DEBRECENI EGYETEM INFORMATIKAI KAR

INFORMÁCIÓ TECHNOLÓGIA TANSZÉK

Webszolgáltatások és a Java

Diplomamunka

Debrecen

2009.

Jeszenszky Péter
Egyetemi adjunktus

Salamon Edit Anita
Halász Csaba
Programtervező-
matematikus szak

1 TARTALOMJEGYZÉK

2 BEVEZETÉS

Napjainkban egyre nagyobb jelentőséggel bírnak a webszolgáltatások. Noha az Internet már több mint 40 éves, csupán az utóbbi időben vált divattá a webszolgáltatások dinamikus fejlődéséhez kapcsolódó komolyabb szabványok kidolgozása. De miért kellett erre mostanáig várni, és egyáltalán, hogyan jutottunk el ideáig? Az Internet nem egy újkeletű dolog, már a hidegháború idején felmerült egy olyan decentralizált, elosztott hálózat kialakításának gondolata, amely nincs központhoz kötve, túlélve ezáltal akár egy komolyabb támadást is. Az ARPANET megalkotását követően hamarosan lehetőség nyílt az első, kezdetleges elektronikus levelek küldésére és fogadására, ezután pedig a File Transfer Protocol tette lehetővé tetszőleges állományok átvitelét. A TCP/IP megalkotásával létrejött a ma is ismert Internet alapja, majd Tim Berners-Lee által megjelentek az első HTML oldalak. 1994-ben a W3C megjelenésével egy olyan konzorcium született meg, ami a mai web szerkezetét is erősen meghatározza. A nagyközönség érdeklődése a 90-es évek közepétől vált észlelhetővé, 1996-ban pedig elterjedt az Internet kifejezés, és miután kereskedelmi termékké vált, egymás után jelentek meg az internetszolgáltatók világszerte. Az Internet és a számítógépek robbanásszerű elterjedése mellett természetes igénnyé vált, hogy a cégek és nagyvállalatok kialakítsák saját webes arculatukat, minél több információt szolgáltatva ezzel az ügyfelek részére. Ezt a célt kezdetben statikus HTML oldalak elégítették ki, ezeken keresztül váltak elérhetővé a különböző multimédia alapú dokumentumok. Az emberek hamar felismerték, hogy üzleti tranzakciók legegyszerűbb lebonyolítása ugyancsak az Internet segítségével válhat valóra, így a különböző szolgáltatások elektronikus formát öltöttek. A Hewlett-Packard volt az első olyan szoftvergyártó, ami az e-Speak megalkotásával egy olyan platformot jelentetett meg, amivel a fejlesztők létrehozni és működtetni tudtak a mai webszolgáltatásokhoz hasonló e-szolgáltatásokat és programegységeket. Alapvető igénnyé vált az üzleti szolgáltatások online elérhetősége. Megjelent a webszolgáltatás, mint fogalom, amelynek a mai napig nincs általánosan elfogadott definíciója (hogy a különböző piacvezető cégek milyen módon definiálták a webszolgáltatásokat, tehát az egyes nézetek szerint mit tekinthetünk webszolgáltatásnak, megtudhatjuk a 3. fejezetből). A webszolgáltatások ekkor még egymástól teljesen független, egyedülálló gépeken futottak, egyetlen feladatuk az emberi fogyasztásra szánt információszolgáltatás volt. De a technológiai fejlődés nem szabott gátat egy új

elképzelés megvalósításának, miszerint a következő cél egymással kommunikálni képes programok megalkotása lett, amelyek nem – vagy csak alig – igényelnek emberi közreműködést. Az első ez irányba tett legjelentősebb lépés a W3C konzorcium által megalkotott SOAP (Simple Object Access Protokoll), egy olyan XML alapú üzenetküldő protokoll, amelynek legfőbb célja a programok közötti információcsere megvalósítása volt egy olyan decentralizált környezetben, mint amilyen a World Wide Web. A SOAP az XML üzenetek továbbítását nyelv- és platformfüggetlen módon képes megvalósítani. Ezen üzenetek szintaktikáját a WSDL (Web Services Description Language) határozza meg, amely a webszolgáltatásokat leíró nyelv. Az UDDI (Universal Description, Discovery and Integration) pedig egy általános leírást nyújt a webszolgáltatásokhoz, megkönnyítve ezáltal ezeknek a webszolgáltatásoknak a felkutatását és integrálását. A WSDL és az UDDI szintén XML bázisúak. Az XML adat mozgatása az alkalmazások között az olyan gyakran alkalmazott protokollokkal történik, mint az FTP, HTTP, SMTP, de ezt a rugalmasságot ellensúlyozza a SOAP legfőbb hátránya, miszerint a szabványokhoz ragaszkodva egy egyszerű elgondolás nem feltétlenül jár együtt egyszerű megvalósítással. Viszont 2000-ben Roy Fielding egy teljesen új szemléletet vezetett be a webszolgáltatások világába, ez a REST (REpresentational State Transfer), egy szoftver-architektúra, amely közelebb áll a tisztán üzenetküldéshez (message passing), mint a távoli eljáráshíváshoz (Remote Procedure Call, RPC). A webszolgáltatások fejlődése ezt követően két vonalon folytatódott, megosztva ezzel a fejlesztőket. A REST hívők elavult technológiának tartják a SOAP-ot, mondván hogy az általuk támogatott szoftver-architektúra stílussal lazábban csatolt és sokoldalúbb rendszerkomponensek alkothatók, míg a nagyvállalatok által támogatott SOAP/WS hívei pedig folyamatosan a REST szabványtalanságára hivatkozva támadják azt. A SOAP sikerességét 2003 júniusában fogadta el a nagyközönség, hiszen ekkor az 1.2-es verzió megszületésével megjelent az első olyan változat, amely valóban alaposan tesztelt, és amely teljes mértékben kiegészíti a webes szabványokat. Ezzel szemben a REST szoftver-architektúra stílus még napjainkban is a fejlődés folyamatában van. Diplomamunkánk célja ennek a technológiának a megismerése, ismertetése, és annak a kérdésnek a megválaszolása, hogy vajon a REST tényleg beváltja-e a hozzá fűzött reményeket.

3 A WEBSZOLGÁLTATÁS DEFINÍCIÓJA

3.1 Az IBM meghatározása

„A webszolgáltatás egy olyan felület, amely a hálózaton keresztül, szabványos XML üzenetekkel elérhető műveletek egy csoportját írja le. A webszolgáltatások kielégítik egy adott feladat vagy feladatcsoport igényeit. A webszolgáltatáshoz szabványos, formális XML leírás tartozik, amelyet a szolgáltatás leírásának nevezünk, és ez tartalmazza a szolgáltatás igénybevételéhez szükséges összes részletet, többek között az üzenetek formátumát, az átviteli protokollokat és a szolgáltatás pontos helyét. A felület elrejti a szolgáltatás megvalósításának részleteit, így a szolgáltatás használata független az azt megvalósító hardver- és szoftverfelülettől, illetve az adott megvalósítás megírásához használt programozási nyelvtől. Ez teszi lehetővé, hogy a webszolgáltatásokon alapuló programok közötti kapcsolat igen laza maradhasson, maguk a megvalósítások pedig elemekre épülő, többféle eljárást felhasználó megoldások legyenek. A webszolgáltatásokat használhatjuk önmagukban, illetve más webszolgáltatásokkal együtt is, és összetett üzleti tranzakciókat hajthatunk végre a segítségükkel” [1].

3.2 A Microsoft első meghatározása

„A webszolgáltatás egy alkalmazáslogikai egység, amely adatokat és szolgáltatásokat biztosít más programok számára. A programok mindenütt elérhető webprotokollokon és adatformátumokon (például HTTP, XML és SOAP) keresztül érik el a webszolgáltatást, anélkül, hogy bármit is tudnának az egyes szolgáltatások megvalósításának módjáról. A webszolgáltatás egyesíti a komponens alapú fejlesztés és a Világháló előnyeit, és ez a Microsoft .NET programozási modelljének sarokköve” [1].

3.3 A Microsoft második meghatározása

„A webszolgáltatás egy programozható alkalmazáslogika, amely szabványos internetprotokollokon keresztül érhető el. A webszolgáltatások egyesítik a komponens alapú fejlesztés és a Világháló előnyeit. A webszolgáltatás egy fekete doboz, amelyet anélkül hasznosíthatunk újra, hogy a szolgáltatások konkrét megvalósításával törődnünk kellene. A jelenlegi komponensmegoldásokkal ellentétben a webszolgáltatásokat nem az objektummodellekre jellemző protokollon keresztül érjük el. A webszolgáltatásokat a

mindenütt elérhető webprotokollokon és adatformátumokon keresztül (például HTTP és XML) használjuk. Sőt, a webszolgáltatás felülete kizárólag a szolgáltatás által elfogadott, illetve előállított üzenetekre korlátozódik. A webszolgáltatást felhasználó programok bármilyen rendszeren futhatnak, és tetszőleges programozási nyelven íródhatnak, feltéve, hogy a szolgáltatás felületén leírt üzeneteket használják" [1].

3.4 A SUN meghatározása

„A webszolgáltatások olyan szoftverelemek, amelyeket az alkalmazások spontán módon felkutathatnak, egyesíthetnek és átszervezhetnek annak érdekében, hogy megoldást találjanak a felhasználó problémájára. A webszolgáltatások elsősorban a Java nyelvre és az XML-re támaszkodnak.

A webszolgáltatás egy rendszertől és megvalósítástól független szoftverelem, amelyre igazak a következő állítások:

- Leírható egy szolgáltatásleíró nyelv segítségével.
- Közzétehető egy szolgáltatásjegyzékben.
- Felkutatható a szabványos módszerekkel (akár futásidőben, akár a tervezés során).
- Meghívható egy jól meghatározott programozási felületen keresztül, általában a hálózaton keresztül.
- Összekapcsolható más szolgáltatásokkal" [1].

4 WEBSZOLGÁLTATÁSOK MEGKÜLÖNBÖZTETÉSE [2]

Most rátérünk arra, hogy a különböző webszolgáltatásokat mi szerint tudjuk megkülönböztetni egymástól. A két legfontosabb információ, amit a kliensnek a szerver szolgáltatásának igénybevételéhez mindenképpen tudatnia kell a szerverrel: milyen műveletet szeretne végrehajtani a kliens, és melyik az az adat vagy adathalmaz, amelyen ezt a műveletet alkalmazni szeretné. A RESTful webszolgáltatások a végrehajtandó műveletet HTTP metódussal határozzák meg. Ezen megvalósítás előnye, hogy a HTTP metódusnevek szabványosítottak, és egyértelműen hordozzák a végrehajtandó utasításra vonatkozó információkat. SOAP webszolgáltatások esetén a kérés HTTP metódusa POST, még abban az esetben is, ha GET kérést hajtunk végre. Ennek oka, hogy a HTTP metódusok közül egyedül a POST-nak van törzse, amelybe el tudják helyezni a megfelelő

SOAP üzenetet. Ezek a webszolgáltatások alkalmazás-specifikus metódusneveket használnak a művelet meghatározásához, amelyet általában a SOAP boríték törzsében helyeznek el. Ezen megvalósítás előnye, hogy tetszőleges metódusnevet használhatunk, nincs korlátozás. Hátrányuk, hogy ezek a metódusnevek nem szabványosítottak, ezért a webszolgáltatás kliensének tisztában kell lennie a metódus nevével, a paraméterek típusával és sorrendjével, a metódus visszatérési értékével, és természetesen ismernie kell a szolgáltatás URI-ját. Ezeket az információkat a szolgáltatást leíró WSDL dokumentumból tudhatjuk meg.

Miután a végrehajtandó metódusra vonatkozó információkat átadtuk a szervernek, már csak azt kell megmondanunk, hogy mely adaton hajtsa végre ezt a műveletet. A RESTful webszolgáltatások és a legtöbb weboldal számára is az URI-ban kell megadni ezt az információt. Vegyük szemügyre a 6.7 fejezetben bemutatásra kerülő példakód URI-ját: http://localhost:8182/search?kwd=rest+api. Itt a végrehajtandó művelet egyértelműen egy HTTP GET kérés, a „/search?kwd=rest+api” pedig azt határozza meg, hogy a „rest api” kulcsszavakra végrehajtott keresés eredményhalmazát szeretnénk lekérni. Ezzel ellentétben egy tipikus SOAP webszolgáltatás a SOAP boríték törzsében helyezi el a művelet operandusát képező adatra vonatkozó információkat.

Két általános webszolgáltatás architektúrát különböztethetünk meg: REST erőforrás-orientált és RPC stílus. A következő fejezetben konkrét megvalósításokon keresztül megismerhetjük ezen architektúra stílusok legfőbb jellegzetességeit.

4.1 REST erőforrás-orientált architektúrák [2]

Az ilyen típusú webszolgáltatások jellegzetessége, hogy ötvözik az erőforrás-orientált architektúra és a REST architektúra adta lehetőségeket. A végrehajtandó metódust HTTP metódussal határozzák meg, az adatot pedig, amin ezt a metódust végre akarják hajtani, az URI-ban adják meg.

Néhány példa RESTful erőforrás-orientált webszolgáltatásokra:

- Amazon’s Simple Storage Service (S3) (http://aws.amazon.com/s3).
- A legtöbb Yahoo!’s webszolgáltatás (http://developer.yahoo.com/).
- A legtöbb csak olvasható webszolgáltatás, amelyek nem használnak SOAP-ot.
- Statikus weboldalak.
- Számos webalkalmazás, különösen a csak olvashatóak, mint a keresőmotorok.

4.2 RPC stílusú architektúrák [2]

Egy RPC (Remote Procedure Call), azaz távoli eljáráshívást alkalmazó webszolgáltatás esetén a kliens egy boríték belsejében helyezi el a végrehajtandó metódusra, valamint az adatra vonatkozó információkat, és a szerver egy hasonló borítékot küld majd vissza kliensének. A legnépszerűbb borítékformátum a HTTP, a webszolgáltatások többsége ezt használja. A SOAP egy másik népszerű boríték formátum. Egy SOAP dokumentum HTTP-n keresztüli továbbítása során a SOAP boríték a HTTP borítékba kerül.

A legkézenfekvőbb példa az RPC architektúrára a webszolgáltatásokban használt XML-RPC protokoll. Az XML-RPC egy egyszerű protokoll, amely a távoli eljáráshívást XML üzenetekkel valósítja meg. A kéréseket XML-ben kódolják és HTTP POST útján küldik el. Az XML válaszok a HTTP válasz törzsébe vannak beágyazva. A távoli metódus meghívásához szükséges információk, mint a metódus neve és az argumentumok egy XML dokumentumba kerülnek. Ez az XML dokumentum a HTTP boríték törzsébe kerül. Az XML dokumentum változtatásaitól függ, hogy melyik metódus kerül meghívásra, de a HTTP boríték mindig ugyanaz. Egy XML-RPC szolgáltatás általában egyetlen URI-t tesz közzé (a végpontot), a támogatott HTTP metódus mindig POST, és az XML dokumentumban megadott methodName elem határozza meg, hogy melyik metódus kerül majd meghívásra. Ezzel ellentétben egy RESTful webszolgáltatás több URI-t is közzétehet, amelyeken több HTTP metódushívás kezelését is lehetővé teheti.

Néhány példa RPC stílusú webszolgáltatásokra:

- Minden szolgáltatás, ami XML-RPC protokollt használ.
- Majdnem mindegyik SOAP szolgáltatás.

5 WEBSZOLGÁLTATÁSOK MEGVALÓSÍTÁSA

A különböző technológiájú webszolgáltatások egyszerűbb összehasonlítása érdekében a következő implementációk mind ugyanazt a szolgáltatást nyújtják. A rövidebb terjedelem és a könnyebb áttekinthetőség kedvéért csupán egész számok összeadását és kivonását megvalósító szolgáltatásról van szó. A metódusok két egész típusú paramétert várnak, és a visszatérési értékük típusa is egész lesz.

5.1 XML-RPC webszolgáltatás

A távoli eljáráshívás bemutatásához készítünk egy egyszerű szerver alkalmazást és egy klienst a szervereljárások hívásához. A kommunikáció Java oldali megvalósításához az Apache XML projekt Apache XML-RPC elemét alkalmazzuk (amely elérhető a http://xml.apache.org/xmlrpc címen).

A metódusok, amelyeket távoli metódushívás során hívni fogunk a `Numhandler` osztályban találhatóak. A `Numhandler` osztály semmi olyasmit nem tartalmaz, amely az XML-RPC-re utalna, mindössze két nyilvános metódus található benne, az összeadás és a kivonás műveletét végrehajtó függvények. Azon lépéseket, amelyekkel elérjük, hogy az ezen osztályban implementált metódusaink elérhetőek legyenek XML-RPC-n keresztül, a `Server` osztály valósítja meg.

```
public static void main(String[] args) {
   try {
      WebServer server = new WebServer(7777);
      server.addHandler("calculate", new NumHandler());
   } catch (IOException e) {
      System.out.println("Nem indithato el a szerver: " + e.getMessage());
   }
}
```

Létrehozunk egy `WebServer` objektumot, amely elindítja a beépített szervert a konstruktorában megadott porton. Miután a szerver elindult, a szerver `addHandler` metódusával létrehozzuk a `Numhandler` osztály egy példányát `calculate` néven.

A `Client` osztály a `Server`-t teszteli. Felhasználja az XML-RPC könyvtárat és végrehajtja a távoli metódushívást.

```
public static void main(String args[]){
   XmlRpcClient client = null;
   try {
      client = new XmlRpcClient("http://localhost:7777");
      Vector<Integer> params = new Vector<Integer>();
      params.addElement(5);
      params.addElement(2);
      Object response = client.execute("calculate.add", params);
      int result = Integer.parseInt(response.toString());
      System.out.println(result);
   } catch (MalformedURLException e) {
         e.printStackTrace();
   } catch (XmlRpcException e) {
         e.printStackTrace();
   } catch (IOException e) {
         e.printStackTrace();
   }
}
```

Az `XmlRpcClient` osztály konstruktora paraméterül egy URL-t vár, ezzel azonosítja azt a szervert, amellyel kapcsolatot fog létesíteni. Az Apache féle megvalósításhoz a távoli metódus paramétereit egy `Vector` objektumba kell elhelyezni. Jelen esetben két egész típusú paraméterről van szó. Az `execute` metódus meghívásával a kliens XML-RPC kérést készít ahhoz a szerverhez, amelyet a konstruktorban meghatároztunk. A kérés meghívja az első paraméterben megadott metódust a második paraméterében megadott paraméterekkel. A szerver válaszát az `execute` metódus visszatérési értékével szolgáltatja, amely egy `Object` típusú objektum. A kivonás meghívása hasonló módon történik, csupán a meghívandó metódus nevét kell módosítanunk:

```
Object response = client.execute("calculate.sub", params);
```

5.2 SOAP webszolgáltatás

Ha SOAP szolgáltatásokat akarunk létrehozni, akkor szükségünk lesz egy Java servlet erőforrásra. A Java servlet motor azért szükséges a SOAP szolgáltatás telepítéséhez, mert az Apache SOAP szolgáltatás, amit rpcrouter-nek nevezünk, valójában egy Java servlet, amelyet SOAP kérések fogadására konfiguráltak. Például az Apache SOAP-ot telepíteni

tudjuk az Apache Tomcat alkalmazásszerverre. Ehhez szükségünk lesz a következő állományokra:

http://archive.apache.org/dist/ws/soap/version-2.2/soap-bin-2.2.zip

http://www.xmethods.net/download/quickstart/quickstart.zip

A szolgáltatás oldali kódot a `Service` osztály tartalmazza. Az XML-RPC-hez hasonlóan ez is egy átlagos Java osztály, szintén csak két nyilvános metódust tartalmaz, az összeadás és a kivonás elvégzését megvalósító függvényeket. Nincs szükség SOAP specifikus könyvtár importálására vagy SOAP specifikus interfész implementálására. Csupán a szolgáltatás által támogatott metódusokat kell implementálnunk. Egyetlen megkötés a metódusokra vonatkozóan, hogy a paramétereiknek szerializálhatóaknak kell lenniük. Az általunk választott egész típusú paraméterek megfelelnek ennek az elvárásnak. A következő feladatunk, hogy Tomcat alkalmazás szerveren keresztül elérhetővé tegyük a szolgáltatást. A kliens oldali kódnak már szüksége van az Apache SOAP API-ra.

```
public static void main(String[] args) throws Exception {
   URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");
   Integer first = new Integer(30);
   Integer second = new Integer(25);
   Call call = new Call();
   call.setTargetObjectURI("urn:calcjavaserver");
   call.setMethodName("add");
   call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
   Vector<Parameter> params = new Vector<Parameter>();
   params.addElement(new Parameter("first", Integer.class, first, null));
   params.addElement(new Parameter("second", Integer.class, second, null));
   call.setParams (params);
   Response resp = call.invoke(url, "" );
   if ( resp.generatedFault() ) {
      Fault fault = resp.getFault ();
      System.out.println("Fault Code   = " + fault.getFaultCode());
      System.out.println("Fault String = " + fault.getFaultString());
   }
   else {
      Parameter result = resp.getReturnValue();
      System.out.println(result.getValue());
   }
}
```

Létre kell hozni egy `org.apache.soap.rpc.Call` objektumot, ez fogja tartalmazni a távoli szolgáltatás eléréséhez szükséges paramétereket, mint a szolgáltatás neve és a meghívandó metódus neve. A távoli metódus meghívásához szükséges paramétereket `org.apache.soap.rpc.Paramater` típusú objektumokként kell létrehoznunk. A `Parameter` osztály konstruktora négy argumentumot vár: ezek a paraméter neve, az osztály típusa, a paraméter értéke és a kódolási stílus. Az összes paramétert hozzáadjuk egy `Vector` objektumhoz, majd pedig a híváshoz csatoljuk a `setParams` metóduson keresztül.

Így minden paramétert egy név/érték pár határoz meg. A paraméterek sorrendjének meg kell egyeznie a távoli metódus paramétereinek sorrendjével. Ha a kliens nem megfelelő sorrendben adja meg a paramétereket, akkor a „no signature match” hibaüzenetet fogja kapni. A `Call` objektum beállítása után meghívhatjuk a távoli szolgáltatást az `invoke` metódus segítségével. Ezen metódus két paramétert vár:

Az rpcrouter servlet-hez tartozó URL-t, amely jelenleg http://localhost:8080/soap/servlet/rpcrouter, és a SOAPAction fejrészt, melynek értékét beállíthatjuk üres sztringre vagy null értékre. Az XML-SOAP rpcrouter az utóbbi értéket nem fogja felhasználni, mivel a SOAP szolgáltatás célhelyének a metódus első paraméterében megadott URI-t fogja tekinteni. Az `invoke` metódus egy SOAP kérést generál, amit HTTP POST segítségével elküld a szervernek. A kérés kijelöli a szolgáltatást, a meghívandó metódust és a metódushoz szükséges paramétereket. A Tomcat szerver fogadja a kérést és továbbítja az Apache rpcrouter servletnek. Az rpcrouter veszi a szolgáltatást megtestesítő `Service` típusú objektumot és ezen keresztül meghívja a távoli metódust. A `Service` objektum a megadott paraméterekkel végrehajtja a metódust és elkészíti a visszatérési értéket. Az rpcrouter fogadja ezt az értéket, becsomagolja egy SOAP válaszba és visszaküldi a kliensnek.

Az `invoke` metódus visszatérési értéke egy `org.apache.soap.rpc.Response` típusú objektum. Ez az objektum a SOAP válaszon felül tartalmazza a hibalehetőségek paramétereit is. A meghívott metódus visszatérési értékét a `Response` objektum `GetReturnValue` metódusával kaphatjuk meg, amelynek visszatérési értéke `Parameter` típusú. Hogy a tényleges értéket megkapjuk, már csak meg kell hívnunk ezen objektum `getValue` metódusát. Ha összeadás helyett a kivonás műveletét szeretnénk végrehajtani, akkor csupán a `Call` objektum metódusát kell módosítanunk:

```
call.setMethodName("sub");
```

5.3 RESTful webszolgáltatás

A szolgáltatást a `Server` osztály tartalmazza. RESTful webszolgáltatások esetén a kliensnek és a szervernek is importálnia kell REST specifikus osztályokat. Az egyik lehetőség a Restlet keretrendszer által nyújtott osztályok használata, amelyeket a http://www.restlet.org/downloads/testing címen érhetünk el. A szerverhez tartozó kód:

```
public static void main(String[] args) {
   Restlet addHandler = new Restlet() {
   @Override
   public void handle(Request req, Response res) {
      Integer firstNumber = null, secondNumber = null;
      try {
         firstNumber =
            Integer.parseInt((String)req.getAttributes().get("first"));
         secondNumber =
            Integer.parseInt((String)req.getAttributes().get("second"));
      } catch (Exception e) {
         System.out.println(e.getMessage());
      }
      if (firstNumber != null && secondNumber != null) {
         Integer result = firstNumber + secondNumber;
         StringRepresentation stringRep =
             new StringRepresentation(result.toString());
         res.setEntity(stringRep);
         res.setStatus(Status.SUCCESS_OK);
      } else
         res.setStatus(Status.CLIENT_ERROR_METHOD_NOT_ALLOWED);
      }
   };
   Component component = new Component();
   component.getServers().add(Protocol.HTTP, 7777);
   component.getDefaultHost().attach("/add?number1={first}&number2={second}",
                                     addHandler);
   component.getDefaultHost().attach("/sub?number1={first}&number2={second}",
                                     subHandler);
   try {
      component.start();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

A `Server` osztályban létrehozunk egy `Component` típusú objektumot. Ehhez az objektumhoz hozzáadunk egy HTTP szervert és egy portot (ami jelenleg 7777), amelyen a szerver figyelni fog. A `Component` alapértelmezett hosztjához csatolhatunk különböző URI mintákat az `attach` metódus segítségével. Ez a metódus két paramétert vár, az első paramétere az alapértelmezett hoszt folytatásaként értelmezhető URI minta, például

”/add?number1={first}&number2={second}”, második paramétere pedig egy `Restlet` objektum. Ennek hatására például a http://localhost:7777/add?number1=12&number2=8 cím böngészőbe történő beírásával a szerver a kérést a megadott `Restlet` típusú objektumhoz fogja irányítani, és az ott megadott utasításokat fogja végrehajtani. Az URI minta használatának köszönhetően a kliens dinamikus URI-k használatával érheti el a szerver szolgáltatásait. A {}-ben megadott first és second részek helyére megadhatja az összeadás vagy kivonás operandusait. Ebben a példában a `Restlet` objektum felüldefiniálja a `handle` metódust, amely a kérés kezeléséért felelős. A `handle` metódusnak két paramétere van, egy `Request`, azaz kérés és egy `Response`, azaz válasz objektum. A kérés URI-ban megadott attribútumait egy `Map<String, Object>` típusú objektum tárolja kulcs-érték párokként.

A kérésből a `getAttributes().get(”kulcs”)` metódus segítségével kérhetjük le az értékeket, amelyek az aktuálisan végrehajtandó metódus paraméterei lesznek. Ha a kliens megfelelően adta meg a művelet végrehajtásához szükséges paramétereket, akkor létrehozunk egy `StringRepresentation` típusú objektumot, amely tartalmazni fogja a művelet elvégzésének eredményét. A `Response` objektum `setEntity` metódusának segítségével ez a `StringRepresentation` típusú objektum átkerül majd a klienshez, ami feldolgozhatja a kérés eredményét. A művelet végrehajtásának sikerességéről vagy sikertelenségéről a `Response` állapotának beállításával értesíthetjük a klienst. Ehhez a `setStatus` metódusát kell meghívnunk, amely paraméterében a `Status` osztály valamelyik konstansát várja paraméterül, amely HTTP kóddal határozza meg a válasz állapotát. Sikeres GET kérés esetén például `Status.SUCCESS_OK`. Ha viszont a kliens nem az elvárásoknak megfelelően próbálja igénybe venni a szolgáltatásunkat, akkor a `Status` osztály több nevesített konstansát is használhatjuk a hiba okának jelzésére. Jelen esetben a `CLIENT_ERROR_METHOD_NOT_ALLOWED` -et használjuk, ha a kliens oldalon nem megfelelően lettek megadva a műveletek paraméterei. A kivonás megvalósítására egy hasonló `Restlet` objektumot kell létrehoznunk, amely az összeadásétól mindössze egyetlen sorban különbözik:

```
Integer result = firstNumber - secondNumber;
```

A szolgáltatás meghívását szemléltető példa kódot a `Client` osztály tartalmazza:

```
public static void main(String[] args) {
   String uri = "http://localhost:7777/add?number1=13&number2=10";
   ClientResource clientResource = new ClientResource(uri);
   try {
      clientResource.get();
      if(clientResource.getResponse().getStatus().equals(Status.SUCCESS_OK))
       {
         String stringRep = clientResource.getResponse().getEntityAsText();
         System.out.println(stringRep);
       }
      else System.out.println("Az osszeadas sikertelen");
   } catch (ResourceException e) {
      e.printStackTrace();
   }
}
```

Ebben az osztályban a kérésekhez létrehozunk egy `ClientResource` objektumot, melynek konstruktorában megadjuk a híváshoz szükséges URI-t. A kérés elküldéséhez csak meg kell hívnunk ezen objektum `get` metódusát, amely az URI-ban meghatározott címre egy HTTP GET kérést küld. A `clientResource` nevű objektum ezek után tartalmazza a szerver által visszaadott választ és az esetleges hibákra vonatkozó státuszkódot. Ha ez a kód `Status.SUCCESS_OK`, akkor a szolgáltatás sikeresen végrehajtódott.

Magára a visszatérési értékre a `clientResource.getResponse().getEntityAsText()` metódussal hivatkozhatunk. Ha a kliens az összeadás helyett a kivonást szeretné végrehajtani, akkor nincs más dolga, mint átírni az URI-t a megfelelő paraméterekkel. Például a következő URI alkalmazásával:

```
String uri = "http://localhost:7777/sub?number1=13&number2=10";
```

6 A REST ARCHITEKTÚRÁRÓL

6.1 Mi az a REST? [3]

Roy Fielding vetette lapra először a REST szót PhD disszertációjában. Ő többek között a HTTP protokoll egyik készítője, valamint az Apache Software Foundation társalapító tagja. Ezen disszertáció 5. fejezete a „Rest-Style" vagy másnéven „RESTful" webszolgáltatások alapelveit fekteti le. A REST a Reprezentációs Állapot Átvitel-t (REpresentational State Transfer) képviseli, és noha a központi absztrakció – az erőforrás – nem szerepel a betűszóban, fontos kiemelni a fontosságát, mivel ebben a szoftver-architektúrában a kérések és a válaszok is az erőforrások különböző reprezentációinak átvitele köré épülnek. Egy erőforrás a RESTful értelemben bármi lehet, aminek van egy URI-ja, amelynek segítségével megcímezhető. Mivel az URI-k képviselik az egységes erőforrás azonosítót, az erőforrás és az URI fogalma alatt a REST terminológiában ugyanazt fogjuk érteni. Persze ezek az erőforrások, mint Web alapú információs erőforrások, értelmetlenek, ha nincs legalább egy reprezentációjuk. Ebben a kliens-szerver architektúra stílusban tehát egy RESTful kérés kijelöl egy erőforrást, amely a szerveren található. Sikeres kérés esetén az igénylő tipikusan egy erőforrás reprezentációt kap (nyilván, a szerver a kérésnek megfelelő reprezentációt szolgáltatja a kliensnek). Ahhoz, hogy egy webszolgáltatás ezt az architektúra stílust képviselje, be kell tartania a Roy Fielding által definiált megszorításokat, amelyek a következők:

- A webszolgáltatás a kliens-szerver szoftver architektúrára épül: egyértelműen el lehet választani a szervert a klienstől. A szerver komponenst szolgáltatások halmaza alkotja és feladatköre a figyelés, a kliensoldali kérések fogadása. A kliens komponens pedig (válasz reményében) kéréseket küld a szervernek egy konnektoron keresztül. A szerver vagy visszautasítja, vagy feldolgozza ezt a kérést, és a megfelelő választ visszaküldi a kliensnek.
- A kommunikáció állapot nélküli, ami azt jelenti, hogy a kéréseknek minden a kérés feldolgozásához szükséges információt tartalmazniuk kell, a szerveren nem kerül tárolásra semmilyen ehhez szükséges többletinformáció. Az állapot nélküliség nem egy újkeletű dolog, már a client-stateless-server (CSS) megszületésekor bevezették ezt a megszorítást, a láthatóság (visibility), megbízhatóság (reliability) és skálázhatóság (scalability) növelése érdekében. A láthatóság azért javul, mert a rendszerfelügyelőnek a kérések adatain kívül semmi

mást nem kell megvizsgálnia abból a célból, hogy megállapítsa a kérés jellegét, természetét. A megbízhatóság azért nő, mert az állapot nélküliség megkönnyíti a részleges hibák helyrehozásából eredő feladatok megoldását. A skálázhatóság pedig azért javult, mert nincs szükség kérések közötti állapot tárolására, ezáltal a szerver képes az erőforrások hatékony kezelésére, szükség esetén azokat azonnal fel is szabadíthatja. Továbbá egyszerűsödik az implementáció, mivel a szervernek nem kell a kérések között erőforrásokat menedzselnie.

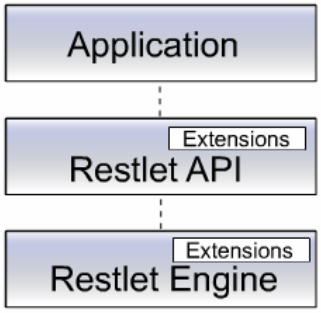
- Gyorsítótárazás lehetősége: A hálózat hatékonyságának növelése érdekében bevezetünk egy olyan cache megszorítást, amely megköveteli, hogy a válaszban elhelyezett adaton belül implicit vagy explicit módon megjelölt címke informáljon az adat gyorsítótárazhatóságáról. Ha a válasz gyorsítótárazható, akkor a kliens gyorsítótára megadja a lehetőséget ezen adat tárolására, hogy később azonos kérés esetén ezt az adatot újra felhasználhassa a kliens. A különböző instrukciók és adatok gyorsítótárban történő tárolása azzal az előnnyel jár, hogy lehetővé teszi néhány interakció teljes vagy részleges kiküszöbölését, ezáltal növelve a hatékonyságot, skálázhatóságot, és a felhasználó által tapasztalt teljesítményt az átlagos várakozási idő minimalizálásával.
- Uniform Interface: Az a legfőbb jellemvonás, ami megkülönbözteti a REST architektúra stílust más hálózati architektúra stílusoktól, hogy különös súlyt helyez a komponensek közötti egységes interfészre. Mivel a szoftvertervezés legfőbb alapelvei a komponens interfészre vonatkoznak, a teljes rendszer architektúra leegyszerűsödik, és az interakciók láthatósága is javul. Az implementációk szétválnak a szolgáltatásoktól, ami független fejlesztésre ösztönöz. Ez azzal a kompromisszummal jár, hogy az egységes interfész csökkenti a hatékonyságot, mióta az információ szállítása inkább egy szabványos formában történik, mintsem az alkalmazás által meghatározott formában. A REST interfész arra lett tervezve hogy hatékony legyen nagy-szemcsézettségű hipermédia adatok továbbítására, a Web gyakoribb eseteire optimalizálva, de ez az interfész nem feltétlenül optimális más típusú architekturális interakciók számára. Ahhoz, hogy egy egységes interfészhez jussunk, összetett architekturális megszorítások bevezetése szükséges a komponens viselkedésében. A REST négy interfész megszorítást definiált, amely a következők:

  - Az erőforrás az azonosítás egysége.
  - Az erőforrás manipulálása reprezentációk kicserélése által történik.
  - Az üzenetkezelés önleíró (self-descriptive) jellegű.
  - Az alkalmazás állapota hipermédia alapokra épül.

- Rétegelt rendszer megközelítés: A kliens nem tudja megmondani, hogy vajon közvetlenül csatlakozik a célszerverhez, vagy egy a kliens és a szerver között elhelyezkedő közvetítő szerverrel áll közvetlen kapcsolatban. A közvetítők növelhetik a skálázhatóságot, például terheléselosztás beiktatásával és osztott gyorsítótárak szolgáltatásával. Egy rétegezett rendszer hierarchikusan szervezett, minden réteg szolgáltatást nyújt a fölötte lévő rétegnek, és az alatta lévő réteg szolgáltatásait használja. A rétegezett rendszerek csökkentik a kapcsolódást az összetett rétegek között azáltal, hogy elrejtik a belső rétegeket a többitől, kivéve a szomszédos külső rétegeket, így növelik a fejleszthetőséget és újrafelhasználhatóságot. A rétegezett-kliens-szerver stílus proxy-t és átjárót (gateway) ad a kliens-szerver stílushoz. Egy proxy egy vagy több kliens komponens között megosztott szerverként működik, fogadja a kéréseket és továbbítja őket esetleges átalakításokkal a szerver komponensek felé. Egy átjáró komponens a szolgáltatásait igénybe vevő kliensek vagy proxy-k számára egy normális szervernek tűnik, de valójában továbbítja a kéréseket esetleges átalakításokkal a belső rétegében található szerver felé. Ezeket a további közvetítő komponenseket összetett rétegekben helyezhetik el, hogy tulajdonságokat adhassanak a réteghez, mint például a terheléselosztás és a rendszer biztonsági ellenőrzése. A rétegezett-kliens-szerver-en alapuló architektúrát az információs rendszerek irodalmában úgy nevezik, hogy két-lépcsős, három-lépcsős vagy több-lépcsős architektúrák. A rétegezett rendszerek elsődleges hátránya, hogy növelik az adatfeldolgozás költségeit és végrehajtási idejét, ezzel csökkentve a felhasználó kiszolgálás teljesítményét.

6.2 Restlet keretrendszer áttekintése

A Restlet keretrendszer két fő részből áll. Az első, a Restlet API, egy szokványos API, ami a REST koncepcióit támogatja és megkönnyíti a hívások kezelését, a kliens- és a szerveroldali alkalmazásokban. A Restlet Engine támogatja ezt az API-t, melynek implementációja megtalálható az `org.restlet.jar` állományban.



6.2.1. ábra – A Restlet keretrendszer

A különbség az API és az implementáció között hasonló a JDBC API és a konkrét JDBC driverek közötti különbséghez.

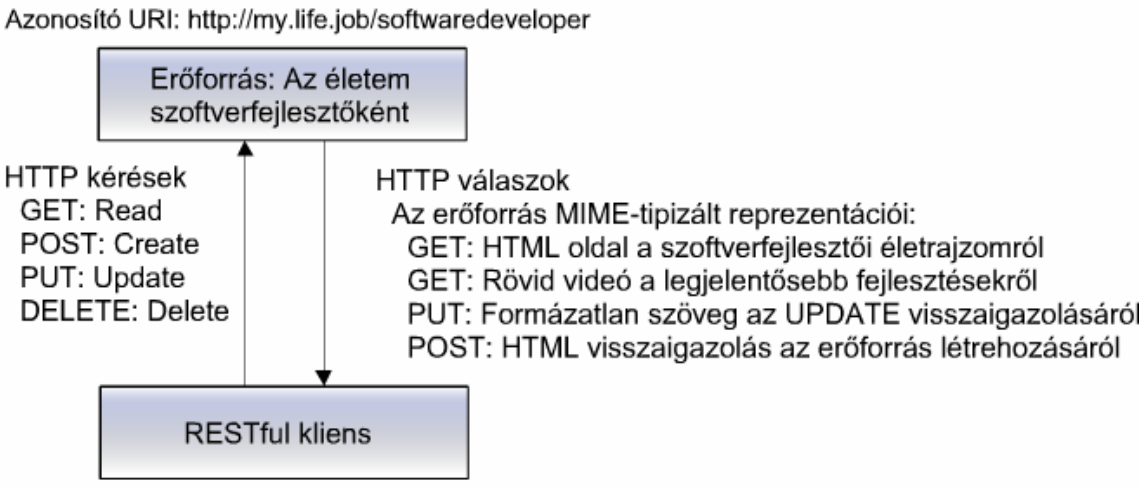
6.3 Műveletek a REST-ben [4]

Látható tehát, hogy a REST és a SOAP meglehetősen különböznek egymástól. Amíg a SOAP egy üzenetközvetítő protokoll, addig a REST a szoftver-arhitechtúra egy stílusa az elosztott hipermédiarendszerekben; vagyis olyan rendszerekben, melyekben szöveget, grafikát, hanganyagot, és egyéb médiákat tárolnak egy hiperhivatkozásokkal összekapcsolt hálózaton keresztül. A World Wide Web egy ilyen rendszer nyilvánvaló példája. A RESTful szemléletmód középpontjában a HTTP áll, ami annak ellenére, hogy magába foglalja a „Transport” szót, nem csupán egy szállítási protokoll, hanem egy API. A HTTP-nek jól ismert igéi (verb) vannak, melyeket hivatalosan metódusoknak, műveleteknek is nevezünk. A következő ábra a HTTP műveleteket ábrázolja, párosítva a megfelelő CRUD (Create, Read, Update, Delete) műveletekkel .

HTTP ige	Jelentése a CRUD terminológiában
POST	Létrehoz egy új erőforrást a kérés alapján.
GET	Erőforrás olvasása
PUT	Erőforrás módosítása a kérés alapján.
DELETE	Erőforrás törlése.

6.3.1. ábra: HTTP igék és CRUD műveletek

Habár a HTTP nem case-sensitive, a HTTP igéket tradicionálisan nagybetűvel írják. További igék is vannak, például a HEAD ige a GET egy variációja, melynek megadása esetén a kliens csak azokra a HTTP-fejlécmezőkre kíváncsi, amiket a kérésben megnevezett objektum letöltésekor a szervertől visszakapna.
A 6.3.2. ábra egy erőforrást (amit a REST terminológiában főnévnek nevezünk) és az azt azonosító URI-t ábrázolja a RESTful klienssel és néhány típusos reprezentációval együtt, amelyet HTTP kérésekre adott válaszokként küldenek el az erőforrás helyett.



6.3.2. ábra – A RESTful kliens kapcsolata az erőforrásokkal

Minden HTTP kérés tartalmaz egy igét, amely jelzi, hogy melyik CRUD műveletet kell majd végrehajtani az erőforráson. Egy jó reprezentáció pontosan illeszkedik a kért műveletre és megfelelő módon rögzíti az erőforrás állapotát. Például ezen az ábrán egy GET kérés visszatérhet szoftverfejlesztői életrajzommal vagy egy HTML dokumentummal, amely egy rövid videó összefoglalót tartalmaz az eddigi fontosabb

hivatkozásaimról. Egy erőforrás tipikus HTML reprezentációja tartalmazhat hiperlinkeket más erőforrásokhoz, ami viszont lehet HTTP kérések célja a megfelelő CRUD igékkel.

6.4 Együttműködése távoli erőforrásokkal [20]

A Restlet keretrendszer jelenleg egy kliens és egy szerver keretrendszer. Például, a Restlet egy saját HTTP kliens konnektor segítségével könnyedén együtt tud működni távoli erőforrásokkal. A REST konnektora egy szoftverelem, amely lehetővé teszi a komponensek közötti kommunikációt valamely hálózati protokoll implementálásával. Ahogy az alábbi ábrán látható, a Restlet nyílt forráskodú projekteken alapuló kliens konnektorok különböző implementációit szolgáltatja.

Server connectors

Extension	Version	Protocols
AsyncWeb	0.8	HTTP
Jetty	6.1	HTTP, HTTPS, AJP
Simple	3.1	HTTP, HTTPS

Client connectors

Extension	Version	Protocols
Apache HTTP Client	3.1	HTTP, HTTPS
Net	1.0	HTTP, HTTPS
JavaMail	1.4	SMTP, SMTPS
JDBC	1.0	JDBC
NRE Local	1.0	CLAP, FILE, WAR

6.4.1. ábra: Szerver- és kliensoldali konnektorok listája

Természetesen a lista folyamatosan bővül, a különböző szerver- és kliensoldali konnektorokról és konfigurációjukról minden információt megtalálhatunk a restlet.org hivatalos weboldalon. Az egyszerűség kedvéért a következő példa egy beépített HTTP konnektor segítségével bemutatja, hogy a Restlet keretrendszer hogyan hallgatja a kliens kéréseit és miként válaszol rájuk. A szerver egy kliensoldalról érkező GET kérés esetén a „hello, world” sztringet adja vissza. A kód szerveren történő futtatásával tehát lehetőségünk nyílik arra, hogy egy webböngésző segítségével a http://localhost:8182 URI-n keresztül megtekintsük a kérés eredményét.

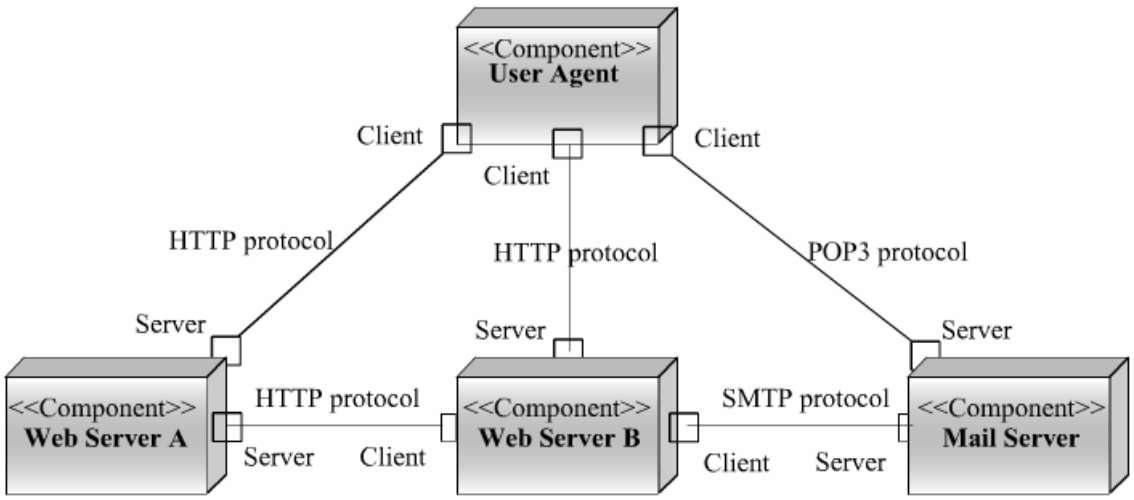
```
public class Sample extends ServerResource {
    public static void main(String[] args) throws Exception {
        new Server(Protocol.HTTP, 8182, Sample.class).start();
    }

    @Get
    public String toString() {
        return "hello, world";
    }
}
```

Tulajdonképpen bármelyik hasonló kezdetű URI működik, így például a http://localhost:8182/test/ is ugyanazt az eredményt szolgáltatja. Megjegyzendő, hogy ha egy másik gépről teszteljük a szervert, akkor ki kell cserélni a „localhost" -ot a szerver IP címére vagy amennyiben definiált, a szerver domain nevére.

6.5 A REST architektúra áttekintése [20]

Vegyük szemügyre a tipikus web architektúrákat a REST szempontjából. A lenti diagrammon a kis négyzetek szemléltetik a konnektorokat, amelyek lehetővé teszik a komponensek közötti kommunikációt. A komponenseket a nagyobb dobozok ábrázolják. A kapcsolatok szemléltetik a részletes protokollokat (HTTP, SMTP, stb), amelyek az aktuális kommunikációhoz szükségesek.



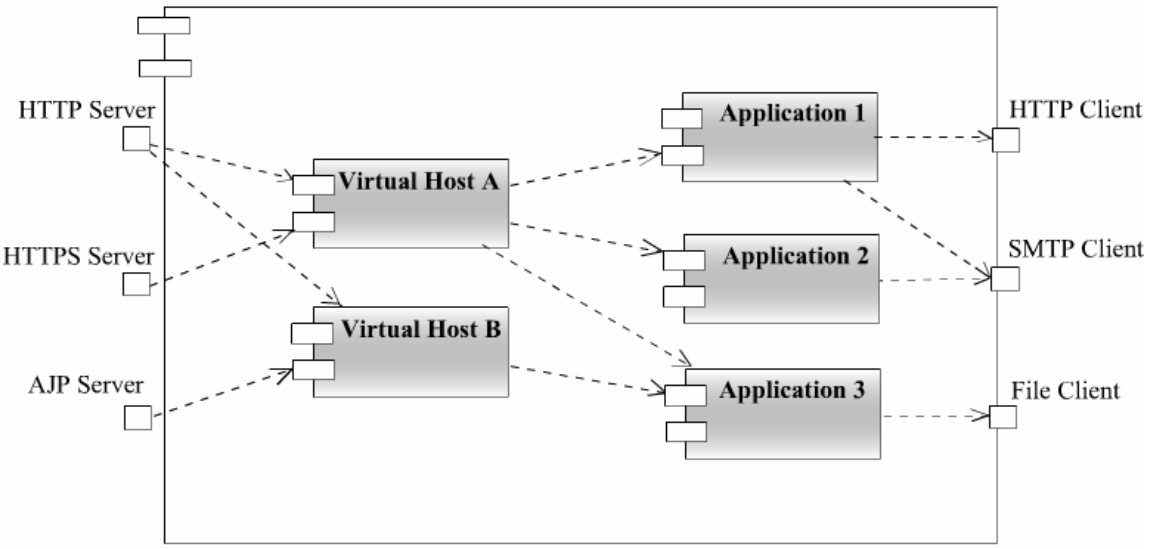
6.5.1. ábra – A komponensek közötti kommunikáció

Ugyanannak a komponensnek több kliens és szerver konnektora is lehet. Példánkban a B webszervernek van egy Server konnektora a User Agent komponenstől érkező kérések

kezeléséhez, és további kliens konnektorai is vannak, amelyek kéréseket küldenek az A webszervernek és a Mail Servernek.

6.6 Komponensek, virtuális hosztok és alkalmazások [20]

A Restlet keretrendszer számos olyan osztályt kínál, amelyek nagymértékben egyszerűsítik összetett alkalmazások fejlesztését. A cél egy RESTful, hordozható és sokkal flexibilisebb alternatívája a Servlet API-nak. A következő diagrammon láthatunk három Restlet típust, amelyek célja ezen komplex esetek kezelése.



6.6.1. ábra – A Restlet komponens felépítése

Ez a Restlet lehetővé teszi különböző konnektorok (Connector), virtuális hosztok (VirtualHost), szolgáltatások (Service) és alkalmazások (Application) halmazának kezelését. Az alkalmazásoktól elvárjuk, hogy közvetlenül kapcsolódjanak virtuális hosztokhoz vagy a belső forgalomirányítóhoz (routerhez). A komponensek különböző szolgáltatásokat nyújtanak: hozzáférés naplózás (acces logging) és állapot beállítás (status setting). Roy T. Fielding szavait idézve a komponens definíciója a következő: „A komponens utasítások és belső állapotok absztrakt egysége, amely az adatok átalakítását szolgáltatja az interfészein keresztül” [3]. Egy komponens konfigurációja program által vezérelt módon (programatikusan) vagy XML dokumentum használatával történhet. Vannak hozzárendelt konstruktorok, amelyek vagy egy XML dokumentumhoz tartozó URI referenciát fogadnak el, vagy egy bizonyos XML dokumentum reprezentációját,

lehetővé téve a támogatott kliens és szerver konnektorok valamint a szolgáltatások listájának egyszerű konfigurációját.

6.7 URI felülírása és átirányítás [20]

Tim Berners-Lee határozta meg a jó URI („cool URI") fogalmát, amelynek pontos definícióját a http://www.w3.org/Provider/Style/URI honlapon találhatjuk. A Restlet keretrendszer egyik előnye, hogy beépített támogatást tartalmaz a jó URI-khoz. Egy Redirector nevű eszköz például lehetővé teszi egy URI felülírását egy új URI-val, miközben automatikusan beállítja az átirányítást. Felmerülhet bennünk a kérdés, hogy miért van szükségünk az átirányításra? A jó URI-k legfontosabb jellemzője, hogy nem változnak. Vannak azonban olyan szituációk, amikor weblapunk átszervezésével, vagy az URI-k alapját képező fájlok áthelyezésével, illetve az URI-k által magukba foglalt információk megváltozásával bizonyos intézkedéseket kell tennünk. A jó URI-t tervezni kell, és figyelembe kell venni, hogy a létrehozás dátuma után bármely más, az URI-ra vonatkozó információ megváltozása a későbbiekben problémához vezethet. Változhat például egy program verziójára vonatkozóan a szerző neve, a dokumentum státusza, elérési szintje („csapat-hozzáférésű", „tag-hozzáférésű", „nyilvános hozzáférésű"), a fájl neve, illetve a kiterjesztése. Az alapvető problémát az okozza, hogy nincs rálátásunk a szerverünkön jelenlévő URI-k módosításának következményeire. A régi URI-t számos weboldal hivatkozhatja, ugyanakkor több ezer web-böngésző könyvjelzője is tárolhatja a szóban forgó hivatkozást. „Ha valaki követ egy hivatkozást és az nem működik, általában megrendül a bizalom a szerver tulajdonosában. Ráadásul érzelmileg is hátrányosan érintheti a látogatót a dolog és célját sem éri el. Ezért nagyon fontos, hogy szükség esetén egy átirányítás segítségével kiküszöböljük újabb holt linkek keletkezését" [5].

A Redirector bemutatása céljából a következő példában definiálunk egy Google-n alapuló kereső szolgáltatást, amely a http://www.restlet.org weboldalon keres tetszőleges kulcsszavak alapján. A „/search" relatív hivatkozás azonosítja a kereső szolgáltatást, amelyhez keresési kulcsszavakat adhatunk meg, amelyeket a „kwd" paraméteren keresztül érhetünk el.

```
public static void main(String args[]) {
   Component component = new Component();
   component.getServers().add(Protocol.HTTP, 8182);
   Application application = new Application() {
      @Override
      public Restlet createRoot() {
         Router router = new Router(getContext());
         String target =
         "http://www.google.com/search?q=site:www.restlet.org+{keywords}";
         Redirector redirector = new Redirector(getContext(), target,
            Redirector.MODE_CLIENT_TEMPORARY);
          router.attach("/search", redirector).extractQuery("keywords",
            "kwd", true);
         return router;
      }
   };
    component.getDefaultHost().attach(application);
    component.start();
}
```

Ezen kód futtatásával a kliens http://localhost:8182/search?kwd=rest+api címre kiadott HTTP GET kérésére a Google által szolgáltatott azon találati oldalt kapjuk meg, amit a „rest api” kulcsszavakra történő keresés eredményezett a www.restlet.org oldalon. A `Redirector` -nak három paramétere van. Az első a szülő kontextus, a második az URI átírását reprezentáló sztring, amely egy URI minta, a harmadik paramétere pedig az átirányítás módját határozza meg, itt választhatjuk ki a kliens átirányítását. Az `attach` metódus által szolgáltatott `Route` típusú objektum `extractQuery` metódusa kinyeri a „kwd” paraméter értéket a kérésből, amíg a hívás az alkalmazáshoz irányítódik. Ha megtalálja a paramétert, akkor átmásolja a kérés „keywords” nevű attribútumába, így a `Redirector` ennek használatával képes lesz elkészíteni a cél URI-t. A `Redirector` használatával tehát készíthetünk saját weboldalunkhoz is egy Google alapú kereső szolgáltatást, csupán a fenti kódban ki kell cserélnünk az átirányítás célját meghatározó URI mintát erre a mintára:

```
String target = "http://www.google.com/search?q=site:mysite.hu+{keywords}";
```

Ennek hatására a klienstől érkező kérést a Google-hez irányítjuk, amely a mysite.hu (amely a saját weboldalunk elérési címe) oldalon fog keresni a keywords listában megadott keresőszavak alapján.

6.8 Hiba oldalak megjelenítése [20]

Lehetőségünk van a státusz oldalak testreszabására, amelyek akkor jelennek meg, ha a hívás kezelése során valami nem az elvárásoknak megfelelően történt, például a hivatkozott erőforrás nem található. Ebben az esetben, illetve ha nem kezelt kivétel váltódik ki, a komponens vagy az alkalmazás egy alapértelmezett státusz oldalt szolgáltat. Ez a szolgáltatás az `org.restlet.util.StatusService` osztályon, illetve a komponens vagy az alkalmazás `statusService` tulajdonságán keresztül érhető el. Annak érdekében, hogy testreszabhassuk az alapértelmezett üzeneteket, egyszerűen csak létre kell hozni a `StatusService` egy alosztályát és felül kell íni a `getRepresentation(Status, Request, Response)` metódust. Ezt követően be kell állítani a testreszabott szolgáltatás példányának a `"statusService"` tulajdonságát.

6.9 Statikus fájlok szolgáltatása [20]

Ha a webszolgáltatásunknak van olyan része, amely statikus weboldalakat szolgáltat, mint például a Javadocs, akkor ehhez nem szükséges Apache szervert használnunk, sokkal egyszerűbben megoldható az `org.restlet.resource.Directory` osztály használatával. Ezen osztály használatának bemutatását egy egyszerű példa illusztrálja:

```
public class Pelda {
	public static final String ROOT_URI = "file:///e:/restlet/docs/api/";
	public static void main(String args[]) {
		Component component = new Component();
		component.getServers().add(Protocol.HTTP, 8182);
		component.getClients().add(Protocol.FILE);
		Application application = new Application() {
			@Override
			public Restlet createRoot() {
				return new Directory(getContext(), ROOT_URI);
			}
		};
		component.getDefaultHost().attach(application);
		component.start();
	}
}
```

A példa lefuttatása előtt arra van szükség, hogy a ROOT_URI-nak egy érvényes értéket állítsunk be, ez jelen esetben `e:/restlet/docs/api/.` Ebbe a könyvtárba kell elhelyeznünk a megfelelő statikus weblap főoldalát `index.html` néven. Futtatás után le is

tesztelhetjük a programunkat a http://localhost:8182 cím böngészőbe beírásával. Alkalmazásunk erre a kérésre a ROOT_URI-ban megadott HTML oldallal válaszol.

6.10 Restlet a mobilkommunikációban [6]

Kezdetben a Restlet csupán a Java SE, valamint a Java EE platformokon volt elérhető, azonban a technológia fejlődésével és a verziószámok növekedésével egyre több Java-alapú platform támogatta, így lépett be a támogatók körébe a Google, és az általa fejlesztett GAE (Google App Engine), valamint a GWT (Google Web Toolkit), amely az 1.7-es verziószám bevezetése óta segíti a RESTful webszolgáltatásokat. Mint az már az előző fejezetekből kiderült, a REST egy szoftver-architektúra stílus az elosztott hipermédia rendszerekben, vagyis olyan rendszerekben, amik elsősorban multimédia-alapú dokumentumok (szöveg, grafika, audió, videó, stb.) tárolására és kezelésére hivatottak. Ennél megfelelőbb szoftver-architektúra aligha létezik multimédiás mobiltelefonok számára, éppen ezért nem meglepő, hogy a Google az első mobiltelefonra szánt, Android névre keresztelt „operációs rendszerét" Restlet alapokkal dobta piacra. Az Android nem csupán operációs rendszer, hanem egy olyan szoftverkollekció, amely megába foglalja mind az operációs rendszert, mind a fejlesztői környezetet, és néhány kulcsalkalmazást is. Azonfelül, hogy támogatja a legelterjedtebb médiafájlokat (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF), ezt a teljes körű szolgáltatást nyílt forráskód támogatása mellett teszi, hiszen az Android SDK szolgáltatja az összes olyan eszközt és API-t, amelyek szükségesek az alkalmazások fejlesztésére ezen a platformon, természetesen Java nyelven. Amiért ennek a felfedezésnek külön alfejezetet szenteltünk nem más, mint a Google befolyása az üzleti életben, hiszen figyelembe véve a tényt, hogy a telekommunikáció a világ egyik leginnovatívabb, legfejlődőképesebb szegmense, könnyen belátható, hogy talán ebben rejlik a legnagyobb lehetőség a REST szoftver-architektúra stílust képviselő alkalmazások és az ezt használó eszközök széles körben történő elterjesztésére.

7 REST API-T TÁMOGATÓ WEBOLDALAK

Számos weboldal üzemeltetője lehetővé teszi, hogy a nyilvános erőforrásokat RESTesített HTTP kéréseken keresztül lekérdezhessük. Ebben a fejezetben bemutatjuk, hogyan néz ki egy lekérdezés, amely például a http://www.last.fm által támogatott API eszköztárát használja. A brit székhelyű Last.fm egy nagy sikerű zenei szolgáltató központ, amely napjainkban egyre nagyobb népszerűségnek örvend. Ezt nem csupán az egyre növekvő zenei kínálatának köszönheti, hiszen az erőforrásaik hatékony elérési lehetőségével egyre több alkalmazásba integrálják az általuk nyújtott szolgáltatásokat. Csupán regisztrálnunk kell a hivatalos honlapon, és ettől kezdve azonkívül, hogy ingyen jutunk nagy mennyiségű multimédia anyaghoz, lehetőségünk nyílik XML-RPC, vagy akár REST kéréseken keresztül is elérni ezeket. Csupán egy API kulcsra és egy titkos kulcsra van szükségünk (ezeket a szolgáltatótól kapjuk). Az API gyökér URI-ját a http://ws.audioscrobbler.com/2.0/ cím azonosítja. Általánosan elmondható, hogy minden metódust csomagnév.metódusnév formában hivatkozhatunk, a megfelelő paraméterekkel. Például a következő URI egy erőforrást azonosít:
http://ws.audioscrobbler.com/2.0/?method=album.getinfo&api_key=713089b61a603a0d082653967f02b6a0&artist=metallica&album=load.
A böngészőnk címsorába írva a fenti URI-t egy XML dokumentumban kapjuk meg a Metallica zenekar Load című albumára vonatkozó olyan fontos információkat, mint az album kiadásának dátumát (`releasedate`), a CD borítót reprezentáló képek közvetlen elérését (`image`), az aktuális hallgatók számát (`listeners`) vagy éppen az album azonosítókódját (`mbid`). Ez utóbbi azonosító azért is jelentős, mivel képezheti más kérések kötelező paraméterét is (ez általában elmondható az összes itt használt azonosítóról).
Egy metódusnak lehetnek kötelező illetve opcionális paraméterei, ezek az aktuális metódusra vonatkozóan az API-ban adottak. Az `album` csomag `getinfo` metódusát a következő ábra írja le, mint a 290-es sorszámú szolgáltatás specifikációja: http://www.last.fm/api/show?service=290

album.getInfo

Get the metadata for an album on Last.fm using the album name or a musicbrainz id. See playlist.fetch on how to get the album playlist.

Params

artist (Optional) : The artist name in question
album (Optional) : The album name in question
mbid (Optional) : The musicbrainz id for the album
lang (Optional) : The language to return the biography in, expressed as an ISO 639 alpha-2 code.
api_key (Required) : A Last.fm API key.

Ebben az esetben az `api_key` az egyetlen paraméter, melynek megadása kötelező. Amennyiben a kérésben ez nem szerepel, a válaszban a hiba jellegét egy állapot kód jelzi:

```
<lfm status="failed">
   <error code="10">
      Invalid API key - You must be granted a valid key by last.fm
   </error>
</lfm>
```

Természetesen minden hibának egyedi kódja van, mely hibakódok ebben az API-ban a következőek:

8 : Operation Failed – Szerver hiba.

2 : Invalid service – Ez a szolgáltatás nem létezik.

3 : Invalid Method – Nem létezik ilyen nevű metódus.

4 : Authentication Failed – Nincs megfelelő jogosultság a szolgáltatáshoz.

5 : Invalid format – A szolgáltatás nem létezik ebben a formátumban.

6 : Invalid parameters – A kérésből hiányzik egy kötelező paraméter.

7 : Invalid resource specified

9 : Invalid session key – Újraazonosítás szükséges.

10 : Invalid API key – Érvényes last.fm API kulcs szükséges.

11 : Service Offline – A szolgáltatás átmenetileg nem elérhető.

12 : Subscription Error – Csak előfizető felhasználók számára elérhető ez a szolgáltatás.

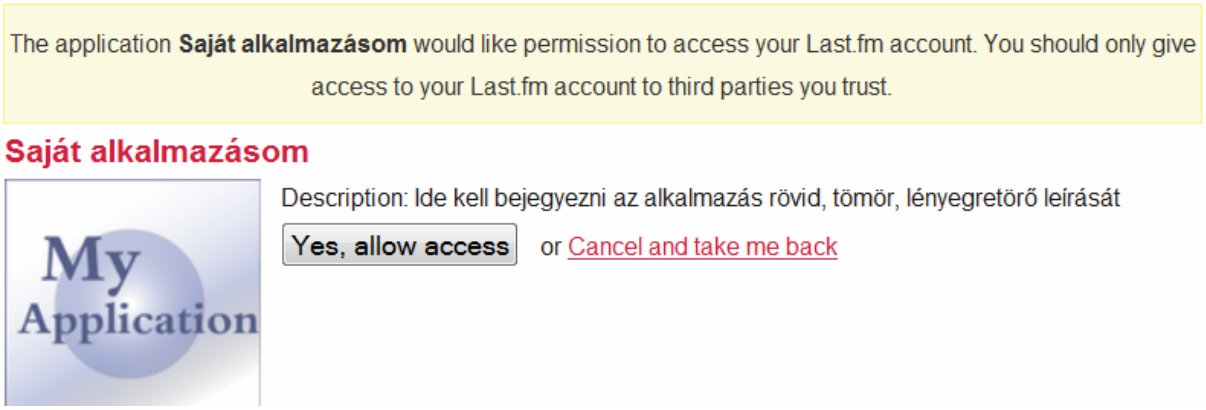
13 : Invalid method signature supplied

Web-böngészőből már tudunk kéréseket indítani, azonban mi a helyzet abban az esetben, ha a kérés egy asztali alkalmazásunk felől érkezik? Ebben az esetben az `Auth` csomag `gettoken` metódusának visszatérési értékét kell használnunk az authentikációhoz. Az API kulcs és az előzőekben leírtak fényében már nem szorul magyarázatra az alábbi kérés:
http://ws.audioscrobbler.com/2.0/?method=auth.gettoken&api_key=713089b61a603a0d082653967f02b6a0, és ezen kérésre kapott válasz:

```
<lfm status="ok">
      <token>0c4dfb92008a10a16f8b69f023e359a9</token>
</lfm>
```

Most már a kezünkben van minden eszköz (`api key`, `token`), amely szükséges az adott alkalmazás bejegyzéséhez:

http://www.last.fm/api/auth/?api_key=713089b61a603a0d082653967f02b6a0&token=0c4dfb92008a10a16f8b69f023e359a9



A „Yes, allow access” gomb megnyomásával lehetővé tesszük, hogy az alkalmazásunk hozzáférhessen a Last.fm account-unkhoz.

8 WEBSZOLGÁLTATÁSOK BIZTONSÁGA

A webszolgáltatások lehetővé teszik különböző külső entitások számára, hogy alkalmazásokat hívjanak meg, erőforrásokat kérjenek le és módosítsanak. Az adatok platformok közötti gyakori átvitele, valamint a bizalmas információk védelme összetett biztonsági intézkedések bevezetését igénylik. Különböző biztonsági szabványok jelentek meg a fenti problémák kezelésére, a következőkben ezek közül mutatunk be néhányat.

8.1 A HTTP biztonsági kérdései [7]

A HTTP egy úgynevezett challenge-response típusú hitelesítést alkalmaz. A felhasználónak bejelentkezési név és jelszó megadásával azonosítania kell magát, mielőtt hozzáférhetne a védett erőforrásokhoz. Ezen megoldás hátránya, hogy a felhasználónév és a jelszó titkosítás nélkül kerül át a szerverhez. A HTTP 1.1 specifikáció már tartalmazza a digest típusú hitelesítést. Ennek lényege, hogy a felhasználó azonosítására vonatkozó adatokat ellenőrző összeg formájában küldik át a szervernek. Az ellenőrző összeget az URI-ból, a HTTP metódusból, a bejelentkezési névből, jelszóból és egy egyszeri értékből generálják. Ezzel a megoldással a felhasználói adatok már nem plain-text formátumban kerülnek átvitelre, így védi az adatokat egy harmadik fél általi illetéktelen felhasználástól. Az ellenőrző összegek meghatározásához az MD5 (Message-Digest algorithm 5) nevű kódolási algoritmust használják. Ennek segítségével az adatokból egy 128-bites hash értéket, üzenetkivonatot hoznak létre. Mivel az algoritmus egyirányú kódolást alkalmaz, ezért nincs lehetőség a kódolt üzenetből visszanyerni az eredeti tartalmat. Az MD5 algoritmus használata ellen szól, hogy az implementációjában apróbb hibát találtak, illetve a későbbiekben más biztonsági résekre is felfigyeltek, ezért javasolt más kódolási eljárás használata, mint például az SHA-1. Természetesen a HTTP lehetőséget ad egyéb hitelesítő technológiák alkalmazására is. Ezek a hitelesítést biztosító megoldások tehát az üzenet sértetlenségének igazolására már alkalmasak, viszont az üzenetek tartalmára vonatkozó titkosítást nem valósítják meg.

8.2 SSL és a HTTPS

Az SSL (Secure Socket Layer) technológia weboldalak és webszolgáltatások biztonságos adatátvitelét teszi lehetővé. Leginkább HTTP kliens-szerver alkalmazásoknál használják. Az SSL tanúsítvány egyedi, a tanúsítvány tulajdonosának authentikációjához szükséges adatokat tartalmaz. Egy hitelesítő szervezet (Certificate Authority) igazolja a tanúsítvány tulajdonosának azonosításának hitelességét. Az SSL tanúsítvány tartalmaz egy titkos és egy nyilvános kulcsot. A nyilvános kulccsal kódolják az információt, a dekódoláshoz pedig a titkos kulcsot használják. Általában 128-bites SSL tanúsítványokat alkalmaznak a magasabb biztonsági szint eléréséhez. Ez viszont azt eredményezi, hogy túl nagy mennyiségű adatot kell mozgatni a kliens és a szerver között, ráadásul a titkosítási eljárás műveletei jelentős mértékben lekötik a processzort. Ezen problémák kiküszöbölésére vezették be az SSL gyorsítók használatát, amelyek olyan hardver vagy szoftver eszközök, amelyek átveszik a szervertől az SSL titkosításhoz kapcsolódó műveletek végrehajtását [8].

A HTTPS (Hypertext Transfer Protocol Secure) a HTTP protokoll és az SSL technológia együttes használatából született protokoll. Használatával biztonságos HTTP kérések és válaszok küldhetőek titkosított SSL csatornán keresztül, ezáltal védhetjük a kliens és a szerver közötti kommunikációt a lehallgatás és a man-in-the-middle támadások ellen. HTTPS működéséhez a HTTP-vel ellentétben nem a 80-as, hanem a 443-as portot használja [9].

8.3 XML Signature és XML Encryption

XML alapú kommunikáció során az XML dokumentumok sima szövegként tartalmazzák az információkat, így illetéktelenek is könnyedén hozzáférhetnek a bennük tárolt adatokhoz. Az átvitelre kerülő dokumentumok eredetiségének igazolására a digitális aláírások szolgálnak megoldásként. Az XML Signature a W3C által meghatározott XML alapú szabvány digitális aláírások elkészítéséhez. A specifikáció a DSS-t (Digital Signature Standard) és az SHA-1 (Secure Hash Algorithm) algoritmusokat használja. Az XML aláírást bármilyen állományhoz készíthetünk, nem csak XML dokumentumokhoz. A SHA-1 algoritmus az aláírás alapját képező dokumentumból egy hash értéket készít, amely az üzenet kivonata. Ezen értékből nem lehet visszanyerni az üzenet tartalmát, de felhasználható annak igazolására, hogy a dokumentum az átvitel során nem változott. Az

XML Signature három különböző lehetőséget kínál egy dokumentum aláírással történő megjelölésére:

- Különálló aláírás (Detached signature): az aláírást és az aláírt dokumentumot külön tárolják.
- Borítékolt aláírás (Enveloped signature): az aláírt dokumentum tartalmazza az aláírást.
- Borítékoló aláírás (Enveloping signature): az aláírt dokumentum az aláírás részeként kerül letárolásra.

Az XML Signature alkalmazásával már igazolni tudjuk, hogy a dokumentum az aláírást követően nem módosult, viszont abban nem lehetünk biztosak, hogy továbbítása során harmadik fél nem jutott hozzá a benne tárolt adatokhoz. Az XML Encryption egy olyan specifikáció, amely XML Signature aláírással ellátott adatok titkosítására szolgál.

8.4 XKMS [10]

Az XKMS (XML Key Management Specification) egy W3C által meghatározott szabvány, amely a webszolgáltatások kommunikációjához használható PKI-hoz (Public Key Infrastructure) tartozó nyilvános kulcsok regisztrálását és szétosztását határozza meg. Az XKMS két részből áll: X-KISS (XML Key Information Specification) és az X-KRSS (XML Key Registration Service Specification). Az X-KISS egy biztonságos szolgáltatást nyújt a nyilvános kulcs információk XML Signature elemekben történő tárolására. Így lehetőség nyílik arra, hogy a kulcsokat tartalmazó elemekkel kapcsolatos feladatok ellátását a kliens helyett egy szolgáltatás végezze el. Ez a szolgáltatás lehetőséget ad digitális aláírások készítésére a nyilvános-titkos kulcspárok használatával. A nyilvános kulcs információk regisztrálásához az X-KRSS egy webszolgáltatást hoz létre. Egy regisztrált nyilvános kulcsot a későbbiekben az X-KISS-sel és más webszolgáltatásokkal együtt tudunk használni. Az XKMS tehát egy olyan biztonságos szolgáltatást határoz meg, amely digitális aláírás készítésére és dokumentumok hitelességének igazolására alkalmas, ezáltal biztosítja, hogy az eredeti címzetten kívül más ne tudja elérni és módosítani a dokumentum tartalmát.

8.5 Web Services Security [11]

A webszolgáltatások általában HTTP protokollon keresztül bonyolítják le a kommunikációjukat. A HTTP lehetőséget nyújt a kliens és a szerver authentikációjára, az üzenet kódolására és aláírására, viszont ezeket csak pont-pont kapcsolat esetén tudja biztosítani. A WS-Security egy olyan keretrendszert határoz meg, amely támogatást nyújt a már meglévő biztonsági szabványok és specifikációk integrálására, mint például a XML Signature, XML Encription, SSL, X.509, Kerberos, ezáltal a több pont közötti biztonságos kommunikáció megvalósítására is lehetőség van. A SOAP üzenetek fokozott védelme érdekében meghatározza, hogyan lehet aláírást és a kódolásra vonatkozó információkat elhelyezni a SOAP fejrészben. A felhasználó hitelesítésére szolgáló információkat a `UsernameToken` elemben, az üzenet kódolására, illetve az aláírásra vonatkozó információkat pedig a `BinarySecurityToken` elemben helyezhetjük el. Ezen felül még szükségünk van annak meghatározására, hogy egy adott felhasználó milyen jogosultságokkal rendelkezik. Az azonosításra és a hozzáférési jogok igazolására biztonsági tokeneket (security token) alkalmaznak. A security token létrehozása PKI, Kerberos használatával, esetleg felhasználónév jelszó pár megadásával történhet. Ezen tokenek használatával még nem garantáltuk a teljes SOAP üzenet biztonságos átvitelét, ugyanis az üzenet lehallgatásával a támadó a fejrészből megszerezheti egy új, hitelesnek tűnő üzenet elkészítéséhez szükséges információkat, ezért fontos a fenti intézkedéseken kívül a SOAP üzenet kódolása és aláírása. A SOAP fejrésznek tartalmaznia kell a kódolásra, illetve az aláírásra vonatkozó információkat is, hogy a címzett dekódolni tudja az üzenetet és ellenőrizni tudja az aláírás hitelességét.

8.6 OpenID [12][13]

A hozzáférés szabályozást alkalmazó webalkalmazások és webszolgáltatások általában felhasználónevet és jelszót kérnek a klienseik azonosításához. A felhasználóknak tehát a különböző szolgáltatások igénybevételéhez regisztrálniuk kell és különféle felhasználónév és jelszó párokat kell megjegyezniük. Az OpenID-t éppen ennek kiküszöbölése, illetve a kényelem és a biztonság fokozása érdekében fejlesztették ki. Ez egy olyan decentralizált, nyílt szabvány, melynek használatával minden felhasználói információ egy helyen tárolható és a felhasználók a weboldalakhoz hasonlóan egy URI-val azonosítsák magukat. Mostanra már a Google, Yahoo, Flickr, Myspace és a Microsoft

is beszállt az OpenID-t nyújtó szolgáltatók közé. Ha szeretnénk egy saját OpenID azonosítót, nem kell mást tennünk, mint hogy ezen weboldalak valamelyikére beregisztrálunk. A felhasználónéven és jelszón kívül más személyes információk megadására is lehetőség van, mint például az e-mail cím, telefonszám, cím, stb. Például ha már rendelkezünk egy Google által szolgáltatott OpenID URI-val, akkor ennek segítségével beléphetünk egy olyan webalkalmazás oldalára, amely támogatja az OpenID-t. Ebben az esetben a webalkalmazás a Google-től kérni fogja, hogy hitelesítse a digitális identitásunkat. Az igazolást XRDS (eXtensible Resource Descriptor Sequence) formátumban küldi majd át a Google, amely egy a Yadis kommunikációs protokoll által támogatott XML alapú dokumentum. Amennyiben a webalkalmazásnak a felhasználónéven és jelszón kívül még több információra van szüksége a bejelentkezésünkhöz, akkor átirányít a Google oldalára, ahol engedélyezhetjük, hogy a Google ezt a személyes információnkat is kiadhassa a webalkalmazás számára. Ezek után pedig új felhasználónév és jelszó nélkül sikeresen megtörténik a bejelentkezés. Viszont egyes szolgáltatók, mint például a Microsoft, Yahoo és a Google korlátozzák az OpenID használatát azáltal, hogy más szolgáltatás által nyújtott OpenID-val nem lehet bejelentkezni a rendszereikbe. Az OpenID protokoll ilyen irányú szétválasztásával egy új protokollt határoznak meg, amely tulajdonképpen a szolgáltatásaikra regisztrált felhasználóknak nyújt egy plusz szolgáltatást azáltal, hogy más webalkalmazások igénybevételéhez nincs szükségük új regisztrációra. Ezzel a megkötéssel az eredeti OpenID technológián alapuló, de mégis annak filozófiájával inkompatibilis protokollt határoztak meg, amely megosztja a közvéleményt.

Az OpenID protokoll egyik hátránya, hogy fennáll a man-in-the-middle támadás lehetősége abban az esetben, ha a felhasználó egy nem megbízható webalkalmazás használatához kéri az OpenID szolgáltatója általi hitelesítést. Már vannak arra irányuló kezdeményezések, amelyek során erre a biztonsági problémára keresik a választ, de a megoldás még várat magára.

8.7 A REST biztonsági kérdései [14]

Mint azt az előző fejezetekből megtudhattuk REST-ben a műveletek meghatározásához HTTP metódusokat használunk. Az URI-k által azonosított erőforrások reprezentációját a PUT, POST és DELETE metódusok megváltoztathatják, viszont a GET nem

módosíthatja. Ez egy alapvető biztonsági elvárás, amely alapján lehetőség lenne az erőforrások hozzáférésének szabályozására. Bizonyos felhasználók például csak GET műveletet hajthatnának végre egy adott erőforráson, mások pedig jogot kapnának annak módosítására is. Ezzel a megközelítéssel az a probléma, hogy a fejlesztőknek maximálisan tisztában kell lenniük a REST elveivel és be is kellene őket tartaniuk, jelenleg viszont ezt nem tudjuk biztosítani. Például a REST API-t támogató Flickr is tartalmaz egy olyan törlés műveletét, amelyet HTTP GET végrehajtásával valósítanak meg. Ilyen feltételek mellett az erőforrásra alkalmazható metódusok körét nem tudjuk korlátozni. Egy másik megközelítés szerint mivel az erőforrásokat az URI-val azonosítjuk, lehetőségünk van az URI által tartalmazott QueryString paramétereinek vizsgálatára. Így a webalkalmazásokban is használt biztonsági intézkedéseket tudunk tenni, mint például a QueryString-ben lévő paraméterek számának vizsgálata, ezen paraméterek tartalmának ellenőrzése. Egy GET kérés esetén az URI egyértelműen meghatározza a kérést, ezáltal lehetőség van az URI tudatában a kérés újrajátszására, tetszőleges sokszori végrehajtására. Az összetettebb REST webszolgáltatásokat nyújtó Amazon és Google ezen probléma kiküszöbölésére az úgynevezett Developer tokeneket alkalmazza. Ezt a tokent a QueryString tartalmazza név-érték párként. Ez egy olyan egyszerű biztonsági paraméter, amellyel például korlátozható az egy napon kiadott webszolgáltatás kérések száma. A kereskedelmi szolgáltatásokat is tartalmazó Amazon webszolgáltatások lehetővé teszik a hitelkártyával történő online fizetés lebonyolítását is, az ilyen jellegű bizalmas információk védelme érdekében összetett Developer tokeneket alkalkalmaznak. Szükség van egy „SubscribeID”-ra, amely tartalmaz egy „Access Key ID”-t és egy „Secret Access Key”-t. Az Access Key ID-t a webszolgáltatás igénybe vevő azonosítására használják, a Secret Access Key-t pedig HMAC (keyed-Hash Message Authentication Code) generálásához szükséges, amelyet a kérések hitelesítéséhez alkalmaznak. A kérések újrajátszásának megelőzésére a HMAC kiszámításához felhasználják az időbélyeget is.

9 EGY KÉPGALÉRIA KEZELŐ PROGRAM BEMUTATÁSA

9.1 Bevezetés

Az előző fejezetekben már betekintést nyerhetett a kedves olvasó abba, hogyan valósíthatunk meg konkrét webszolgáltatásokat XML-RPC, SOAP, illetve az általunk is preferált REST alkalmazásával. Az így született programkódok egyszerűsége és tömörsége csupán az implementáció megértését segítik, mivel nem volt célunk terjedelmes forráskódokkal nehezíteni az olvasást. Ebben a fejezetben azonban kicsit komplexebb feladat bemutatására kerül sor, mégpedig egy képgaléria kezelő program segítségével prezentáljuk a REST adta lehetőségeket, annak minden előnyével és hátrányával.

9.2 A fejlesztői környezet, és a munkát segítő eszközök

9.2.1 Eclipse

Szükségünk volt egy olyan szoftverre, ami széleskörűen támogatja a fejlesztés során előforduló feladatokat. Elsődleges szempont volt a kényelmes eszközhasználat, illetve a funkciók kiterjesztésének lehetősége. A nyílt forráskódú Eclipse fejlesztői környezet mellett döntöttünk, hiszen azon kívül, hogy nagyon jó a támogatása, és rengeteg ingyenes és kereskedelmi plugin érhető el hozzá, jó felépítésű és könnyen használható is. Több száz elérhető bővítmény segítségével szinte az összes ma használatos fejlesztési technológiát támogatja. Példának okáért az alkalmazásunk grafikus felületének gyors megvalósításához nélkülözhetetlen bővítményként szolgált a Visual Swing nevű Eclipse plugin.

9.2.2 Maven

A fordítás automatizálásában és a projekt kezelésében nagy segítséget jelentett az Apache által fejlesztett Maven, egy olyan nyílt forráskódú szoftver, amely funkcionalitását tekintve leginkább az Apache Ant-hoz hasonlít. Elsősorban Java projektekben használják. A Projekt Objektummodell (Project Object Model) [15] fogalma egyre ismertebbé vált napjainkban. A mi esetünkben arról van szó, hogy egy XML dokumentum (pom.xml) tartalmazza a projekt deklaratív leírását, így részletes információt szolgáltatva az adott projektről, a projekt fejlesztőiről, levelezési lista címekről, függőségekről és azok

hierarchiájáról, különböző jelentésekről, stb. A végrehajtandó tevékenységeket céloknak (goals) nevezzük, ezekből vannak előre definiáltak (pl. a projekt életciklusának lépései: compile, test, package, install, deploy), de természetesen a felhasználónak is lehetősége van saját, projekt-specifikus célok definiálására. A másik fontos fogalom a repository, ezen a néven illetjük a különböző hosztok fájlrendszereinek azon mappáit, ahol a letöltendő komponensek találhatóak. Természetesen repository-kból nem csak letölteni tudunk, a Maven támogatja az általunk elkészült szoftvercsomagok feltöltését is.

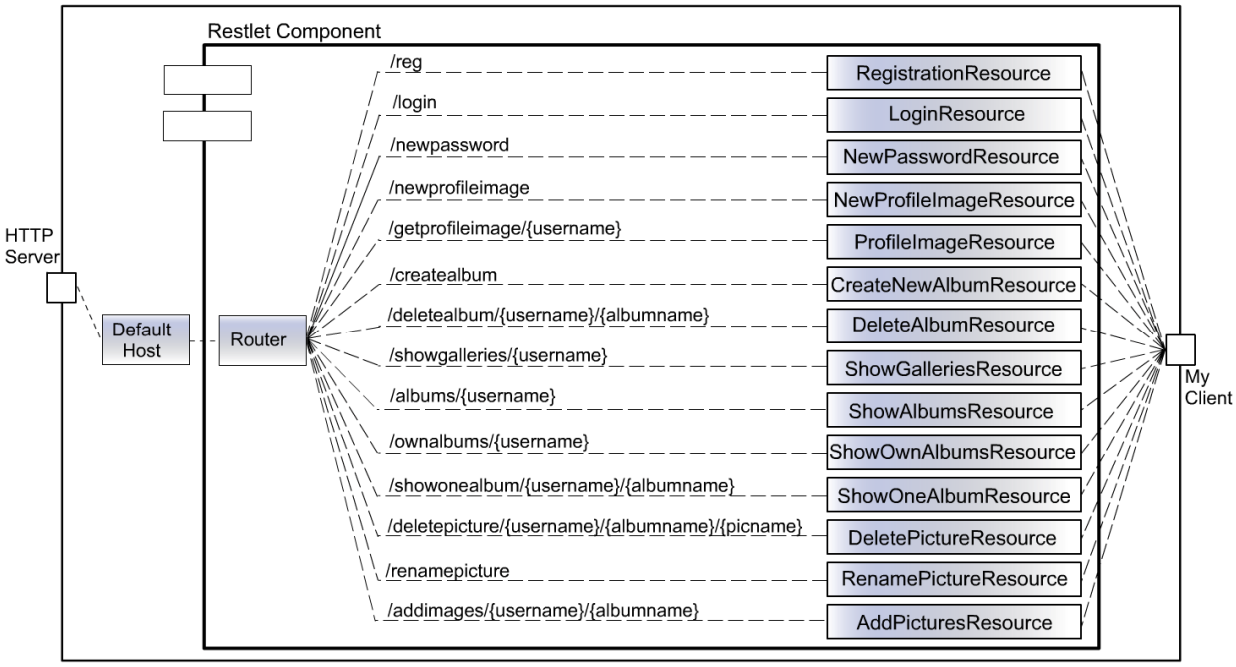
9.3 A szerver és a kliensek kommunikációja

Célunk egy több felhasználós képnyilvántartó alkalmazás létrehozása volt. A különböző felhasználók nyilvántartását a szerver végzi, amely egy állományból olvassa a felhasználók adatait és regisztráció esetén ebbe az állományba menti el az új adatokat. A szerver kódja:

```
public class Server {
   public static void main(String[] args) throws Exception {
      Component component = new Component();
      component.getServers().add(Protocol.HTTP, 7777);
      component.getDefaultHost().attach("/app", new ResourceApplication());
      component.start();
   }
}
```

A 5.3 fejezetben bemutatott példához hasonlóan egy `Component` típusú objektum szolgál a szerver alapjául, melynek konstruktora egyértelműen meghatározza a használni kívánt protokollt és a portot. Ezen objektum alapértelmezett hosztjához csatoljuk a „/app” URI mintát abból a célból, hogy az erre a mintára illeszkedő URI esetén a kliens oldalról érkező kérés továbbítása a `ResourceApplication` osztály egy példányához történjen. A `ResourceApplication` osztály az `Application` osztálytól örökölt `createRoot` metódus felülírásával biztosítja, hogy ez a Restlet képes legyen az összes beérkező kérés fogadására. Ez a metódus tartalmaz egy `Router` típusú objektumot, amely lehetővé teszi, hogy a kéréshez tartozó URI és az itt megadott URI minták illeszkedése alapján a megfelelő Restlet target-hez tudjuk irányítani a kliens kérését. A kliens számos szerver oldali szolgáltatást igénybe vehet, így például a felhasználó a regisztrációt és a

bejelentkezést követően megváltoztathatja a személyes adatait (profilkép, jelszó, egyéb adatok), a grafikus felület segítségével megtekintheti saját és más regisztrált felhasználók képgalériáit, új albumokat hozhat létre, és módosíthatja a már meglévő albumok tartalmát (képek lecserélése, hozzáadása, törlése). A következő ábra specifikálja a program teljes funkcionalitását és bemutatja a HTTP szerver, valamint a kliens közötti kapcsolatot.



9.3.1.ábra: A szerver és a kliens közötti kapcsolat

Tehát ha az alapértelmezett hosztra érkezett kérés által kijelölt URI illeszkedik például a `/showonealbum/{username}/{albumname}` URI mintára, akkor a vezérlés a forgalomirányító által a `ShowOneAlbumResource` osztályra kerül, amely egy paraméterben megadott felhasználó megadott nevű albumának összes képének szolgáltatásáért felelős. Mivel a kérések kiszolgálási mechanizmusa hasonló, ezért a szóban forgó példa részletes tárgyalása hasznos lehet akár a teljes program megértésében:

```
public class ShowOneAlbumResource extends ServerResource {
   String userName;
   String albumName;

   protected void doInit() throws ResourceException {
      this.userName = (String) getRequest().getAttributes().get("username");
      this.albumName =
            (String) getRequest().getAttributes().get("albumname");
      setExisting(this.userName != null && this.albumName != null);
   }
   @Get
   public Representation get() {
      File[] images = ListFileNames.getAllImage(userName, albumName);
      ImageFiles imageFiles = null;
      if(images != null)
         imageFiles = new ImageFiles(images);
      ObjectRepresentation<ImageFiles> objRep =
          new ObjectRepresentation<ImageFiles>(imageFiles);
      return objRep;
   }
}
```

A `doInit` metódus az osztályra való hivatkozáskor lefut és inicializálja az adattagokat, illetve levizsgálja, hogy az URI-ban megadtak-e minden információt a kérés teljesítéséhez. A `ServerResource` osztály és alosztályainak példányai kezelni tudják a szerver konnektor által fogadott hívásokat, csupán implementálnunk kell hozzá a megfelelő REST metódust. A legegyszerűbb mód ennek megvalósításához a következő annotációk valamelyikének használata: `@Get`, `@Post`, `@Put`, `@Delete`. Jelen esetben a `@Get` annotációval ellátott metódus szemantikailag ekvivalens a HTTP GET metódussal. Az `ImageFiles` egy általunk létrehozott osztály, amely egyetlen kollekció adattagjában tárolja a kérésben meghatározott felhasználó megfelelő albumjában lévő összes képét. Ezen osztály egy példányát fogjuk az `ObjectRepresentation` osztály konstruktorában megadni. Így egy szerializálható Java objektum reprezentációját kapjuk, amelyet a kérés megválaszolásaként visszaküldhetünk a kliensnek.

Egy másik lehetőség, hogy definiálunk egy Restlet handlert. Tekintsünk tehát most egy olyan példát, amikor az alapértelmezett hosztra érkező kérés által kijelölt URI a `/deletepicture/{username}/{albumname}/{picname}` URI mintára illeszkedik. Ezt

a megoldást szemlélteti a következő kód, amely egy meghatározott nevű felhasználó adott nevű albumából töröl egy képet.

```
public class DeletePictureResource extends ServerResource {
   public static Restlet gethandler() {
       Restlet handler = new Restlet() {
          @Override
          public void handle(Request req, Response res) {
              Method http_verb = req.getMethod();
              if (http_verb.equals(Method.DELETE)) {
              try {
                 String username = (String)req.getAttributes().get("username");
                 String albumname=(String)req.getAttributes().get("albumname");
                 String picname = (String)req.getAttributes().get("picname");
                 boolean success = (new File("Users/" + username + "/" +
                                    albumname + "/" + picname)).delete();
                 if(success)
                    res.setStatus(Status.SUCCESS_OK);
                  else res.setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
              } catch (Exception e) {
                 e.printStackTrace();
                  }
              } else
                  res.setStatus(Status.SERVER_ERROR_NOT_IMPLEMENTED);
              }
           };
       return handler;
   }
}
```

Az annotáció használatával ellentétben itt a `handle` metódus törzsében meg kell vizsgálnunk, hogy a kliens felől érkező kérés melyik HTTP metódusnak felel meg, és ez alapján adjuk meg a további utasításokat.

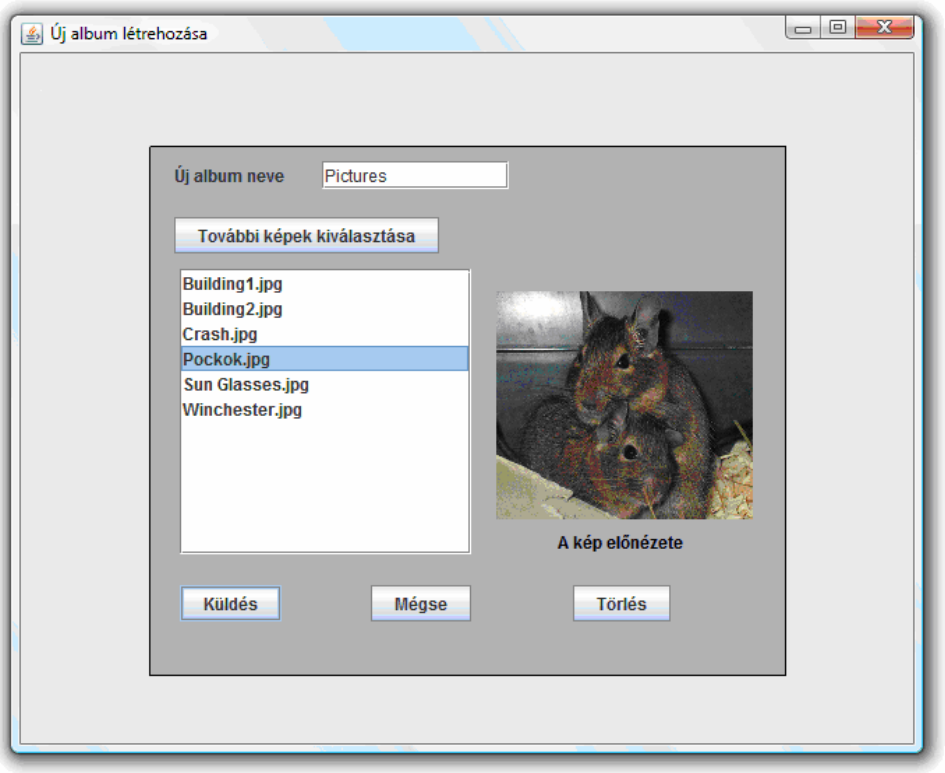
Amennyiben a szerver oldalon nem támogatott kérés érkezik a kliens felől, akkor a válasz státuszkódját `Status.SERVER_ERROR_NOT_IMPLEMENTED`-re állítjuk, ezzel tudatjuk a klienssel, hogy kérése nem került feldolgozásra.

A kliens oldal megvalósításáért a `MyClient` osztály felelős, amelyben rendre megtalálható az összes kérés küldését megvalósító metódus implementációja. Példának

okáért a `send_getAlbum` metódus állítja össze és bonyolítja le azt a kérést, amelyet a szerver oldalon már bemutatott `ShowOneAlbumResource` osztály fog kezelni.

```
public static ImageFiles send_getAlbum(String userName, String albumName){
   String uri = "http://localhost:7777/app/showonealbum/" + userName +
                                                          "/" + albumName;
   ClientResource getOneAlbumResource = new ClientResource(uri);
   ImageFiles imageFiles = null;
   try{
      Representation rep = getOneAlbumResource.get();
      ObjectRepresentation<ImageFiles> objRep =
                                    new ObjectRepresentation<ImageFiles>(rep);
      imageFiles = objRep.getObject();
   } catch (Exception e){
      e.printStackTrace();
   }
   return imageFiles;
}
```

Szükségünk van egy `ClientResource` objektumra, amely segítségével el tudjuk küldeni a kérést és megkaphatjuk a szerver válaszát. Ennek az objektumnak meghívjuk a `get` metódusát, amely ennek eredményeként a `ClientResource` konstruktorában megadott URI-val összeállított HTTP GET kérést küld a szervernek, amely válaszként a kérésnek megfelelő erőforrás reprezentációját szolgáltatja. Tudjuk, hogy a szerver `ObjectRepresentation` típusú objektumot ad vissza, amely egy szerializálható `ImageFiles` típusú objektum reprezentációja, ebből az objektumból a `getObject` metódus meghívásával megkaphatjuk az `ImageFiles` típusú objektumot, amely tartalmazza a megfelelő képeket tartalmazó kollekciót. Az így kapott képek a kliens oldalon tetszőleges módon feldolgozhatóak, illetve megjeleníthetőek a grafikus felületen.



9.3.2. Ábra: Képek feltöltése kliens oldalról a szerverre

9.3.3. Ábra: Szerveren tárolt képek megtekintése a kliens oldalon

A korábbiakban a szerver oldalon már bemutatott `DeletePictureResource` által kezelt kérésekre is nézzünk egy példát.

```
public static boolean send_deletePicture(String albumname, String picname){
   Request request = new Request();
   String uri = "http://localhost:7777/app/deletepicture/" +
                Identity.getUserName() + "/" + albumname + "/" + picname;
   request.setResourceRef(uri);
   Client client = new Client(Protocol.HTTP);
   request.setMethod(Method.DELETE);
   try {
      Response response = client.handle(request);
      if(response.getStatus().equals(Status.SUCCESS_OK))
         return true;
   }catch (Exception e) {
      e.printStackTrace();
   }
   return false;
}
```

Ebben a metódusban összeállítunk egy `Request` típusú objektumot, amely a kérés paramétereit fogja tartalmazni, mint az URI és a HTTP metódus neve. A kérés elküldéséhez most egy `Client` típusú objektumot használunk, konstruktorában a kívánt protokoll megadásával. A kérés elküldésének lebonyolításához ezen objektum `handle` metódusa kerül meghívásra, felparaméterezve a szóban forgó `request` objektummal. Ez a metódus visszatérési értékként a szerver által visszaadott választ szolgáltatja egy `Response` típusú objektum formájában. Mivel jelen esetben törlésről van szó, ezért a kliensnek csupán arról kell meggyőződnie, hogy a szerver sikeresen törölte a kérésben meghatározott erőforrást, így elég a válasz státuszkódját megvizsgálva megbizonyosodnunk a művelet végrehajtásának sikerességéről.

9.4 Szerveroldali naplók készítése

Naplózás használatával a szerver oldalon nyilvántarthatjuk a kliensek felől érkezett kéréseket, valamint a kérésekre adott válaszok státuszkódját. Ezek ismeretében a későbbiekben a hibák felderítése egyszerűbbé válik. A `Component` osztály

alapértelmezetten tartalmaz egy `logger`-t, amely a szerver oldali konzolra írja ki az információkat egy meghatározott formában:

```
2009.11.05. 16:40:23 org.restlet.engine.log.LogFilter afterHandle
INFO: 2009-11-05    16:40:23     127.0.0.1     -      -      7777   PUT
      /app/login    -      200    2      23     63     http://localhost:7777
      Noelios-Restlet/2.0m4      -
```

A második sor a W3C Extended Log File Format által meghatározott formátum, amely ma már széles körben elterjedt. A Restlet természetesen lehetővé teszi tetszőleges, a felhasználó által meghatározott formátumú naplók bevezetését és azoknak fájlba történő kiírását. Példánkban a `java.util.logging` csomag osztályait fogjuk használni a naplózás végrehajtásához. Ehhez a megvalósításhoz először is a `Component` objektumot tartalmazó osztályban szükségünk van egy `Logger` objektumra.

```
private static Logger logger = Logger.getLogger("my.restlet.logger");
```

A `getLogger` metódus visszaad egy a paraméterében megadott nevű `Logger` típusú objektumot, amennyiben az már létezik, ha pedig nem, akkor létrehoz egy újat. A névre (`my.restlet.logger`) azért lesz szükség, mert ezzel fogjuk megteremteni a kapcsolatot a `Logger` és a `Component logService` objektuma között. A napló fájlba történő kiírását a következő kódrészlet szemlélteti:

```
LogService logService = component.getLogService();
logService.setEnabled(true);
logService.setLoggerName("my.restlet.logger");
logService.setLogFormat("{cia}\t{m}\t{S}\tREF:{hr}{rp}\tAGENT:{cig}");
FileHandler fileHandler = new FileHandler("log/logging.config", true);
fileHandler.setFormatter(new MyFormatter());
logger.addHandler(fileHandler);
```

Egy `FileHandler` objektum segítségével megadjuk a naplózás kimenetéül szolgáló log fájlt, valamint engedélyezzük, hogy ezen fájl tartalmához mindig hozzáfűzze a következő bejegyzést. Ugyanezen objektum `setFormatter` metódusának használatával

megadhatjuk a kimeneti fájlban megjelenő bejegyzések formátumát. A megjelenítendő adatokat {név} formátumban adhatjuk meg, mint például

- `{cia}` request.clientInfo.address,
- `{m}` request.method,
- `{S}` response.status,
- `{hr}` request.hostRef.remainingPart,
- `{rp}` request.resourceRef.path,
- `{cig}` request.clientInfo.agent.

A teljes listát a Restlet API `org.restlet.util.Template` osztálynál találhatjuk meg. Alapértelmezetten a logger XML bejegyzéseket helyez el a fájlban, ami viszont tartalmaz több olyan információt is, amelyre nem feltétlenül lesz a későbbiekben szükségünk és csak növeli a napló méretét. Például egy bejelentkezéshez tartozó naplórészlet:

```
<?xml version="1.0" encoding="windows-1250" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2009-11-05T19:20:53</date>
  <millis>1257445253984</millis>
  <sequence>1</sequence>
  <logger>my.restlet.logger</logger>
  <level>INFO</level>
  <class>org.restlet.engine.log.LogFilter</class>
  <method>afterHandle</method>
  <thread>11</thread>
  <message>127.0.0.1      PUT    200    http://localhost:7777/app/login
      Noelios-Restlet/2.0m4</message>
</record>
```

Természetesen lehetőségünk van megváltoztatni a fájlba írandó napló szerkezetét is. Ehhez szükségünk lesz egy osztályra, amely kiterjeszti a `Formatter` absztrakt osztályt és felülírja annak `format` metódusát.

```
public class MyFormatter extends Formatter {
   @Override
   public String format(LogRecord record) {
      SimpleDateFormat dateFormat =
          new SimpleDateFormat("yyyy.MM.dd HH:mm:ss");

      return (dateFormat.format(new Date())  + " - " +
         record.getMessage() + "\n");
   }
}
```

Ezen osztály egy objektumát adjuk át a `filehandler setFormat` metódusának, így a log fájlban csak a dátum, és a tényleges naplózandó sor fog szerepelni. A naplófájlunkba bejelentkezés esetén a következő sor fog bekerülni:

```
2009.11.05 19:22:10 - 127.0.0.1  PUT    200    http://localhost:7777/app/login
Noelios-Restlet/2.0m4
```

10 REST vs SOAP

Egy ESRI által publikált esettanulmány szerint míg 2003-ban szinte mindenhol SOAP-ot alkalmaztak webszolgáltatások megvalósítására, 2006-tól kezdődően egyre több helyen bukkant fel a REST, bár ekkor még a SOAP interfészek nagyobb funkcionalitást nyújtottak. Az utóbbi időben magas fokú versengés alakult ki a két szemléletmód támogatói között, ami sosem szabott határt a fejlődésnek, sőt kifejezetten elősegíti azt. Részben ennek a versengésnek is köszönhető, hogy ma már a SOAP és REST interfészek nagyjából egyforma funkcionalitást nyújtanak, és mindezt jól dokumentált, közvetlen elérhető formában. Egy webszolgáltatás kialakításánál azonban fontos az azzal szemben támasztott követelmények alapos átgondolása, és napjainkban az igazi kihívás már nem is annak eldöntése, hogy melyik technológia a jobb, hanem egy konkrét projekt megvalósítása céljából melyiket érdemes választani.

10.1 Az XML két arca, a reprezentációk harca

A SOAP az információkat tipikusan XML használatával reprezentálja. Ez egy egyszerű, széles körben elterjedt, szabványos és magas kifejező erejű nyelv, amely mind a gép számára, mind pedig az ember számára könnyen olvasható. Szinte bármilyen adatstruktúra leírására alkalmas, és ez a hierarchikus struktúra megfelel a legtöbb dokumentum típusnak, ráadásul logikailag ellenőrizhető a formátuma. Talán sikerült is felsorolni ennek az általános célú leíró nyelvnek a legfőbb előnyeit, sajnos azonban szembe kell nézni a legfőbb hátrányával, miszerint az XML nem különösen hatékony. Az köztudott, hogy ennek a nyelvnek elég bőbeszédű a szintaxisa, és azonfelül, hogy ez nehezíti az olvashatóságot, a nagy terjedelem és a redundáns elemek növelik a tárolási költséget, és ezáltal az alkalmazások hatékonyságát. Ez talán azért van, mert a SOAP a kezdetektől fogva a vállalati világra koncentrált, és ez a szabvány-orientált szemléletmód eredményezte az XML-hez való ragaszkodást, ami azonban teljesen lehetetlenné teszi a SOAP-os webszolgáltatások bizonyos alkalmazási területeken történő bevetését. Gondoljunk csak például a telekommunikációra: a különböző mobil eszközök (mobiltelefonok, PDA-k) közötti adatcsere korlátozott sávszélességen történik, amely nem alkalmas nagy méretű XML-el leírt adatok átvitelére, arról nem beszélve, hogy az eszközök tárolókapacitása is csekély. Bizonyos esetekben a tömörítés csökkentheti a problémát, de az olcsóbb mobil eszközök processzora nem rendelkezik olyan teljesítménnyel, ami ezt lehetővé tenné.

A REST-nek ezzel szemben megvan az a nagy előnye, hogy nem definiál szabványos adat reprezentációt, mivel az erőforrások tetszőleges reprezentációját képes kezelni. Nem kell messzire mennünk ahhoz, hogy megmutassuk ennek egy nyilvánvaló példáját. A Google 2009 márciusában jelentette be, hogy készek teljesen megszüntetni a SOAP Search API-t. A legfontosabb indok a következő volt: a REST jobb támogatást nyújt böngésző kliensek részére, mivel különböző formátumokat engedélyez. Például JavaScript kliensek számára időigényes műveletnek minősül az XML alapú dokumentumok feldolgozása, noha a SOAP nem kínál más alternatívát. Éppen ezért sokkal célravezetőbb és költséghatékonyabb erre a célra JSON-t (JavaScript Object Notation) használni, ami egy olyan programozási nyelvtől független, pehelysúlyú adatcsere-formátum, amely egy életképes alternatívát nyújthat nagy mennyiségű adat kliensoldali feldolgozásához.

A tetszőleges adat reprezentáció használatának lehetősége azonban a REST egyik hátrányának is tekinthető, mivel a lehetőségek számának növekedésével egyenesen arányosan nő a kliensoldali kódok komplexitása, ami egyébként sem az átláthatóságról híres. A fejlesztő dolga továbbá a kliensoldalon alkalmazott programozási nyelv, a fejlesztői környezet megválasztása mellett a használt API verziószámának megválasztása, szem előtt tartva azt a fontos szempontot, hogy a különböző verziók (funkcionalitás tekintetében) gyakran köszönő viszonyban sem állnak egymással. Ha ehhez még azt is hozzávesszük, hogy sok szoftverfejlesztő nem szeret a kliens oldalon nyers HTTP hívásokat írni, akkor kénytelenek vagyunk levonni a következtetést, miszerint a SOAP (kliens oldalról megközelítve) olykor sokkal egyszerűbb, mint a REST.

10.2 A protokollfüggetlenség egy (nem) kívánt jellemző?

Mivel a SOAP független az alatta levő kommunikációs csatornától, így tetszőleges átviteli protokoll (HTTP, JMS, FTP, stb.) felett megvalósítható. Támogatói gyakran tudják be a REST hibájának, hogy az protokollfüggő, és hogy alkalmazása a HTTP-re korlátozódik. A REST hívők szerint azonban a protokollfüggetlenség nem egy kívánt jellemző, sokkal inkább tekinthető tervezési hibának, állításukat pedig a következő magyarázattal indokolják: a SOAP protokollfüggetlenségének kialakítása azon az elgondoláson alapul, hogy megpróbáltak több különböző technológiát egyetlen absztrakciós réteg mögé rejteni, de az absztrakciókra általában jellemző, hogy hézagosak, így gyakran egy általunk választott protokoll jobb választásnak tűnhet, mint az az absztrakció, amivel helyettesíteni szeretnénk, különösen akkor, ha olyan széles körben elfogadott protokolltól függünk, mint amilyen az a HTTP. Ezenkívül érdemes megemlíteni, hogy bár a SOAP tetszőleges átviteli protokoll felett megvalósítható lenne, mégis a HTTP felett alkalmazzák leginkább (mégpedig azért, mert ezt probléma nélkül átengedik a tűzfalak, míg más protokollokról ez nem feltétlenül mondható el).

10.3 Interfész leíró nyelvek használata a REST-ben?

A SOAP ma már aligha képzelhető el WSDL nélkül, hiszen a szabványokhoz és az általános elvárásokhoz ragaszkodva nélkülözhetetlenné, megszokottá vált egy a webszolgáltatásokat leíró nyelv használata, és a betűszó szinte elválaszthatatlanná vált a SOAP-UDDI-WSDL hármastól. Nem meglepő, hogy a SOAP támogatói körében

gyakran támadják a REST-et azzal az indokkal, miszerint nincs általánosan elfogadott formális módja a RESTful webszolgáltatások interfész-leírásának, ami a műveletekről, azok neveiről, illetve a be- és kiviteli paraméterekről ad információkat. Mielőtt rátérnénk ennek tárgyalására, fontos kihangsúlyozni, hogy a REST, mint erőforrás-centrikus szoftver-architektúra szükségtelenné teszi az interfész leíró nyelvek használatát. Ha ugyanis az erőforrásainkat körültekintően implementáljuk, és kihasználjuk azt a lehetőséget, miszerint egy erőforrásnak több reprezentációja is lehet, könnyedén előállíthatunk olyan önmagukat dokumentáló erőforrásokat, amelyre kiadott alkalmas GET kérés szolgáltatja majd az erőforrást leíró dokumentumot, ami információt adhat az erőforrás által magába foglalt adatokról, az erőforráson elvégezhető műveletek listájáról, és a támogatott tartalomtípusokról. Egy másik lehetőség, hogy az erőforrás reprezentációját XML-re „korlátozzuk", ekkor ugyanis még mindig lehetőségünk van a célunk elérése érdekében az XML sémanyelvek, az XML nyelvtanokat leíró típusnyelvek és a DTD-k (Document Type Definition) használatára. Ha ezek ismeretében is ragaszkodunk egy szabványos leírónyelvhez a RESTful szolgáltatásunkhoz, akkor a megoldást egy újabb betűszó nyújtja. A WADL (Web Application Description Language) főként arra lett tervezve, hogy gép által könnyen feldolgozható protokoll leíró formátumot szolgáltasson arra a célra, hogy ezek hatékonyan együtt tudjanak működni a különböző HTTP-alapú webalkalmazásokkal. Egy webalkalmazás leíró a legfontosabb elvárásoknak megfelelően a következő információkat tartalmazza [17]:

- Erőforrások listája: ez megfeleltethető egy olyan oldaltérképnek, amely a kínált erőforrásokról tájékoztat.
- Erőforrások közötti kapcsolatok: leírják az erőforrások közötti linkeket, mind okozati, mind tájékoztató jelleggel.
- Erőforrásokra alkalmazható metódusok: azok a HTTP metódusok, amelyek alkalmazhatók az erőforrásokra. Rövid ismertetőt tartalmaz a lehetséges bemenetekről, kimenetekről és a támogatott formátumokról.
- Erőforrás reprezentációs formátumok: a támogatott MIME típusok és az alkalmazott XML sémák.

Megjegyzendő, hogy a 2.0-ás verziószámtól kezdve bizonyos hiányosságokkal már a WSDL is támogatja a RESTful webszolgáltatásokat, habár azt is fontos megemlíteni, hogy sem a WADL, sem a WSDL 2.0 nem alkalmas hipermédia-rendszerek hatékony leírására, noha a REST-nek elvileg éppen ez az erőssége. Ebből az következik, hogy a

fejlesztők rá vannak kényszerítve az erőforrásokra vonatkozó terjedelmes dokumentációk írására, amely egyrészt nem sorolható a közkedvelt tevékenységek közé, másrészt meghosszabbítja a fejlesztés időtartamát. Ez egy nyilvánvaló példája annak, hogy a REST-nek vannak (még) hiányosságai, amelyeket remélhetőleg a közeljövőben pótolni fognak majd.

10.4 Mi a helyzet a funkcionalitással?

Volt már arról szó, hogy napjainkban a SOAP és a REST interfészek nagyjából ugyanazt a funkcionalitást nyújtják. Hogy a két technológia közül melyikre essen a választás egy konkrét projekt megvalósítása során, segít eldönteni a [18]-ban fellelhető következő táblázat.

	SOAP/WS-*	REST
Protocol invoking operations	SOAP	HTTP
Transport protocol	HTTP,TCP, others	HTTP
Data formats	XML	XML, JSON, others
Estabilishing a security context	WS-SecureConversation	SSL
Conveying security tokens	WS-Security	HTTP, SSL
Acquiring security tokens	WS-Trust	No standard
Language for describing interfaces	WSDL	No standard
Providing end-to-end reliability	WS-ReliableMessaging	No standard
Supporting distributed ACID transactions	WS-AtomicTransaction, WS-Coordination	No standard
Defining policy	WS-Policy	No standard
Acquiring interface definitions	WSMetadataExcahnge	No standard

Ha jobban megnézzük, a szabványok tekintetében szinte semmi közös nincs a két technológiában. A SOAP támogatói az adatcserén túl WS-* szabványok széles skálájára támaszkodnak, amely szabványok pontosan meghatározzák a különböző feladatköröket. Amíg például az azonosítással és az authentikációval kapcsolatos követelményeket a WS-Security specifikáció írja le, addig a megbízható üzenetküldésért a WS-ReliableMessaging vonható felelősségre, és teljesen más szabványokra épül az ACID (atomicity, consistency, isolation, durability) tranzakciók támogatása, hogy csak néhányat említsünk. (A REST sok esetben semmilyen szabványhoz nem ragaszkodik, jól mutatják

ezt a táblázat utolsó sorai) Felmerül azonban egy kérdés: szükségünk van egyáltalán a WS-* által nyújtott lehetőségekre, vagy megelégszünk a jóval egyszerűbb RESTful szemléletmóddal is? A nagyvállalati világban még mindig érdemesebb a SOAP szemléletmód támogatása mellett a szabványokhoz ragaszkodni, elősegítve ezzel a vállalatok közötti hatékony együttműködést, ugyanakkor érdemes a SOAP mellett dönteni abban az esetben is, ha elsősorban műveleteket és nem pedig erőforrásokat szeretnénk megosztani. Ezzel szemben a REST előnyösebb olyan szolgáltatások esetén, amelyeknél a cél adatok, erőforrások elérhetőségének megteremtése webes szolgáltatásokon keresztül.

11 Összegzés

A diplomamunka megírása során igyekeztünk átfogó, általános érvényűséggel ismertetni a Java-bázisú webszolgáltatásokat, kiemelt figyelmet fordítva a REST szoftver-architektúra stílusra, és az általa támogatott webszolgáltatásokra, melyek óriási figyelmet kaptak az elmúlt években. Mialatt a nagyvállalati rendszerek üzemeltetői kellő megelégedéssel használták a SOAP-ot a tőle elválaszthatatlan UDDI és WSDL párossal karöltve, addig ezzel párhuzamosan egyre több, a Restletet támogató Java-alapú platform jelent meg. Noha a SOAP/WS és a REST hívők közötti éles határvonal nyomatékosítja a két technológia közötti kontrasztot, egyértelműen nem lehet kijelenteni, hogy az egyik jobb lenne a másiknál.

12 KÖSZÖNETNYILVÁNÍTÁS

Köszönetet mondunk témavezető tanárunknak, Jeszenszky Péternek a diplomamunkánkhoz nyújtott segítségért. Köszönettel tartozunk továbbá Vajda Istvánnak [vajdyphoto.com], aki lehetővé tette számunkra, hogy a képgaléria program bemutatásához felhasználjuk fotóit. Külön köszönettel tartozunk szüleinknek, akik mindvégig mellettünk álltak, nélkülük ez a diplomamunka nem születhetett volna meg. Szeretnénk megköszönni továbbá Dr. Juhász István támogatását, akire egyetemi éveink alatt bármikor számíthattunk. Salamon Edit Anita külön köszönetet mond Pongor Gábornak, amiért a diplomamunka készítése során mindenben támogatta.

13 ÁBRAJEGYZÉK

14 IRODALOMJEGYZÉK

[1] Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, Peter Brittenham, Yuichi Nakamura, Paul Fremantle, Dieter Koenig, Claudia Zentner: *Java alapú web szolgáltatások*. Kiskapu kiadó, 2001.

[2] Leonard Richardson and Sam Ruby: *RESTful Web Services.* O'Reilly Media Inc, 2007.

[3] Roy Thomas Fielding: *Architectural Styles and the Design of Network-based Software Architectures*, Dissertation, 2000.

[4] Martin Kalin: *Java Web Services: Up and Running, 1st Edition*, O'Reilly Media Inc, 2009.

[5] Cool URIs don't change: http://www.w3.org/Provider/Style/URI

[6] What is Android: http://developer.android.com/guide/basics/what-is-android.html

[7] MD5: http://hu.wikipedia.org/wiki/MD5

[8] Secure Socket Layer (SSL): How It Works: http://www.verisign.com/ssl/ssl-information-center/how-ssl-security-works/index.html

[9] HTTP Secure: http://en.wikipedia.org/wiki/HTTP_Secure

[10] Eric Newcomer: *Understanding Web Services: XML, WSDL, SOAP, and UDDI,* Pearson Education Corporate Sales Division, 2002.

[11] Understanding WS-Security: http://msdn.microsoft.com/en-us/library/ms977327.aspx

[12] Alázza a Google az OpenID-t http://www.hwsw.hu/hirek/37284/google_openid_microsoft_yahoo_bejelentkezes_login_account_fiok.html

[13] OpenID and CardSpace: http://benlog.com/articles/2007/02/06/beamauth-two-factor-web-authentication-with-a-bookmark/

[14] Security and REST Web Services: http://2007.xtech.org/public/asset/attachment/76

[15] POM Reference: http://maven.apache.org/pom.html#What_is_the_POM

[16] Vajda István fotós blogja: http://vajdyphoto.com/

[17] Marc J. Hadley: *Web Application Description Language (WADL)*, Sun Microsystems Inc, 2006.

[18] 2009. Developer Summit: http://proceedings.esri.com/library/userconf/devsummit09/index.html

[19] Gottdank Tibor: *Webszolgáltatások XML alapú kommunikáció az Interneten*, ComputerBooks Kiadói Kft, 2005.

[20] Restlet 2.0 Tutorial: http://www.restlet.org/documentation/2.0/tutorial