Webové služby

17. listopadu 2020

World Wide Web (WWW) vznikl primárně jako nástroj pro sdílení a prohlížení vzájemně provázaných dokumentů. Jeho architektura se však ukázala jako nadčasová, a proto webové technologie plní dnes i další funkce zcela nezamýšlené při jeho vzniku. Jednou z těchto funkcí je propojení počítačových systémů, kdy WWW slouží jako platforma komunikaci typu machine-to-machine (M2M) a umožňuje snadno integrovat různé systémy do větších celků.

1 Úvod

World Wide Web byl postaven na dvou pilířích, (i) protokol HTTP určený pro přenos dat mezi serverem a klientem a (ii) jazyce HTML určeném k popisu (hyper)textových dokumentů, které jsou primárně určeny pro čtení lidmi. Změníme-li formát přenášených dat, což je velice jednoduché, je možné na této bázi vybudovat systém pro komunikaci mezi dvěma počítačovými systémy.

Software, který poskytuje své veřejné rozhraní pomocí webového serveru jiným zařízením, se označuje jako *webová služba*. Webové služby jsou poskytovány pomocí standardizovaného protokolu, což nabízí celou řadu zajímavých možností, které se ve velké míře dají použít v praxi.

1. Webové služby umožňují integrovat existující aplikace.
2. Jelikož jsou webové služby na hodně vysoké úrovni abstrakce a nejsou svázány s žádným konkrétním programovacím jazykem, je možné propojit aplikace psané v zcela odlišných jazycích.
3. Používání webových služeb předpokládá komunikaci přes jasně definované rozhraní, což umožňuje snadno programovat a integrovat aplikace, kde jednotlivé služby vytváří různí vývojáři (firmy).
4. Díky důrazu na programovaní proti rozhraní je možné pomocí webových služeb snadno rozdělit systém do izolovaných komponent. To je výhodné jak z pohledu návrhu programu tak z provozních důvodů, kdy je možné jednotlivé komponenty systému rozdělit přes víc oddělených počítačů.

Poskytování webových služeb se dnes stalo běžným standardem, jako příklady rozhraní uveďme třeba

- STAG https://stagservices.upol.cz/ws/web/,
- nebo služby Googlu https://developers.google.com/apis-explorer/.

Historicky vykrystalizovaly dva přístupy k tvorbě webových služeb. Jsou to (i) webové služby postavané na protokolu SOAP a (ii) architektura RESTful API. Oba dva přístupy si ukážeme na platformě Java EE.

2 Web services

V úvodní kapitole jsme mluvili o webových službách v širším významu. S označením *Webové služby (Web Services*) se dá setkat také ve specifickém případě, kdy označuje sadu standardů sloužích ke komunikaci typu klient-server. Pro rozlišení budeme v tomto případě používat původní anglické označení *Web services*.

Web services jsou určeny pro komunikaci typu klient-server, kdy klient posílá požadavky na server a ten vrací odpovídající výsledek. Tento přístup by se dal připodobnit ke vzdálenému volání procedur. Jednotlivé požadavky jsou serializovány do formátu SOAP[1] a jsou přenášeny pomocí protokolu HTTP(S)[2].

Součástí standardů pro Web Services je i jazyk WSDL (Web Service Description Language)[3], který slouží k strojově čitelnému popisu rozhraní. To na jedné straně umožňuje na straně serveru vygenerovat popis rozhraní v jazyce WSDL (např. z rozhraní v Javě) a na druhé straně to umožňuje vygenerovat rozhraní na straně klienta pro připojení k serveru.

Web services mají podporu[4] jak na platformě Java SE i Java EE.

2.1 Server

Vytvoření Web service je v Java EE velice jednoduché. Stačí označit třídu anotací `@WebService` a metody, které mají být součástí veřejného anotací `@WebMethod`, jak ukazuje následující kód.

```
@WebService(serviceName = "HelloService")
public class HelloService {

    @WebMethod
    public String hello(String name) {
        return "Hello " + name;
    }

    @WebMethod(operationName = "repeat")
    public String repeatText(
            @WebParam(name = "text") String text,
            @WebParam(name = "n") int count) {

        StringJoiner sj = new StringJoiner(", ");
        for (int i = 0; i < count; i++) {
            sj.add(text);
        }
        return sj.toString();
    }
}
```

[1] Simple Object Access Protocol, což je XML formát
[2] lze použít i další protokoly, např. SMTP, FTP, JMS
[3] opět XML formát
[4] označovanou jako JAX-WS

Web service pojmenovaná `HelloService` nabízí dvě jednoduché metody, jedna vrátí text rozšířený o slovo `"Hello "` a druhá zopakuje zadaný text n-krát. V případě druhé metody je ukázáno, že metoda je publikována pod jiným jménem a s jinými jmény argumentů.

Takto vytvořenou službu lze nalézt na adrese odpovídající jejímu jménu:
`http://localhost:8080/lecture06/HelloService`, kde se dá najít i WSDL soubor popisující dané rozhraní: `http://localhost:8080/lecture06/HelloService?wsdl`.

2.2 Klient

Pro přístup k Web service je potřeba vytvořit z WSDL souboru odpovídající rozhraní. K tomu slouží program `wsimport`, který by měl být součástí aplikačního serveru.

Následující příkazy vygenerují části kódu nutné pro připojení k Web service, konkrétně k demonstrační úloze a k rozhraní STAGu s informacemi o předmětech.

```
wsimport -keep 'http://localhost:8080/lecture06/HelloService?wsdl'
wsimport -keep 'https://stagservices.upol.cz/ws/services/soap/predmety?wsdl'
```

Argument `-keep` udává, že má generátor zachovat zdrojové kódy tříd, jinak ponechá pouze zkompilované třídy.

S Web service je možné pracovat dvěma způsoby. Nejjednodušší je nechat na aplikační serveru vytvoření odpovídajících objektů a jejich vložení do zadaného atributu třídy pomocí anotace `@WebServiceRef`.

```
@WebServiceRef(HelloService_Service.class)
private HelloService helloService;
```

Všimněte si, že atribut `helloService` pracuje s vygenerovaným rozhraním `HelloService` a v anotaci je použita vygenerovaná třída `HelloService_Service` s danou službou.

Práce s Web service je pak stejná jako s každým jiným objektem.

```
output = helloService.hello(text);
```

Alternativně můžeme vytvoření služby a odpovídajících objektů řešit explicitně, jak ukazuje následující kód.

```
URL wsdl = new URL("http://localhost:8080/lecture06/HelloService?wsdl");
QName serviceName = new QName("http://lecture06.pja.upol.cz/", "HelloService");

Service service = Service.create(wsdl, serviceName);
HelloService helloService = service.getPort(HelloService.class);

// příklad použití
output = helloService.repeat(text, 3);
```

Nejdříve vytvoříme objekty identifikující danou Web service, následně vytvoříme objekt třídy `Service` a z něj získáme objekt umožňující volat službu přes rozhraní `HelloService`. Toto alternativní řešení je vhodné pro situace, kdy nemáme k dispozici vhodný aplikační server, např. v Jave SE aplikacích.

3 RESTful API

Representational state transfer (REST) je architektura webových služeb, která staví na protokolu HTTP 1.1[5] a taktéž umožňuje komunikaci typu klient-server. Na rozdíl od Web services, které umožňují volat libovolné procedury, které jsou definované daným rozhraním, architektura REST předpokládá, že se manipuluje s objekty, kde každý objekt má jednoznačný identifikátor daný URI (např. `http://example.com/res/item/1`) a k manipulaci s objekty slouží sada standardních příkazů a odpovědí protokolu HTTP:

- `GET` – získání objektu,
- `POST` – vytvoření objektu,
- `PUT` – aktualizace objektu,
- `DELETE` – odstranění objektu.

Co jsou jednotlivé objekty není specifikováno, ale nejčastěji jsou objekty reprezentovány jako dokumenty ve formátu JSON nebo XML, ale může to být jiný vhodný formát např. CSV.

Architektura typu REST je z podstaty bezestavová a je žadoucí, aby komunikace byla i cachovatelná. Jednoduchost tohoto přístupu umožňuje snadné testování a debugování a je to jeden z důvodů, proč se architektura REST rozšířila a těší široké oblibě.

V Javě je základní podpora[6] na platformě Java EE, ale lze ji doplnit i do Java SE.

3.1 Server

Vytvoření REST API je v Java EE velice přímočaré. U tříd, které mají obstarávat veřejné rozhraní, doplníme anotaci `@Path` udávající, kterou část URL daná třída obsluhuje a u jednotlivých metod pomocí anotací `@GET`, `@POST` atd. vyznačíme na jaké akce reagují a případně doplníme další informace.

Následující třída představující jednoduchou službu nabízející dvě akce typu GET.

```
@Path("/demo")
public class RestDemo {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String helloWorld() {
       return "Hello world";
    }
```

[5] a novějších
[6] označovaná jako JAX-RS

```
    @GET
    @Path("hello/{name}")
    @Produces(MediaType.TEXT_PLAIN)
    public String helloWorldWithName(@PathParam("name") String name) {
       return "Hello " + name;
    }
}
```

Aby tuto třídu a její rozhraní bylo možné použít, je potřeba ještě vytvořit konfigurační třídu dědící z `javax.ws.rs.core.Application` a u ní nastavit cestu, kde bude REST API k dispozici (anotace `@ApplicationPath`) a jednotlivé třídy poskytující rozhraní zaregistrovat, viz přiložený kód.

U služby je možné použít dvě URL:

- http://localhost:8080/lecture06/rest/demo/ – vracející text "Hello world",
- http://localhost:8080/lecture06/rest/demo/hello/*name* – vracející text "Hello *name*".

Vyzkoušet si toto rozhraní můžete s libovolným nástrojem, který podporuje komunikaci protokolem HTTP. Nejčastěji se používá příkaz `curl`.

```
curl -X GET 'http://localhost:8080/lecture06/rest/demo/'
curl -X GET 'http://localhost:8080/lecture06/rest/demo/hello/world'
```

Vrátíme-li se ke kódu třídy, máme v ní ještě několik anotací, které stojí za vysvětlení. Anotaci `@Path` můžeme použít společně s danou metodou a specifikovat tak další úroveň cesty. Do cesty můžeme vložit i parametry pro danou metodu. Ty pak vyznačíme pomocí anotace `@PathParam`. Pokud bychom parametry chtěli předávat jako tradiční parametr v URL, použili bychom `@QueryParam`.

Anotací `@Produces` lze specifikovat formát dat. V našem případě je použit formát *text/plain*, ale jde pochopitelně použít i další formáty.

V přiložených zdrojových kódech je ukázán složitější příklad představující databázi uživatelů, kde každý uživatel má přiřazený svůj číselný identifikátor, jméno a příjmení. Ukažme si z této třídy dvě zajímavější metody.

První z nich je metoda `get`, která vrací informace o uživateli s daným id.

```
@GET
@Path("{id}")
public synchronized Response get(@PathParam("id") int id) {
    if (!users.containsKey(id))
        return Response.status(Response.Status.NOT_FOUND).build();
    return Response.ok(Collections.singleton(users.get(id)), MediaType.APPLICATION_JSON)
                   .build();
}
```

V této metodě nevracíme rovnou obsah, ale objekt třídy `Response`, který nám umožňuje specifikovat chybový stav (404 Not Found), pokud daný uživatel neexistuje. V případě, že daný uživatel existuje, je vrácen status (200 Ok) a objekt představující uživatele je serializován do formátu JSON.

Druhá metoda, která stojí za zmínku, je metoda `create`, která vytvoří uživatele.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public synchronized Response create(User user) {
    if (users.containsKey(user.getId()))
        return Response.status(Response.Status.CONFLICT).build();
    users.put(user.getId(), user);
    return Response.ok().build();
}
```

Této metodě je jako argument předáván dokument ve formátu JSON (viz anotace `@Consumes`), který je přes JSON Binding[7] konvertován na objekt třídy `User`.

Vyykoušejte si následující příkazy:

```
curl -X GET 'http://localhost:8080/lecture06/rest/user/list'
curl -X GET 'http://localhost:8080/lecture06/rest/user/1/'
curl -X GET 'http://localhost:8080/lecture06/rest/user/0/'
curl -X POST -H "Content-Type: application/json" -d \
  '{"id": 10, "name": "john", "surname": "doe"}' 'http://localhost:8080/lecture06/rest/user/'
curl -X DELETE 'http://localhost:8080/lecture06/rest/user/1/'
```

3.2 Klient

Klient pro práci s REST API je postaven s velké části na návrhovém vzoru Builder, což umožňuje velice pohodlné sestavení požadavku.

Ukážeme si to na příkladu přístupu k rozhraní vytvořeném v předchozí podkapitole.

```
Client client = ClientBuilder.newClient();
Response response = client.target("http://localhost:8080/lecture06/rest/demo/hello")
        .path(input)
        .request()
        .get();
output = response.readEntity(String.class);
```

Nejdříve vytvoříme klienta. Následně sestavíme požadavek, určíme cíl, se kterým budeme komunikovat a dále sestavíme požadavek. Zde je možné specifikovat další vlastnosti požadavku, např. metoda `path` nám umožňuje vložit parametr jako součást cesty[8], metoda `request` vytvoří požadavek, který je možné zavoláním příslušné metody (`get()`, `post()` atd.) vyvolat.

[7] https://javaee.github.io/jsonb-spec/

[8] Pokud bychom chtěli předat standardní parametr URL slouží k tomu metoda `queryParam`

S objektem třídy `Response` je možné dále pracovat. Jednak můžeme získat stavové informace, např. návratový kód nebo formát dat. Můžeme ale i data převést na odpovídající objekty. K tomu slouží metoda `readEntity`.

V tomto případě jsme s vráceným objektem naložili jako s řetezcem. Pokud by byl vrácen objekt odpovídající například třídě `User` je možné uvést `User.class` a JSON binding se postará o konverzi.

V případě kolekcí jako je seznam, můžeme buď použít generické třídy typu `JsonArray`, kdy data převedeme na obecnou strukturu bez bližší typové informace, nebo můžeme použít třídu `GenericType` k určení typu, jak ukazuje následující příklad.

```
Response response = client.target("http://localhost:8080/lecture06/rest/user/list")
        .request()
        .get();
List<User> result = response.readEntity(new GenericType<List<User>> () {});
```

Závěr

V tomto semináři jsme si představili dva různé přístupy k tvorbě webových služeb. První, technicky propracovanější ale mnohem složitejší řešení, jde nalézt zejména v systémech určených pro velké korporace. Druhé, jednodušší a agilnější řešení, je však mezi programátory rozšířenější, protože práce s ním je intuitivnější a pohodlnější. V tomto semináři jsme si představili jen základní úvod do problematiky, proto doporučuji p.t. čtenářům, aby se na problematiku a zejména související fenomény jako je dokumentace API, vyhledávání služeb apod. podívali podrobněji v rámci sebevzdělávání.