogosultságot kérhetnek a használatukhoz a webszolgáltatások kezeléséhez jogosult felhasználótól. A felhasználó a szolgáltatás-brókeren keresztül kezelheti a csatlakozott alkalmazások és szolgáltatások jogosultságait.

- Felhasználó

A felhasználó kezeli a webszolgáltatások és webalkalmazások jogosultságait. Képes új alkalmazásokat és szolgálatatásokat feljogosítani a meglévő hálózatba csatlakozáshoz, és képes visszavonni a már meglévő jogosultságokat és engedélyeket.

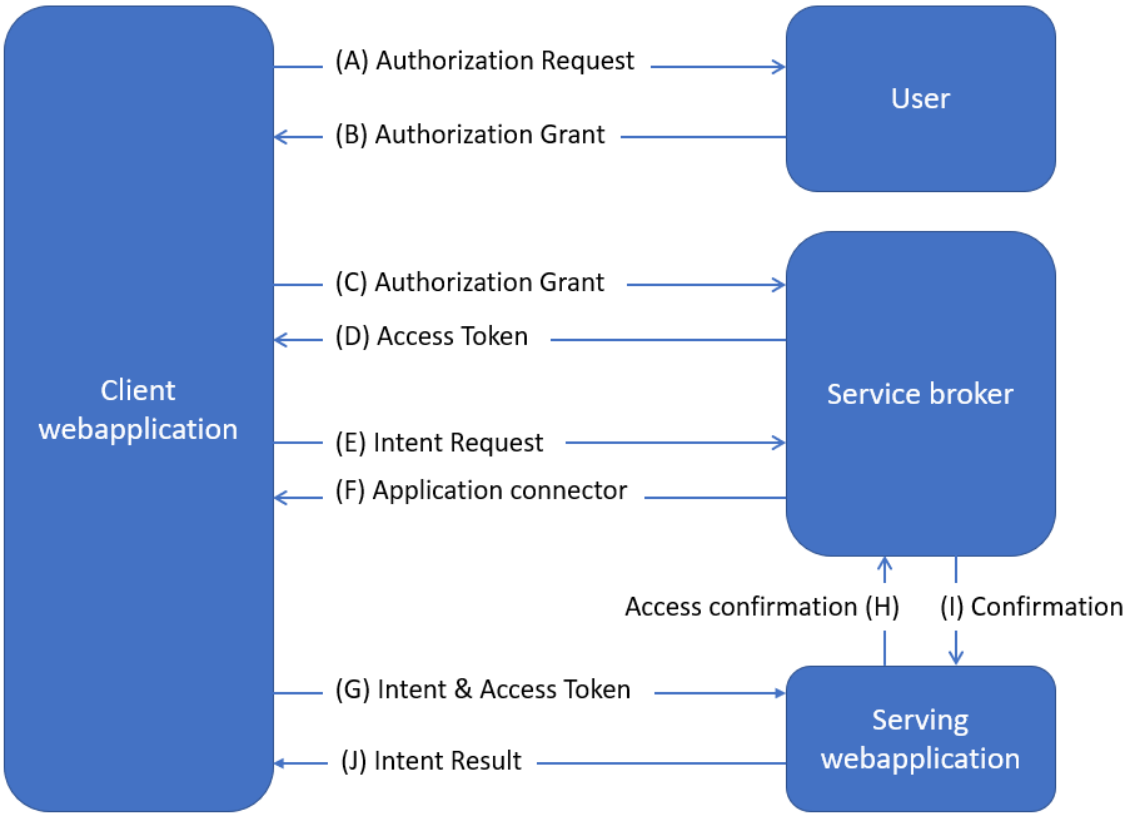
3.3 Feladatmegosztás folyamata

Ahhoz, hogy egy alkalmazás egy szándék végrehajtását, azaz egy utasítást kérjen a szolgáltatás-brókertől vagy egy másik alkalmazástól egy hozzáférési kulccsal kell rendelkezzen, amit csakis kizárólag a szolgáltatás bróker nyújthat. Ehhez, a szolgáltatás bróker az OAuth 2.0 biztonsági keretrendszer hitelesítését használja.

A webalkalmazások feladatmegosztása kétféle módon történhet:

1. Az alkalmazások közvetlenül küldenek egymáshoz utasítást, azaz szándékot, a szolgáltatás-bróker kihagyásával. A szolgáltatás-bróker hitelesíti a kliens alkalmazás hozzáférését a fogadó alkalmazáshoz.
2. Az alkalmazások közvetetten küldenek utasítást egymáshoz a szolgáltatás-brókeren keresztül, és a hitelesítés is a szolgáltatás-brókernél történik.

3.3.1 Közvetlen utasítás küldés



7. ábra: Folyamatábra webalkalmazások közötti közvetlen feladatmegosztásra

A.) A kliens alkalmazás (Client Application) jogosultságot kér (Authorization Request) a felhasználótól az alkalmazás bejegyezéséhez a szolgáltatás-brókerhez, és az engedélyekhez, így használhassa a meglévő alkalmazásokat. Az alkalmazás indirekt a szolgáltatás-brókeren keresztül kéri a felhasználótól a beleegyezést, pl. egy kiugró böngésző ablakban.

B.) A kliens alkalmazás megkapja a megbízást (Authorization Grant), amivel bizonyítja a felhasználó megerősítését.

C.) Az OAuth 2.0 hitelesítési típustól függően, a megbízást elküldi a szolgáltatás-brókernek, hogy egy hozzáférési kódot kapjon

D.) A hozzáférési kulcsot (Access Token) megkapja az alkalmazás, ezentúl az alkalmazás az alkalmazás hálózat része, és kéréseket indíthat a szolgáltatás-brókerhez.

E.) Az alkalmazás egy feladatvégzési szándékot küld a szolgáltatás-brókernek.

F.) A szolgáltatás-bróker ellenőrzi, hogy az alkalmazásnak van-e jogosultsága az utasításhoz, ha igen, akkor egy alkalmazás leírót ad vissza, ami az utasítás elvégzéséhez alkalmas alkalmazást azonosít – ez a szándékfeldolgozás folyamata - és egy hozzáférési kulcsot, amivel jogosult a szándék kérésére a kiszolgáló alkalmazástól (Serving Application).

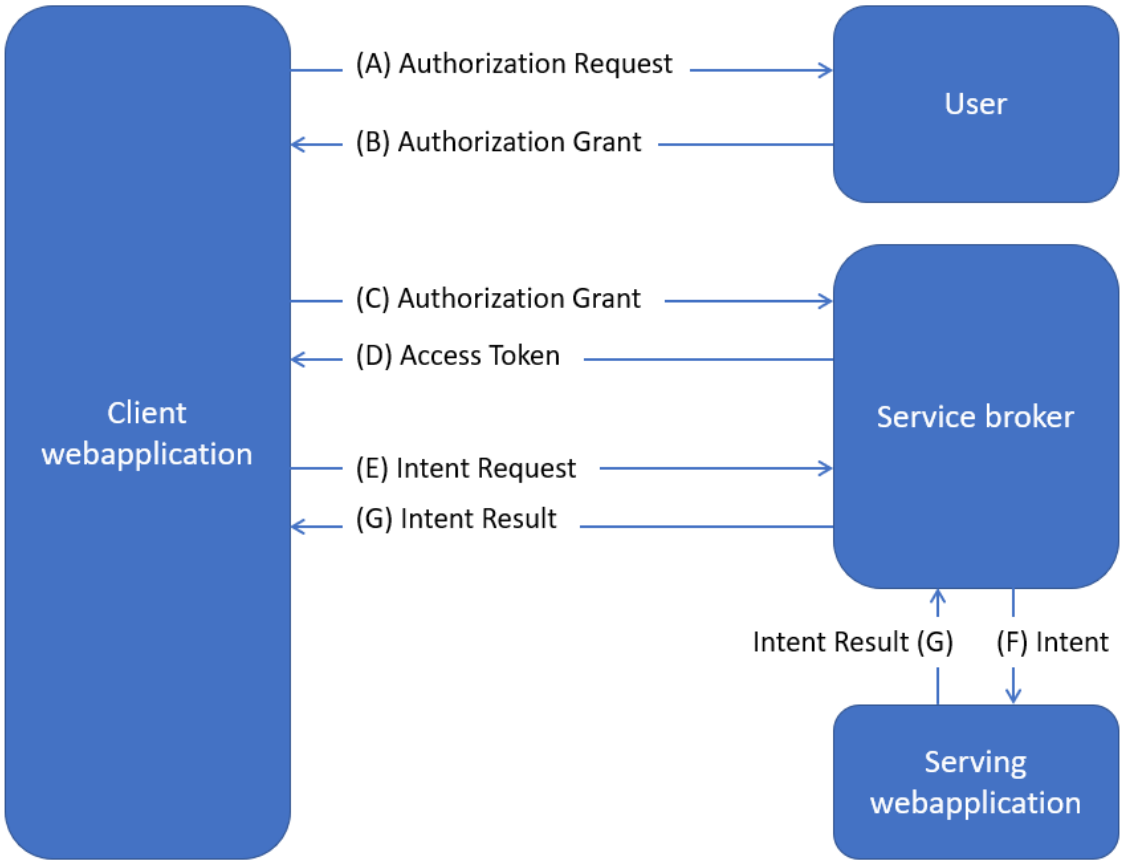
G.) Az alkalmazás elküldi az E.) pontban küldött utasítást és az F.) pontban kapott hozzáférési kulcsot az azonosításhoz

H.) A kiszolgáló alkalmazás (Serving Application) ellenőrzi, hogy a kliens alkalmazás jogosult-e a kéréshez.

I.) A szolgáltatás-bróker megerősíti vagy megcáfolja a kliens alkalmazás jogosultságát. Ha megerősítette, akkor az utasítást végrehajtja a kiszolgáló alkalmazás.

J.) Amennyiben eredményt vár a küldő alkalmazás azt a kiszolgáló alkalmazás visszaküldi.

3.3.2 Követett utasítás küldés



8. ábra: Folyamatábra webalkalmazások közötti közvetett feladatmegosztásra

A közvetett utasításküldés hasonlít a közvetlen utasításküldéshez. A különbség csupán az, hogy az utasítás az AppNet Service Broker-en megy keresztül:

E.) A kliens alkalmazás egy feladatvégzési szándékot küld a szolgáltatás-brókernek.

F.) A szolgáltatás-bróker ellenőrzi, hogy az alkalmazásnak van-e jogosultsága az utasításhoz, ha igen, akkor a szolgáltatás-bróker tovább küldi a megfelelő alkalmazásnak az utasítást

G.) A kiszolgáló alkalmazás azonosítja, hogy a szolgáltatás-bróker küldte az utasítást, így végrehajtja azt. Ha van várt eredmény akkor visszaküldi a szolgáltatás-brókernek, ami továbbítja a küldő alkalmazásnak.

3.3.3 Analízis a két folyamat modell között

A két folyamat különböző szituációkban hasznos. A 1. esetben, közvetlen utasítás küldéskor, a szolgáltatás-bróker feladata csupán az alkalmazás kiválasztása, és a hitelesítés. A hitelesítés az OAuth 2.0 hitelesítési keretrendszer módszerének megfelelően működik.

Ekkor nem kell az utasítást továbbítsa a kiszolgáló alkalmazáshoz, és nem kell az esetleges eredményt a kliens alkalmazáshoz vissza továbbítani. Ez egy elosztott hálózatban egy fontos szempont, hiszen nagyméretű utasítás objektumok szállításakor megnő az adatforgalom.

A másik szempont az üzenetek száma. Látjuk, hogy közvetlen utasítás küldéskor több üzenetváltás történik (közvetlen 10db üzenet, amíg a közvetett csak 8db). Ez annak a következménye, hogy a kliens alkalmazás hitelesítését kettéválasztjuk; a kliens alkalmazás megkap egy hitelesítési kulcsot, de azt a kiszolgáló alkalmazásnak ellenőriznie kell. Az ellenőrzési lépés legalább egyszer szükséges, hiszen ezzel biztosítjuk, hogy a kliens alkalmazásnak valóban van jogosultsága az utasításra. Viszont, ha egy utasítás folyamot küld a kliens alkalmazás akkor nem kell a szolgáltatás-brókernek a hitelesítést megismételni, hanem egy bizonyos ideig érvényesnek hagyhatjuk a hitelesítési kulcsot, és azon ideig a kiszolgáló alkalmazásnak nincs szüksége az ellenőrzésre. Ekkor a két alkalmazás közvetlenül küldhet üzeneteket

egymásnak a hálózaton. Az érvényességi időt bármelyik fél meghatározhatja, és legrövidebb idejű ajánlat lép érvénybe.

Egyszeri utasítások küldésére a 2. eset, a közvetett utasítás küldés esetén egyszerűbb megoldást kínál hiszen az üzenettovábbítást a szolgáltatás-bróker intézi, így nem kell szétválasztani a hitelesítést. Utasítás folyam küldésekor már nem éri meg ezt módszert használni, hiszen minden üzenet egy kerülő utat tesz meg a szolgáltatás-brókerhez, és feleslegesen hitelesítünk újra. Természetesen a szolgáltatás-bróker implementálhat egy cache technikát a hitelesítésre és a célalkalmazás feloldásra.

Abban az esetben, ha kritikus utasításokat kér az alkalmazás, a szolgáltatás-bróker követelheti, hogy közvetetten történjen az utasítás végrehajtás, hogy naplózza az üzeneteket.

3.4 Application - webalkalmazás

A webalkalmazások integrálásához egy webalkalmazás leíró formátumot kell alkalmazni, hogy egységesen lehessen értelmezni az alkalmazások tulajdonságait. Az ilyen leírásokat manifest leírásnak nevezzük. Egy manifest fájl információt tartalmaz egy koherens komponens csoportról. Több technológia használ manifest leírást, pl.: Java-ban a fő osztály kinevezéséhez, vagy Androidban az alkalmazás információinak és komponenseinek leírására. Ahhoz, hogy egy alkalmazás az AppNet keretrendszerben használható kell legyen, a manifest fájlt be kell regisztrálni az AppNet által fenntartott alkalmazás nyilvántartásba. Az AppNet webalkalmazás leíró struktúrája nagyban ihletett az Androidban használt Manifest [15] fájltól.

Egy webalkalmazás három féle komponensből állhat össze, ezeket a manifest fájlban deklaráljuk:

- Activity: egy weboldalnak feleltethető meg, ahol összetartozó funkciókat tudunk használni.
- Service: egy szerver szolgáltatás, amit egy URL-lel tudunk azonosítani. Nincs felhasználó felülete. Erre az URL-re utasításokat küldhetünk, amik a szerver oldalon egy folyamatot vagy változást hoznak létre.
- Reciever: Egy szerveroldali eseményhallgató, ahova eseményeket tudunk küldeni, majd azokra a szerveroldalon reagál, Nincs felhasználói felülete.

Mindhárom komponens utasítás szűrőket regisztrálhat, amivel jelzi, hogy milyen utasítások elvégzésére képes. A manifest fájlban engedélykéréseket jelenthetünk ki, továbbá az alkalmazásunk korlátozására új engedélyeket definiálhatunk.

3.4.1 Manifest fájl szerkezete

Alább látható az AppNet webalkalmazás manifest fájl struktúrája:

```
<?xml version="1.0" encoding="utf-8"?>
<app-manifest xmlns:appnet="http://schemas.appnet.com/appnet"
appnet:versionName="" appnet:versionCode="" >

  <domain appnet:url=""/>
  <callback-url appnet:url=""/>
  <documentation-link appnet:url=""/>
  <uses-permission appnet:name=""/>

  <define-permission appnet:name=""
                     appnet:grantType='["auth code" | "implicit" | ... ]'
                     appnet:label=""
                     appnet:icon=""
                     appnet:premissionGroup=""
                     appnet:description=""
                     appnet:protectionLevel='["normal" | "dangerous" ]'/>

  <define-permission-group appnet:name=""
                           appnet:label=""
                           appnet:icon=""
                           appnet:description=""/>

  <application appnet:name=""
               appnet:icon=""
               appnet:label=""
               appnet:permisison="">

    <activity appnet:url=""
              appnet:name=""
              appnet:icon=""
              appnet:label=""
              appnet:permisison="">

      <intent-filter>
        <action appnet:name=""/>
        <category appnet:name=""/>
        <data appnet:scheme="string"
              appnet:host="string"
              appnet:port="string"
              appnet:path="string"
              appnet:pathPrefix="string"
              appnet:pathPattern="string"
              appnet:mimeType="string"/>
      </intent-filter>
    </activity>

    <service>
      <intent-filter> . . . </intent-filter>
```

```
    </service>

    <receiver>
      <intent-filter> . . . </intent-filter>
    </receiver>

  </application>

</app-manifest>
```

3.4.2 Fontosabb tagek leírása

3.4.2.1 <app-manifest>

A manifest gyökéreleme. Az AppNet névteret és az alkalmazás verzióját definiálja.

3.4.2.2 <domain>

Megadja az alkalmazás doménjét. Minden az alkalmazáson belül definiált entitás egyértelműen azonosítható a doménnel prefixelve. Egy doménhez több alkalmazás is tartozhat.

3.4.2.3 <callback-url>

A visszahívási URL, ahova a hozzáférési kulcsot küldheti a szolgálatatás bróker a hitelesítést követően.

3.4.2.4 <uses-permission>

Engedély/jogosultság kérést jelent ki. A normál szintű engedélyeket elég a manifest-ben kijelenteni, de a veszélyes szintű engedélyeket futási időben kell megadnia a felhasználónak.

3.4.2.5 <define-permission >

Egy új engedély típust definiálhatunk ezzel. Meg kell adjuk az engedély egyértelmű nevét, rövid leírását, OAuth 2.0 alapú hitelesítési típusát, címkéjét és a veszélyességi szintjét. Az engedély típust egy csoport alá lehet rendelni.

3.4.2.6 <define-permission-group>

Egy engedély csoportot definiál, amivel több engedélyt lehet csoportosítani.

3.4.2.7 <intent-filter>

Specifikálja a szándék típust amire a komponens reagálni tud. Ezzel regisztrálja a komponenst a szűrt szándékok végrehajtására. Az elem <action>, <category> és <data> elemeket tartalmazhat, több darabot mindegyikből, amik alapján a szűrés történik.

3.4.2.8 <action>

Egy akciót rendel egy szűrő elemhez. Az akció nevét kell megadni az elemben. Az akció neve egy egyedi sztring. Néhány standard akciót definiálva van az AppNet-en belül, mint például ACTION_SEND, amivel megosztást lehet kérni.

3.4.2.9 <data>

Adat specifikációt ad a szándék szűrőhöz. A specifikáció lehet csupán egy adattípus (a `mimeType` attribútum) vagy lehet még URI mintaszűrőt is megadni. Az adatot azonosítani lehet egy URI-vel, ami többféle URI séma lehet. Az URI részeit az attribútumokkal lehet definiálni az URI részeinek megfelelően [20]:

```
<scheme>://<host>:<port>[<path>|<pathPrefix>|<pathPattern>]
```

Ha a séma nem specifikált, akkor minden más ignorálva lesz. Ha a host nem specifikált akkor a port és minden következő attribútum ignorálva lesz.

Ha több <data> elem van egy <intent-filter> elemben akkor is egy adatszűrőt definiálunk.

Attribútumok leírása:

appnet:scheme

Az URI sémáját jelöli. Például: `http, file, content`. Böngészőben ma már tárolhatunk fájlokat [21] de cross-domain fájlmegosztás nem létezik, tehát webalkalmazások közötti kliens-oldali fájlcsere még nem lehetséges, viszont a Mozilla Developer Network tett rá javaslatot [22]. A jövőben a CORS szabványt fájlmegosztással lehetne kibővíteni ehhez a funkcióhoz. Egyelőre körülményes módon kell megoldani a kliensoldali fájlmegosztás (pl. Cross-Origin Messaging-el [24]).

Amennyiben csak a `mimeType` van megadva akkor a `content:` vagy `file:` sémák feltételezettek.

appnet:mimeType

MIME média típus szűrésére, pl. image/jpeg vagy text/plain.

3.4.2.10 <category>

Alkalmazás kategóriát ad a szűrőhöz. A kategória név egyedi sztring kell legyen. Saját kategóriát is használhatunk, vagy a standard kategóriák közül választunk, pl.: `CATEGORY_APP_CONTACTS, CATEGORY_APP_CALENDAR.`

3.4.2.11 <application>

Az alkalmazás komponenseit tartalmazza. Ikont és címkét lehet hozzárendelni, hogy a felhasználó könnyen azonosíthassa. Engedélyeket lehet hozzárendelni hogy korlátozzuk a hozzáférés más alkalmazásból.

3.4.2.12 <activity>

Egy Activiy-t tudunk vele deklarálni. Több Activiy-je lehet egy alkalmazásnak. Egy activiy attribútumai ugyanazok, mint az <application> tagé, de egy URL-t is meg kell határozni, ahol ez a webalkalmazásnak ezen oldala elérhető.

3.4.2.13 <service>

Egy felhasználói felület nélküli webszolgáltatást deklarál. Az URL az elérési pontja. Ugyanazok a attribútumok vonatkoznak rá, mint az <activity >-re.

3.4.2.14 <reciever>

Egy esemény fogadó deklarálása. Az URL-hez lehet küldeni az utasításokat. Ugyanazok az attribútumok vonatkoznak rá, mint az <activity >-re.

3.4.3 Az alkalmazás regisztrációja

Ahhoz, hogy egy alkalmazást használni tudjunk feladatmegosztásra más általunk feljogosított alkalmazásokkal, az alkalmazást először regisztrálni kell egy nyilvános AppNet webalkalmazás nyilvántartásba. Ez hasonló módon történhet, mint a mai alkalmazás nyilvántartásokba, mint pl. a Google Play Store, vagy App Store. Emlékezzünk vissza, hogy az UDDI-ben szükség volt a fejlesztő fél személyazonosítása és hitelesítése. Egy webes felhasználói felületen erre is lehetőséget tudunk adni.

Mivel OAuth 2.0 alapú hitelesítést használ a Servive Broker a kérések hitelesítéséhez, ezért **a regisztráció során kap a webalkalmazás kliens azonosítót és kliens titkot**.

3.4.4 Csatlakozás egy felhasználó alkalmazás hálójához

Amikor a felhasználó egy AppNet keretrendszert támogató webalkalmazást látogat, a webalkalmazás jogosultságot kérhet a felhasználó alkalmazás hálójába való csatlakozáshoz. A csatlakozás az OAuth 2.0 hitelesítés folyamatát követi. A webalkalmazás a felhasználót átirányítja a szolgáltatás-bróker oldalára továbbítva az alkalmazás azonosítóját, ahol a felhasználó hitelesíti magát. A hitelesítés után a felhasználó dönthet, hogy csatlakoztatja az alkalmazást. Ha a felhasználó feljogosítja az alkalmazást, akkor a szolgáltatás-bróker egy hozzáférési kulcsot ad vissza az alkalmazásnak az előre meghatározott visszahívási URL-en. Ezzel a kulccsal tud később az alkalmazás kéréseket indítani a szolgáltatás-brókerhez.

3.4.5 Szándék objektum

A szándék azaz Intent objektumok a webszerverek és böngészők között folynak, ezért az Intent egy JSON típusú objektum. Az AppNet intent hasonlít az Android intent-hez [14]. Az Intent szerkezete a következő:

```
{
  "action": "",
  "data": "",
  "mime-type": "",
  "category": [],
  "component": "",
  "extras": {},
  "flags": [],
}
```

A `component` elem egy URI ami azonosítja a cél komponenst. Egy komponens URI-ja az őt tartalmazó alkalmazás doménjéből, verziójából és komponens nevéből jön áll. A többi elem hasonlít az Android Intent-ben alkalmazotthoz.

3.4.6 Szándékfeldolgozás

A szándékfeldolgozás során dönti el a szolgáltatás-bróker, hogy mik a szándék fogadására legalkalmasabb komponensek. Ekkor a bróker szerver három tulajdonság alapján hasonlítja össze az intent objektumot a regisztrált intent-filter-ekkel. A szűrés során a szándék objektum szándékszűrő teszteken megy át, és ha egy szándékszűrő tesztjeire sikeresek a tesztek akkor egy alkalmas komponenst találtunk.

3.4.6.1 Akció teszt

Ha az intent-ben megadott action egyezik az intent-filter legalább egyik action nevével akkor sikeres egy teszt. Ha a filter legalább egy action elemet tartalmaz, de az intent nem, akkor is sikeres a teszt.

3.4.6.2 Kategória teszt

Ahhoz, hogy egy intent átmenjen a kategória teszten, az intent-ben definiált minden kategóriának az intent-filterben kell lennie. Az ellenkező nem feltétlenül – egy intent-filter több kategóriát deklarálhat, mint amennyi egy intent-ben hogy sikeres legyen. Tehát egy kategória nélküli intent mindig átmegy ezen a teszten.

3.4.6.3 Data teszt

Az adat teszttelése szigorúbb szabályok szerint történik. Ekkor az intent `data` és `mime-type` elemével dolgozunk. Amennyiben sem az intent sem a filter nem tartalmaz adat leírót, akkor az intent átmegy a teszten. Az egyéb esetek részletesen definiálva vannak az Android fejlesztői oldalán, és az AppNet Adat tesztje azt a specifikációt követi [14].

3.4.7 Kliens API

A webalkalmazások az AppNet szolgáltatás bróker funkcióit a kliens REST API-n keresztül érik el. A szolgáltatás bróker kizárólag Transport Layer Security (TLS) tanúsítvánnyal [26] rendelkező webalkalmazással kommunikál. A TLS segítségével titkosítottan megy minden HTTP üzenet a szerver és kliens között. Erre azért van szükség, hogy megvédjük a hitelesítési kulcsok titkosságát, és a felhasználói adatok kikerülését egy harmadik, nem biztonságos félhez, hiszen egyébként titkosítás nélkül nyerssszövegként kerül az interneten átküldésre, amit egy MITM támadó megfigyelhet. A szolgáltatás bróker egy OAuth 2.0 keretrendszerre fog épülni, és a legtöbb OAuth 2.0 keretrendszer alapból követeli, hogy TLS tanúsítvánnyal rendelkezzen mindkét fél.

A kliens API-val tudja egy webalkalmazás az AppNet szolgáltatásait használni. A fő funkciók:

- Hitelesítés kérése - Authentication
- Szándékvégrehajtás és -ellenőrzés - Intent Resolution
- Engedélyek kérése – Access Request

Az AppNet szolgáltatás bróker API-ját az OpenAPI 2.0 specifikációt követő SwaggerHub webalkalmazás segítségével készítettem. A SwaggerHub webszolgáltatás [27] segítségével jól-formált OpenAPI definíciókat tudunk készíteni, amikből később kliens- és szerveroldali forráskódot tudunk generálni.

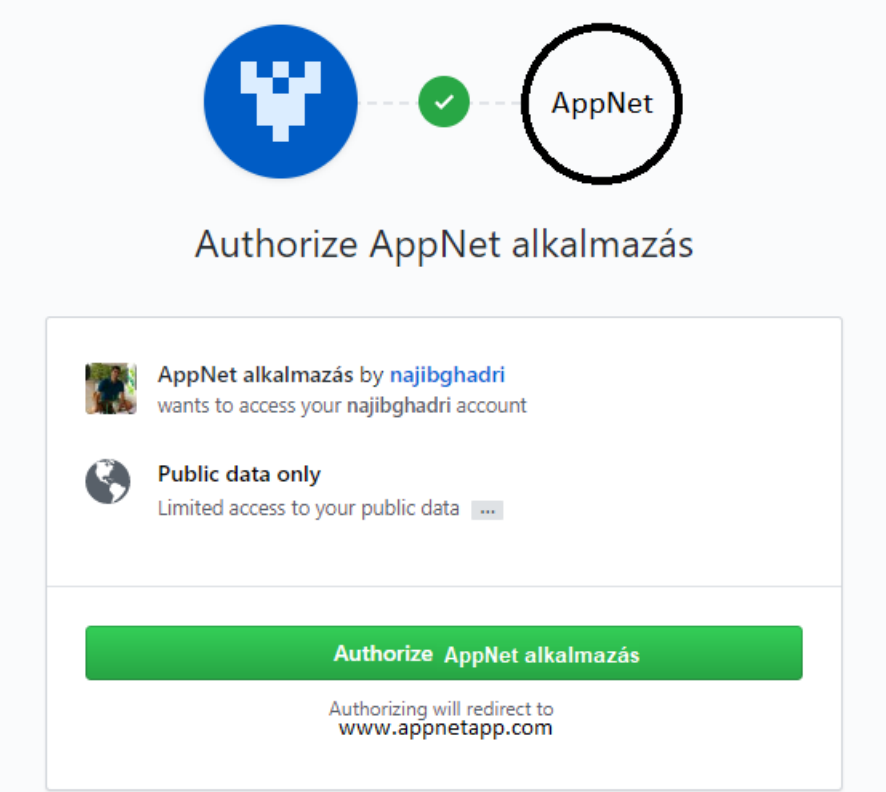
Az általam készített AppNet Service Broker API 1.0 publikus definíció publikusan elérhető (1).

3.4.7.1 Hitelesítési interfész

A hitelesítési interfészen a szolgáltatás brókeren keresztül kérhetünk hitelesítést és egyben engedélyt a `normal` típusú `permission`-ökre. A hitelesítés kérés végpontja:

```
GET /oauth.authorize (1)
```

Ehhez a végponthoz kell küldeni a webalkalmazásnak a kliens azonosítóját, amihez a szolgáltatás bróker a hitelesítési kulcsot fogja rendeli. A kliens azonosítóval tudja az AppNet beazonosítani a webalkalmazást. Mivel az alkalmazás már regisztrálva van ilyenkor ezért az AppNet pontosan tudja, hogy milyen engedélyekre van szüksége az alkalmazásnak. Ekkor a felhasználónak az AppNet megjelenít egy hitelesítési kliens ablakot, ami az Android telepítéshez való hozzájárulási ablaknak felel meg. Az ablak megkérdezi a felhasználót, hogy valóban hitelesíteni szeretné-e az alkalmazást a hálózatba való kapcsolódáshoz.



9. ábra: Hitelesítési ablak példa. Ezt az AppNet szerver jeleníti meg a felhasználónak

A hitelesítés megerősítése után a szolgáltatás bróker kliens ablak visszairányítja a felhasználót a webalkalmazáshoz a hitelesítési kulccsal. A hitelesítési kulcs a hozzáférési kulcsunk is egyben hiszen webalkalmazások során az OAuth 2.0 implicit hitelesítési típust kell használjuk. Az OAuth scope alapértelmezett `application` típusú, amivel a szolgáltatás bróker alapvető funkcióit lehet elérni. A hozzáférési kulcsi megszerzése után használhatjuk az AppNet szolgáltatás bróker szolgáltatásait. Amennyiben viszont nem hiteles a kliens azonosító hibaválaszt kap vissza az alkalmazás.

A hitelesítés kérések a szándékvégrehajtás A.) lépésétől D.) lépésig tartó szekvenciája.

A HTTP kérések hitelesítéséhez az `Authorization` HTTP fejlécben „`Bearer KULCS`” kell legyen az `application/json` média típusú üzeneteknél. URL lekérés alapú kérések esetén egy `token` paraméterben kell megadni a kulcsot.

A következő kérések csakis kizárólag hitelesítve történhetnek. A legtöbb kérés `GET` alapú mivel egy weboldal megjelenítésére szükség lehet, viszont a szándékokra való reagálás csak a szándék objektum tartalmától függ, nem a HTTP kérési metódustól.

Mint látni fogjuk a rendszer fő problémája, hogy a REST architektúrát nem követi, hiszen nem HTTP metódus – URI párosokkal történik a kommunikáció, hanem állapottal rendelkező kliensek objektumokkal való kommunikációjával.

3.4.7.2 Szándékfeldolgozás activity komponenssel

A 3.3-as fejezetben leírtaknak megfelelően kétfajta feladatmegosztási folyamaton mehet végbe a szándék végrehajtása; közvetlen és közvetett szándékvégrehajtás.

A **közvetett szándékvégrehajtás** során az alkalmazás a 3.4.5-ös fejezetben definiált struktúrának megfelelő szándék objektumot küld az AppNet szerverhez. Ez a 3.3.1-as fejezetben leírt közvetlen utasítás folyamat E.) lépésének felel meg.

A szerver a 3.4.6-os fejezetben leírt szándékfeldolgozás specifikációjának megfelelően dolgozza fel a szándékot és választja ki az alkalmas activity komponenst/komponenseket a szándék végrehajtásához. Válaszul egy Connector (1) objektumot küld vissza, ami leírja a kiszolgáló alkalmazás activity célkomponensét, ahova küldeni lehet a szándékot a végrehajtáshoz.

A Connector tartalmazza a kiszolgáló alkalmazás azonosítóját, az activity URL-jét, továbbá megad egy hozzáférési kulcsot, amit csakis kizárólag a küldött szándék a szolgáltatás bróker által kijelölt activity célkomponenssel való végrehajtását hitelesíti és egy időpontot, ami a kulcs érvényességét jelzi. Addig az időpontig nem szükséges újból kulcsot ennek a szándéknak a végrehajtásához. Az AppNet szolgáltatás bróker megjegyzi a kiadott kulcsot és hozzárendeli a kérő alkalmazás azonosítójához, és elmenti.

Abban az esetben, ha a webalkalmazásnak nincs jogosultsága ezt az operációt végrehajtani, akkor hibát kapunk vissza. Ez a közvetlen utasítás folyamat F.) lépésének felel meg. A szándék feldolgozás végpontja:

```
GET /resolve.activity (1)
```

Az üzenettest tartalma az Intent objektum, és válaszul vagy hibát kapunk, vagy egy Connector objektumot, amivel szándékot küldhetünk egy activity-hez. A kérés hitelesítése a fejezetben korábban említett módon történik.

Miután az alkalmazás megkapja a Connector objektumot a benne meghatározott kulcssal és a célkomponens URL-jével, a webalkalmazás elküldheti a szándékot és a kulcsot a célkomponenshez. Ez a közvetlen utasítás folyamat G.) lépésének felel meg.

Elvárt hogy a kiszolgáló alkalmazás a manifest fájljában ígérteknek megfelelően értelmezni tudja a szándék kérést a komponenshez rendelt URL-en. Az Intent objektumot a kiszolgáló activity végpontjára `GET` HTTP kéréssel küldeni, hiszen így lesz egy böngésző ablakban megjelenítve a weboldal, és elérhető a kliens oldalról az Intent tartalma.

Az Intent objektumot a `GET` kérés üzenettestében kell átadni, továbbá a kulcsot a hitelesítéshez az `Authorization` HTTP fejlécben „`Bearer KULCS`” formában.

Amennyiben eredményt is várunk a célkomponenstől, akkor a query-ben `result_redirect_uri` és `request_code` paramétereket kell megadni. A `result_redirect_uri` címen fogja a kliens alkalmazás activity-je fogadni az eredmény intent-et és a `request_code` random számmal fogja azonosítani az állapotot, amivel folytatni kell az activity weboldalt. Erre azért van szükség mert az átirányítások során a kliensoldali állapotok elvesznek, viszont az állapotot el lehet menteni az alkalmazás szerveren vagy a böngészőben, és vissza lehet állítani a `request_code`-dal.

A kérés megérkezésekor a fogadó alkalmazásnak ellenőrizni kell, hogy valóban van jogosultsága a kért szándékra. Ez a közvetlen utasítás folyamat H.) lépése. Ekkor az AppNet szolgáltatás ellenőrző szolgáltatását kell használni:

```
GET /apps.permission.confirm (1)
```

Paraméterként a query-ben a kliens alkalmazás által küldött kulcsot küldi el, természetesen a saját kulcsával hitelesítve a kérést. A kérésre két válasz érkezhet:

- 400: hiba: a kérő alkalmazás nem jogosult a szándékelvégzés kérésére.
- 200: AccessConfirmResponse (1)

Az AccessConfirmResponse leírja a kulcshoz milyen jogosultságok tartoznak. Ha a kiszolgáló alkalmazás manifest fájlja alapján az engedélyek megegyeznek a kulcshoz tartozó kérésekkel akkor kiszolgálhatja az alkalmazás a kérést. A válasz a közvetlen utasítás folyamat I.) lépése.

Siker esetén a kiszolgáló webalkalmazás egy új webablakban jelenik meg. Implementálástól függően akár teljesen vagy részlegesen a szándék feldolgozható a kiszolgáló alkalmazás szerveroldalán, vagy a kliens ablakban.

Az intent feldolgozása a fejlesztők dolga, hiszen ez az alkalmazás törzsében történik. Androidban az alkalmazások megírására a keretrendszer fejlesztési metodológiát biztosít. Ez az Android kifejlett SDK-jának (Software Development Kit) köszönhető. Ahhoz, hogy AppNet-et támogató webalkalmazások fejlesztése egyszerűbb legyen egy AppNet SDK szükségeltetik. Egy lehetséges SDK-ról a következő alfejezet beszél.

Ha a szándékvégrehajtásnak eredménye is van akkor azt vissza kell küldeni a klienswebalkalmazásnak. Az eredményt is egy szándék objektumban küldi vissza a feldolgozó komponens a `result_redirect_uri` címre egy `GET` kéréssel, a szándék objektumot az üzenettestben, és query paraméterként a `request_code`-ot és a szándék végrehajtás eredményét jelző `result_code`-ot, ami az eredmény státuszát jelöli. A `result_code` `RESULT_OK` vagy `RESULT_CANCELED` lehet. Ez a közvetlen utasítás folyamat J.) lépése.

A **közvetett szándékvégrehajtás** során az alkalmazásoknak sokkal kevesebb feladatuk van. Ekkor a kliens alkalmazásnak csupán a szolgáltatás brókernek kell elküldje a szándék objektumot. Egy activity indítása a szolgáltatás brókeren keresztül az alábbi végpontra küldött intent-tel történik.

```
GET /start.activity (1)
```

A bróker a HTTP fejlécben található kulcs alapján azonosítja a kliens alkalmazást. Az azonosítást követően ellenőrzi, hogy az alkalmazásnak valóban van-e jogosultsága a szándék kéréséhez. Amennyiben a kliens alkalmazásnak nincs jogosultsága az operációhoz, vagy az operáció csak futásidőben kérhető el, mert veszélyes szintű, akkor a szolgáltatás bróker egy futásidejű engedélykérési dialógust mutathat a felhasználónak. Ehhez, kérést előnyösebb egy új ablakban indítani, amire a kliens ablaknak JavaScriptben objektum referenciája van. Ekkor a felhasználó dönthet, hogy megadja az engedélyt az operációra, vagy nem.

Androidban a 6.0-ás verzió óta a veszélyes szintű engedélyek csak futásidőben kérhetők el. Ez egy nagyon hasznos funkció, hiszen a felhasználó megválaszthatja külön-külön az engedélyeket, amiket egy alkalmazásnak ad. Továbbá nem kell a

telepítés során a felhasználót összezavarni az alap engedélyek és a veszélyes engedélyek egymásmelletti felsorolásával.

Az AppNet szolgáltatás bróker is egy hasonló dialógussal kérdezheti meg a felhasználótól az engedély megadását. A dialógus ugyanolyan dialógus lehet, mint a 9-as ábra.

Ezek után, ha a kliens alkalmazás jogosultnak bizonyul, akkor a szolgáltatás bróker a szándékszűrés alá veti az intent-et a 3.4.6-os fejezetben írt szándékfeldolgozás szerint. Az szándék feldolgozására több komponens is alkalmas lehet. Ha több komponens alkalmas a szándék feldolgozására akkor az AppNet szerver egy választási ablakot mutathat, mint Androidban amikor egy bizonyos feladatot több alkalmazással hajthatunk végre. Erre egy egyszerű webes dialógust el tudunk képzelni.

A célalkalmazás kiválasztása után a szolgáltatás bróker egy új ablakban nyitja meg az activity-t a végpontjára tett GET kéréssel a korábban specifikált módon. A kiszolgáló alkalmazás a kérés hitelesítéséhez megvizsgálja a HTTP `Authorization` fejléc tartalmát. Mivel a kérés magától az AppNet-től érkezett, ezért felségjogot élvez és végrehajtjuk a kérést, mivel tudjuk, hogy az AppNet szerver már minden hitelesítésen átvitte a kliens alkalmazás kérését.

Ahhoz, hogy azonosítsuk, hogy a szolgáltatás brókertől érkezett a kérés két módszert választhatunk: rávesszük a brókert is hogy kulcsot kérjen minden webalkalmazástól, vagy választunk egy közös titkos kulcsot. Az utóbbi esetben használhatjuk a kliens titkot, amit regisztráció során megkaptunk. Ez egy működő módszer, viszont mivel a kérés, egy kliens ablakból érkezett, és webböngészőkbe töltött kliens kódban ekkor szövegesen benne kell lennie a kulcsnak, ha valamilyen CSRF gyengepont van bármelyik félen, akkor rossz kezekbe kerülhet a titkos kulcs. Biztonságosabb, ha egy frissülő kulcsot használunk az AppNet által kért kérések hitelesítéséhez. Ez viszont túl nagy komplexitást tesz az AppNet webalkalmazás fejlesztők vállára, ezért, ha csak azt várjuk el, hogy biztonságosan legyen a kliens titok kulcs kezelve akkor elegendő a megoldás.

Ha eredmény várunk a cél activity-től, akkor az alábbi végpontot kell használni:

```
GET /start.activityForResult (1)
```

A korábbiaknak megfelelően kell hitelesíteni magunkat, és a szándék mellett egy átirányítási URI-t és a random `request_code`-ot kell küldeni.

3.4.7.3 Szándékfeldolgozás service komponenssel

A service komponensek szerveroldali folyamatokat reprezentálnak. Nem rendelkeznek felhasználói felülettel, és nem képesek felhasználói felületet felnyitni szerveroldalról egy felhasználónak (ez nyilvánvaló, hiszen HTTP felett dolgozunk, és a csak a böngészők indíthatnak TCP vagy akármilyen kapcsolatot egy szerverrel). Amíg egy activity komponenssel csak egy másik activity komponens léphet kapcsolatba, addig egy service komponenssel egy másik service, vagy akár egy activity komponens is küldhet szándékot.

A service komponensek is szándék objektumokra tudnak reagálni. Szándék objektum küldése egy alapértelemzett service komponenshez az alábbi végponton történik:

```
GET /start.service (1)
```

A szolgáltatás bróker szerver az activity indításának megfelelő módján történik. A szolgáltatás bróker ellenőrzi a jogosultságokat, és ha a kliens jogosult akkor a szándékot továbbítja a service komponenshez. Ekkor a service komponens a kliens titokkal hitelesíti a kérést, és végrehajtja a szerveroldali feladatot.

A service komponensek közvetlen elérésére is lehetőség van, ez ugyanúgy működik, mint az activity közvetlen utasítás kérés esetén, viszont service komponensekkel. Ehhez a végpont az alábbi:

```
GET /resolve.service (1)
```

Ebben az esetben is egy Connector objektummal érünk el egy service komponenst, és a kiszolgáló service az AppNet azonosításának megbukása miatt az AppNet szervertől kérdezi le a kulcs jogosultságát az `/apps.permission.confirm` végponton az előbbieknek megfelelően.

3.4.7.4 Szándék feldolgozás reciever komponenssel

A reciever végpontok eseményekre iratkoznak fel. A korábban írtak szerint tehát a reciever komponensekkel lehet eseményekre feliratkozni, mint a publish-subscribe mintában. Az esemény ebben az esetben is egy intent objektum. Az AppNet szerver miden reciever komponensnek elküldi egyszerre az eseményt, ha feliratkoztak rá.

Egy esemény elküldése az alábbi végponton történik:

```
POST /send.broadcast (1)
```

3.4.7.5 További funkcionalitás

```
GET /apps.permission.request
```

Engedély kérése, egy dialógust nyit meg a felhasználó számára.

```
POST /apps.register.broadcastReciever
DELETE /apps.unregister.broadcastReciever
```

Egy reciever futásidejű regisztrálása vagy törlése.

3.4.8 Javascript SDK

Ahhoz, hogy az AppNet webalkalmazások fejlesztése egyszerű legyen egy JavaScript könyvtárat kell készíteni a fejlesztők számára. Ezzel a könyvtárral a fenti kéréseket és kérés szekvenciákat JavaScript függvényekként lehet meghívni az activity weboldalakból. Az SDK könyvtár nagyban egyszerűsíti a fejlesztést. A megálmodott fejlesztési menet egy Android alkalmazás fejlesztéséhez hasonlít, esetleg egyszerűbb is lehet. A következő függvények nem teljes listája mutatná milyen egyszerű szintre lehet a fenti komplex folyamatot leegyszerűsíteni:

- requestAuth()
- startActivity(intent), startActivityForResult(intent, requestCode)
- startService(intent)
- sendBroadcast(intent)
- requestPremission(permissionName)

A JavaScript további potenciális lehetősége az új Cross-Domain Messaging postMessage() API használata [24]. Ezzel, amint neve sugallja, különböző domainekből származó webablakok között lehet biztonságosan, ellenőrzéssel JavaScript objektumokat küldeni. Ezzel a REST architektúra miatti átirányítás-alapú információ közlést helyettesíteni lehet egyszerű kliens oldali üzenetküldéssel. Az egyik fő egyszerűsítés az Intent objektumok ablak alapú küldése lenne. Mivel az activity-k az Intent-eket egyébként is legtöbbször a kliensoldalon dolgozzák fel, ezért érdemes az Intent-et az activity weboldalára a betöltés után a kliens alkalmazás ablakából a célkomponens ablakába a window.postMessage(message) WebAPI használatával átküldeni. A kulcsok és a hitelesítés maradhat a HTTP kérésekben. A tervezett startActivity() JavaScript függvény az intent-et tartalmazó kérés helyett egy AppNet

szerver által kiszolgált weboldalra küldi az intent-et a postMessage() segítségével, később az a kliens ablak is így küldheti más activity-nek az intent-et, és az az activity az eredményt vissza.

Ez egy előnyben részesített implementáció lehet, hiszen nem sértjük meg a REST architektúrát, és az intent objektumok felesleges szerverhez való és vissza küldését spóroljuk meg.

3.5 WebService

A webszolgáltatások egy teljesen más részét képzik az AppNet webszolgáltatás integráló keretrendszernek. A webszolgáltatások nem felelnek meg a szándék-alapú utasítás rendszernek, hanem a saját maguk által képviselt interfészen képesek kommunikálni. A webszolgáltatások bármilyen protokollt vagy keretrendszert használhatnak, az egyetlen kritérium, hogy egy olyan interfész leíró nyelvben íródjanak, ami elterjedt, így az AppNet keretrendszer képes specializálni a leíró típusra, és ezáltal információt kinyerni a leírásból.

Az AppNet keretrendszer nem vállalja a webszolgáltatások összeköttetését, viszont vállalja az OAuth 2.0 alapú hitelesítésüket, ezáltal a felhasználók képesek a webszolgáltatásokat dinamikus fel- és lecsatolni egymásról.

A legtöbb mai webszolgáltatás REST alapú, és a hitelesítés OAuth 2.0 alapú, és az API leírása történhet OpenAPI használatával. Egy OpenAPI leírás tartalmazza a hitelesítési módszert, az esetleges OAuth scope-okat, és a hitelesítési végpontokat.

A szolgáltatás integráló modul más API típusokat is támogathat:

- gRPC proto leírások
- WSDL leírások
- egyéb új leírási típust.

Az AppNet webszolgáltatás integrálás során API leírásokhoz rendel API implementációkat. Ez a hozzárendelés hasonlít az intent - intent-filter kapcsolathoz. Az alapvető funkcionalitás az, hogy egy webszolgáltatás egy API azonosítójával az AppNet szervertől implementációkat kér. Ezzel a bevezetésben tárgyalt UDDI és Google Discovery Service utódjaként próbálok egy használhatóbb, egyszerűbb, rugalmasabb szolgáltatás felfedezési és szolgálatás integrálási rendszert javasolni.

Az integrálás hatékonysága két dologra épül: az automatizálás mértéke, és a vállalatok egy közös API könyvtár fejlesztése.

Az automatizálást a leírások értelmezésével lehet elérni. Az API leírásokból ki lehet nyerni a hitelesítési módszert, és egy felhasználó által kezelt összes szolgáltatás jogosultságát – bármilyen támogatott szolgáltatások lehetnek – egy közös felületen kezelheti. Ezt a kontroll felületnek hívhatjuk.

3.5.1 Webszolgáltatás leírása

Egy webszolgáltatást is egy manifest-tel tudunk leghatékonyabban leírni:

```
<?xml version="1.0" encoding="utf-8"?>
<webservice-manifest          xmlns:appnet="http://schemas.appnet.com/appnet"
appnet:versionName="" appnet:versionCode="" >

  <implemented-api appnet:url=""/>
  <domain appnet:url=""/>
  <documentation-link appnet:url=""/>

  <auth-definition appnet:url=""/>
  <service-implementation
          appnet:url=""
          appnet:name=""
          appnet:icon=""
          appnet:label="">

  </service-implementation>

</webservice-manifest>
```

3.5.1.1 <webservice-manifest>

A szolgáltatásleíró gyökér eleme, a verziót adja meg.

3.5.1.2 <implementead-api>

Ez az elem adja meg az implementált API globálisan egyedi azonosítóját és egyben elérési útját. Ezzel az URI-val tud egy másik szolgáltatás egy implementációt kérni a szolgáltatás brókertől. Ilyenkor egy implementációt regisztrálunk az API-hoz.

3.5.1.3 <domain>

A vállalat doménje, azért, hogy rendelhetni lehessen a webszolgáltatás egy szervezethez. Így a webszolgáltatások csoportosíthatók.

3.5.1.4 <service-implementation>

A szolgáltatás web címe. Ezen a címen érhető el az implementált API-nak megfelelő futó szolgáltatás. A webcím bármilyen URL séma lehet, nem csak http-re korlátozott.

3.5.1.5 <documentation-link>

További szolgáltatás implementáció és vállalat specifikus információk elérhetősége.

3.5.1.6 <auth-definition>

Az OAuth definíció URL-je. Egy OAuth definíció OpenAPI specifikáció szerint megírt leírás. A leírás legfontosabb tartalma a scope-ok leírása és a jelentésük, továbbá a szolgáltatás által kínált hitelesítési végpontok.

Bármilyen protokollú/keretrendszerű szolgáltatást szeretnénk integrálni az AppNet követelmény az, hogy a szolgáltatás hitelesítése és a kliens szolgáltatások jogosítása HTTP REST alapon működjön. Emiatt bármilyen szolgáltatás más szolgáltatással való hitelesítését a közös kontroll felületből lehet kezelni.

3.5.2 Webszolgáltatás regisztráció

Ahogyan a webalkalmazásokat is regisztráltuk, a webszolgáltatásokat is ugyanazon online felületen lehet regisztrálni a manfiest feltöltésével.

3.5.3 Szolgáltatás felfedezés és összekapcsolás

A szolgáltatások más szolgáltatásokat kérhetnek egy felhasználó szolgáltatás hálózatában. Egy felhasználó szolgáltatás hálózata alatt a felhasználó által hitelesített szolgáltatás-szolgáltatás kapcsolatokat értjük.

A szolgáltatás kérés HTTP REST API-n történik az egységesítés szempontjából. Egy szolgáltatás kérés során egy API azonosítót adunk meg, és implementációk kapunk a szolgáltatás brókertől. A kérés végpontja:

```
GET /resolve.webservice (1)
```

A paraméter egy URL sztring, ahol az API leírás elérhető, és egyben azonosítható. Az AppNet szerver megkeresi a regisztrált implementációkat, és visszaad

egy listát az implementációkról. Egy webszolgáltatás implementáció JSON leírása a következő:

```
WebService : {
  "documentation": "",
  "label": "",
  "version-name": "",
  "version-code": "",
  "auth-definition": "",
  "implementation-url": "",
  "domain": "",
  "icon": "",
}
```

A listából a kliens szolgáltatás kiválasztja a szolgáltatásokat, ahova csatlakozni szeretne, és kérést indít az AppNet szerverhez a hitelesítéshez. Ekkor a szolgáltatás megjelöli a kívánt scope-okat és az AppNet szerver a felhasználótól elkéri a megerősítést. Ha azt megkapta, akkor az AppNet bróker hitelesítési kérést indít a kiválasztott implementációhoz. A kiválasztott implementáció a bróker által kért scope-okra ad hozzáférési kulcsot, amit a bróker továbbít a kliens szolgáltatáshoz.

Minden jogosultság változtatás az AppNet szolgáltatás brókeren keresztül folyik. Az implementációk regisztráláskor – a webalkalmazásokhoz hasonlóan - egy titkos kulcsot kapnak, amivel tudják azonosítani, ha az AppNet bróker tesz kérést. A szolgáltatások megbíznak az AppNet kéréseiben, ezért hitelesítés után végrehajtják őket. A hitelesítési kéréseket az AppNet az auth-definíciók szerint készíti.

Ezzel a rendszerrel egy rugalmas, bővíthető szolgáltatás bróker készíthető. Az egyetlen korlátozás a szolgáltatásokra az OAuth 2.0 hitelesítési interfész implementálása, de ettől függetlenül a szolgáltatások egymással bármilyen protokollt és keretrendszert használhatnak.

3.6 Az AppNet felhasználói funkcionalitása

A felhasználók a webes felületen minden alkalmazás és szolgáltatás kezelést képesek kell legyenek elvégezni. A korábban tárgyalt alkalmazások jogosultságainak a kezelésére és az alkalmazások le- és felcsatolására, továbbá a szolgáltatások közötti jogosultságok kezelésére kell lehetőséget adni egy online felhasználó felületen. Ezeknek a funkcióknak a kezeléséhez egy egyszerű könnyen kezelhető felületet kell biztosítani. Például szolgál az Android alkalmazás kezelő felülete, ahol az alkalmazások

jogosultságait kapcsolóval állíthatjuk és gombnyomásra törölhetünk alkalmazást a rendszerből.

4 Implementáció

Az AppNet Service Broker implementációja sok technológiára kell épüljön. A szerver adatbázist kell használjon, és OAuth 2.0 keretrendszert kell támogasson. A feladatnak legalkalmasabbnak a Java alapú Spring [28] keretrendszert találtam.

A Spring keretrendszer magasszinten támogat adatbázis kezelést és újonnan OAuth 2.0 hitelesítési szerver készítést. Egy ilyen webalkalmazás megírása több időt vett volna igénybe, mint a féléves munkám, és valószínűleg nem lenne lehetséges egyedül. Ezért az implementációt redukáltam a core, azaz mag modulokra, amik a legfontosabb logikát implementálják. A későbbi esetleges implementáció esetén a core könyvtárat más keretrendszerrel is használni lehet.

4.1 Fő modulok

A core-t alapvetően két komponensre lehet bontani:

- Kliens kiszolgáló réteg
- Felhasználói réteg

A kliens kiszolgálói réteg felelős a webalkalmazások és a webszolgáltatás kérések kiszolgálásáért. A felhasználói réteg a webes felület és a felhasználói funkciókért felelős.

4.1.1 Webalkalmazás kiszolgálás

A webalkalmazás kiszolgálás fő logikai komponense a szándék feldolgozó komponens. A szándék szűrőket egy relációs adatbázisba menti a réteg, és a szándék feldolgozás során a három tesztre kerül sor.

4.1.2 Webszolgáltatás kiszolgálás

A webszolgáltatás kiszolgálás során egyszerűen egy hash-map alapú tárolóból tudjuk elérni egy API URL-hez tartozó szolgáltatás implementációkat. A hitelesítési komplexitás az OAuth réteg feladata.

5 Összefoglalás

A keretrendszer egy biztonságos és egységes módszert biztosít a webalkalmazások és webszolgáltatások összekapcsolására. Amennyiben a vállalatok közösen fejlesztenek egy közös API könyvtárat, a kliensek képesek több szolgáltatáshoz kapcsolódni közös interfészekkel.

A keretrendszer számos továbbfejlesztési lehetőséget biztosít. A támogatott API leírókat RDF alapú leírókkal lehet kibővíteni, ami egy ontológia alapján választ ki megfelelő szolgáltatásokat egy tudásalapú kliensnek. Ez egy bővítménye lehet a keretrendszernek.

A mobil és webes technológiák konvergenciája figyelhető meg napjainkban. Ma egy böngésző funkcionalitása felér egy operációs rendszerével [29]. A jövőben elképzelhető egy olyan operációs rendszer, ami ötvözi a webes és mobil technológiákat egy operációs rendszerben.

Az AppNet segítségével a fejlesztők képesek egyszerűen integrálni webszolgáltatásokat bármilyen protokollban. A webalkalmazás fejlesztés SDK-val való támogatása esetén az interaktív webalkalmazások fejlesztése is lehetővé válik. A cél az, hogy minél egyszerűbben lehessen webszolgáltatásokat és webalkalmazásokat összekapcsolni. Lehetséges, hogy nem Intent alapú összeköttetés lesz az alkalmazások között, hiszen sérti a REST architektúrát, és sok változtatást igényel, de valamilyen feladatmegosztási architektúrára szükség lesz a jövőben.

5.1 Használati példák

Egy appnet.action.WEBCONSOLE alapú intent filter esetén egy service komponenssel egy felhő alapú shell-t tudunk készíteni az olyan alkalmazásokkal amik ezt az intent-et feldolgozzák. Ez egy gyors és minimalisztikus interfészt biztosít egyszerű webalkalmazásokhoz.

A Facebook Messenger egy jól ismert webalkalmazás, amiben tudunk képeket beilleszteni a számítógépünkről. Azonban ha egy appnet.action.PICK, type: image/jpeg tartalmú intent-et küldünk az AppNet-hez, és a Google Photos egy olyan activity-t regisztrált ami ezt a szándékot fogadja, akkor direkt tudunk a Google-ből képet küldeni

a Facebook-hoz. Ez egy olyan vállalatok közötti integrációt valósít meg, ami egyelőre nem létezik.

Hasonlóan lehet egy közös felhő tároló API-t definiálni, amit ha a Google Drive, Dropbox és egyéb felhőtárolók támogatnak, akkor egy közös interfésszel tudjuk kezelni egy felhasználó összes felhőalapú fájlját.

Köszönetnyilvánítás

Szeretnék köszönetet nyilvánítani a konzulensemnek, Dr. Simon Balázsnak, aki látott fantáziát az ötletemben, és segített a munkámban a félév során. Szeretném továbbá megköszönni a családomnak és a barátaimnak a támogatást.

Irodalomjegyzék

[1] IBM, Uche Ogbuji, *Using WSDL in SOAP applications*, https://www.ibm.com/developerworks/library/ws-soap/index.html (2000. nov.)

[2] UDDI Specification Community, http://uddi.xml.org/

[3] Microsoft, Chris Lovett, 2000. dec., UDDI: an XML Web Service, https://msdn.microsoft.com/en-us/library/ms950813.aspx

[4] Stackoverflow, *Is the public UDDI movement dead or, was it ever alive?*, 2009, okt., https://stackoverflow.com/questions/1525045/is-the-public-uddi-movement-dead-or-was-it-ever-alive/

[5] Wikipédia: *Service-oriented architecture*, https://en.wikipedia.org/wiki/Service-oriented_architecture (revision 12:49, 15 November 2017)

[6] Wikipédia: *Service-oriented architecture – Patterns*, https://en.wikipedia.org/wiki/Service-oriented_architecture#Patterns (revision 12:49, 15 November 2017)

[7] Google API Discover Service, 2017, https://developers.google.com/discovery/

[8] Wikipédia: *Representational state transfer*, https://en.wikipedia.org/wiki/Representational_state_transfer (revision 11:51, 17 November 2017)

[9] Wikipédia: *HTTP*, https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol (revision 15:57, 20 November 2017)

[10] Github: OAI, OpenAPI Specification, https://github.com/OAI/OpenAPI-Specification (2017)

[11] gRPC website, https://grpc.io/ (2017)

[12] Statista, *Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2017*, https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[13] Wikipédia: *Publish-subscribe pattern*, https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

[14] Android Developers, *Intents and Intent Filters*, https://developer.android.com/guide/components/intents-filters.html

[15] Android Developers, App Manifest, https://developer.android.com/guide/topics/manifest/manifest-intro.html

[16] Wikipédia, *Command pattern*, https://en.wikipedia.org/wiki/Command_pattern, (revision 16:49, 29 November 2017)

[17] Internet Engineering Task Force, Request for Comments: 6749, *The OAuth 2.0 Authorization Framework*, https://tools.ietf.org/html/rfc6749

[18] DigitalOcean, Mitchell Anicas, *An Introduction to OAuth 2*, https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2

[19] Internet Engineering Task Force, Request for Comments: 6749, *The OAuth 2.0 Authorization Framework, 1.8 Interoperability*, https://tools.ietf.org/html/rfc6749#section-1.8

[20] Wikipédia, Uniform Resource Identifier / Syntax, https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Syntax, (revision 22:48, 30 November 2017)

[21] Mozilla Developer Network, *Web Storage API*, https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

[22] Mozilla Developer Network, *Device Storage API*, https://developer.mozilla.org/en-US/docs/Archive/B2G_OS/API/Device_Storage_API

[23] Mozilla Developer Network, *Cross-Origin Resource Sharing*, https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[24] Mozilla Developer Network, *Cross-Origin Messaging*, https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

[25] Mozilla Developer Network, *MIME types*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

[26] Wikipédia, *Transport Layer Security*, https://en.wikipedia.org/wiki/Transport_Layer_Security (revision 14:18, 27 November 2017)

[27] SwaggerHub, https://app.swaggerhub.com/

[28] Spring Framework, https://spring.io/

[29] Mozilla Developer Network, *WebAPI*, https://developer.mozilla.org/en-US/docs/WebAPI

Függelék

(1) SwaggerHub, Ghadri Najib, *AppNet Service Broker API*, https://app.swaggerhub.com/apis/najibghadri/AppNetService/1.0

(2) Github, Ghadri Najib, *AppNet*, https://github.com/najibghadri/AppNet