

4. Anotácie JAPI pre pokroilé mapovanie na objekty D2000

- 4.1. Mapovanie typu D2000 Unival
 - 4.1.1 Mapovanie hodnoty a platnosti hodnoty unival-u
 - 4.1.2. Mapovanie unival atribútu
- 4.2. Mapovanie štruktúrovaných Unival hodnôt
 - 4.2.1. Príklad s vysvetlivkami
 - 4.2.2. Priame použitie triedy UnivalConvertor pre využitie mapovania
- 4.3. Definícia mapovania RPC volaní
 - 4.3.1. Zoznam parametrov pre volanie RPC
 - 4.3.2. Volanie z JAPI do ESL – príklad s komentármi
 - 4.3.2.1 Jednoduché mapovanie vstupného a výstupného parametra
 - 4.3.2.2. Mapovanie vstupno-výstupného parametra
 - 4.3.2.3. Mapovanie synchronnej procedúry s odloženým vyhodnotením výsledku
 - 4.3.2.4. Mapovanie parametra so štruktúrovaným typom
 - 4.3.2.5. Mapovanie asynchrónneho volania a asovej znaky hodnoty
 - 4.3.2.6. Mapovanie procedúry s dvomi výstupnými štruktúrovanými parametrami
 - 4.3.3. Volanie z ESL do JAPI – príklad s komentármi
 - 4.3.4. Použitie anotácie CallerInformation
- 4.4. Použitie anotovaných objektov
 - 4.4.1. Príklad volania z JAPI do ESL
 - 4.4.2. Príklad volania z ESL do JAPI

Vo verzii 10.1.39-build4 pribudla do knižnice JAPI možnosť aplikane definovať objekty, ktoré výrazným spôsobom prispievajú k typovo bezpečnej komunikácii pri RPC volaniach. Balík `sk.ipesoft.d2000.d2japi.annotations` obsahuje triedy a anotácie pre definovanie mapovania.

Mapovanie anotáciami má dve hlavné asti:

- Mapovanie štruktúrovaných *unival* hodnôt na objekty typu `java.util.List` a späť. Prvky zoznamu sú objekty zodpovedajúce JavaBean konvencii.
- Mapovanie RPC volaní pomocou tzv. *EventProxy* a *EventHandler* objektov.

Obidve asti využívajú princípy mapovania D2000 *Unival*-u popísané v nasledujúcej asti.

4.1. Mapovanie typu D2000 *Unival*

Transformácie hodnôt typu *unival* na základné Java typy a späť je jedným zo základných kameov pokroíleho mapovania.

Unival hodnota je kompozitná, skladá sa z hodnoty a viacerých atribútov. V mnohých prípadoch je však zaujímavá len hodnota samotná, prípadne indikátor jej platnosti, ale len málokedy ďalšie atribúty. Hodnota a jednotlivé atribúty sa preto mapujú na objekty základných Java typov každý zvlášť.

4.1.1 Mapovanie hodnoty a platnosti hodnoty *unival*-u

Atribút platnosti hodnoty a hodnota samotná sú vzájomne veľmi silne previazané. *Unival* bu obsahuje platnú hodnotu, alebo má hodnotu neplatnú a vtedy sa správa, ako keby hodnotu vôbec nemal. Z toho dôvodu sú hodnota a atribút platnosti hodnoty mapované spoločne na objektové typy. Napríklad celočíselná hodnota je mapovaná typom `Integer`. Ak je hodnota *unival*-u platná, mapuje sa na hodnotu, ak je neplatná, mapuje sa ako `null`.

Pre mapovanie hodnoty *unival*-u slúžia anotácie `ParameterValue`, `ReturnValue` a `ColumnValue`. Ich spoločným znakom je, že sú priamo i nepriamo parametrizované typom hodnoty *unival*-u a tomuto typu musí zodpovedať typ anotovaného objektu.

Nasledujúca tabuľka uvádza podporované transformácie:

D2000 typ	Java typ
Logický (BOOL)	VBool, Boolean ¹
Celočíselný (INT)	Integer
Reálny (REAL)	Double
Absolútny as (TIME)	Date, Long ²
Relatívny as (REAL)	Double
Text (TEXT)	String
Štruktúrovaný (RECORD)	List<T> ³

4.1.2. Mapovanie *unival* atribútu

Pre mapovanie *unival* atribútov slúžia anotácie `ParameterAttribute`, `ReturnAttribute`, `ColumnAttribute`. Ich spoloným znakom je, že sú parametrizované typom *unival* atribútu, na ktorý definujú väzbu. Rovnako ako hodnota *unival*-u, aj atribúty sú mapované výlune na objektové typy. Avšak pokus o transformáciu prázdneho odkazu (null) na hodnotu *unival* atribútu je vyhodnotený ako chyba. Nasledujúca tabuika obsahuje zoznam podporovaných transformácií:

Unival atribút	Java typ
asová znaka hodnoty (\TIM)	Date, Long

Zoznam podporovaných atribútov je krátky, obsahuje len mapovania, ktoré boli doteraz potrebné. V budúcnosti bude tento zoznam rozšírený na požiadanie.

4.2. Mapovanie štruktúrovaných *Unival* hodnôt

Štruktúrované *Unival* hodnoty sú štandardným spôsobom reprezentované typom `UnivalRecord`, ktorého hodnotová as obsahuje objekt typu `sk.ipesoft.d2000.datatable.Table`. Pre mnohé aplikácie je však výhodnejšie pracovať s jednoduchšou reprezentáciou, v ktorej je štruktúrovaná hodnota reprezentovaná objektom typu `java.util.List`, ktorého prvky reprezentujú jednotlivé riadky. V jednom zozname sú všetky prvky objektmi tej istej aplikácie definovanej triedy, ktorá je anotáciami mapovaná na bunky štruktúrovanej hodnoty.

4.2.1. Príklad s vysvetlivkami

Na obrázku je uvedená definícia štruktúry *SD.Person*, ktorá posluží ako príklad. Nasleduje mapovanie definície štruktúry na triedu `Person`.

Index	Meno	Popis	Typ	Štartovacia hodnota
1	Id		Celočíselný	Vypnuté
2	Name		Text	Vypnuté
3	Active		Logický	Vypnuté

```
import sk.ipesoft.d2000.d2japi.annotations.UnivalAttributeType;
import sk.ipesoft.d2000.d2japi.annotations.UnivalConvertor;
import sk.ipesoft.d2000.d2japi.annotations.structureBinding.ColumnAttribute;
import sk.ipesoft.d2000.d2japi.annotations.structureBinding.ColumnValue;
import sk.ipesoft.d2000.d2japi.annotations.structureBinding.ConvertedExceptionValue;
import sk.ipesoft.d2000.d2japi.annotations.structureBinding.MulticonvertedColumnValue;
import sk.ipesoft.d2000.d2japi.annotations.structureBinding.StructureDefinition;
import sk.ipesoft.d2000.d2japi.sharedResources.ConversionResult;
import sk.ipesoft.d2000.datatable.ColumnType;
```

```
@StructureDefinition(name = "SD.Person")
public class Person {
    private ConversionResult convertedName;
    private Integer id;
    private List<ConversionResult> multiConvertedName;
    private String name;
    private Long nameTime;

    public Person() {
        this.id = null;
        this.name = null;
        this.nameTime = 0L;
        this.convertedName = null;
        this.multiConvertedName = Collections.emptyList();
    }

    public Person(Integer id, String name, Long nameTime) {
        this.setId(id);
        this.setName(name);
        this.setNameTime(nameTime);
    }

    public ConversionResult getConvertedName() {
        return convertedName;
    }
}
```

```

    }

    @ConvertedColumnValue(name = "Name")
    public void setConvertedName(ConversionResult convertedName) {
        this.convertedName = convertedName;
    }

    @ColumnValue(name = "Id", columnType = ColumnType.integer)
    public Integer getId() {
        return id;
    }

    public final void setId(Integer id) {
        this.id = id;
    }

    public List<ConversionResult> getMultiConvertedName() {
        return multiConvertedName;
    }

    @MulticonvertedColumnValue(name = "Name")
    public void setMultiConvertedName(List<ConversionResult> multiConvertedName) {
        this.multiConvertedName = multiConvertedName;
    }

    @ColumnValue(name = "Name", columnType = ColumnType.text)
    public String getName() {
        return name;
    }

    public final void setName(String name) {
        this.name = name;
    }

    @ColumnAttribute(name = "Name", attribute = UnivalAttributeType.valueTime)
    public Long getNameTime() {
        return nameTime;
    }

    public final void setNameTime(Long nameTime) {
        if (nameTime == null) {
            throw new NullPointerException("nameTime is null");
        }
        this.nameTime = nameTime;
    }
}

```

Trieda Person je anotáciou @StructureDefinition označená pre mapovanie štruktúrovanej hodnoty. Hodnota atribútu anotácie name indikuje väzbu na objekt SD.Person.

Anotácia @ColumnValue na metóde getName indikuje, že metóda slúži ako *getter* hodnoty v bunke stpca Name (poda atribútu anotácie name = "Name") a že sú hodnoty v tomto stpci textového typu (poda atribútu anotácie columnType = ColumnType.text). JAPI bude pri spracovaní tejto anotácie vyžadovať prítomnosť setter metódy setName a tiež bude oakáva, že *property* Name⁴ je typu String. Obdobným spôsobom funguje anotácia @ColumnValue na metóde getId.

Anotácia @ColumnAttribute na metóde getNameTime indikuje, že metóda slúži ako *getter* asovej znaky hodnoty (poda atribútu anotácie attribute = UnivalAttributeType.valueTime) v bunke stpca Name (poda atribútu anotácie name = "Name"). JAPI bude pri spracovaní tejto anotácie vyžadovať prítomnosť setter metódy setNameTime a tiež bude oakáva, že *property* NameTime⁴ je typu Date alebo Long. Pri konverzii na *unival* by bolo považované za chybu, ak by bola hodnota NameTime rovná null, preto metóda setNameTime takúto hodnotu odmietne nastaviť.

Anotácia @ConvertedColumnValue na metóde setConvertedName indikuje, že metóda slúži ako setter formátovanej hodnoty v bunke stpca Name. Ide o formátovanie hodnoty pre zobrazenie na používateľskom rozhraní.

Anotácia @MulticonvertedColumnValue na metóde setMultiConvertedName indikuje, že metóda slúži ako setter formátovanej hodnoty v bunke stpca Name pre všetky dostupné jazykové mutácie.

Všimnite si, že trieda Person neobsahuje mapovanie buniek stpca Active. Pri mapovaní štruktúrovaných *unival* hodnôt je mapovanie jednotlivých stpcov nepovinné. Pri mapovaní z *unival* u sú hodnoty bez mapovania ignorované. Pri mapovaní do *unival* u sú hodnoty bez mapovania nahradené neplatnými hodnotami. Chýbajúce asové znaky sú doplnené aktuálnym asom.

4.2.2. Priame použitie triedy UnivalConverter pre využitie mapovania

So štruktúrovanými hodnotami sa v prostredí *JAPI* stretávame na dvoch miestach – parametre RPC volaní a hodnoty objektov typu štruktúrovaná premenná. Ak sú pre RPC použité pokroilé anotácie, štruktúrované hodnoty sú konvertované podľa mapovania automaticky. Inak je možné konvertovať medzi typom *UnivalRecord* a mapovanou triedou *rune*, za pomoci objektu triedy *UnivalConverter*.

Každá inštancia triedy *UnivalConverter* (v nasledujúcej kapitole oznaovaná ako *converter*) potrebuje pre svoju prácu aktuálne informácie o definíciách štruktúr, ktoré pripojený *JConnector* sprístupuje objektom typu *D2StructureDefinitionResolver*. Na väčšinu praktických úloh je plne vyhovujúci tzv. *defaultUnivalConverter*.

```
UnivalConverter converter = session.getConnector().getDefaultUnivalConverter();
```

V prípade, že by bolo žiaduce pracovať s viacerými rôznymi inštanciami, pričom každá z inštancií by poznala len vybrané mapovania, je možné získať novú inštanciu nasledovným spôsobom:

```
UnivalConverter converter = session.getConnector().createUnivalConverter();
```

Z optimalizačných dôvodov je možné vykonať registráciu mapovacích tried *rune*, v inicializovanej aplikácii, pred prvým skutočným konvertovaním štruktúrovaných hodnôt. V tomto kroku dochádza k analýze a vyhodnoteniu správnosti použitia mapovacích anotácií. Mapovanie však nie je kontrolované voči aktuálnej definícii štruktúry v systéme *D2000*.

```
converter.registerStructure(Person.class);
```

Explicitná registrácia je však nepovinná, *converter* ju v prípade potreby vykoná automaticky, pred prvou konverziou štruktúrovanej hodnoty.

```
List<Vector> list = converter.decodeStructure(record, Vector.class);
UnivalRecord<?, ?, ?> data = converter.encodeStructure(list, Vector.class);
```

Formátovanie hodnôt jednotlivých buniek pre zobrazenie na používateľskom rozhraní v konkrétnom jazyku je možné vykonať po dekódovaní hodnôt následným volaním:

```
Converter dictionaryConverter = session.createDictionaryConverter();
converter.fillConvertedValues(list, Person.class, dictionaryConverter);
```

Alebo pre konverziu do všetkých jazykov:

```
List<Converter> dictionaryConvertors = new ArrayList<>();
for (DictionaryLanguage language : session.getConnector()
    .getSharedResourcesCache().getDictionaryLanguages())
    dictionaryConvertors.add(
        session.getConnector().createDictionaryConverter(language.getIndex()));
converter.fillMulticonvertedValues(list, Person.class, dictionaryConvertors);
```

4.3. Definícia mapovania RPC volaní

Podľa rôznych kategórií je možné RPC volania deliť nasledovne.

Podľa toho, kto je volajúci a kto volaný:

- Odchádzajúce RPC – keď *JAPI* klient volá RPC implementovanú niekde inde (napr. v *ESL*). Volajúci zadáva parametre a po skončení RPC získava výsledok.
- Prichádzajúce RPC – keď *JAPI* klient vystaví nejakú RPC a niekto iný ju chce zavolať. Volaný kód spracuje parametre, vykoná určitú rutinu a odovzdá výsledok.

S využitím *JAPI* knižnice je možné vytvárať odchádzajúce RPC aj prijímať prichádzajúce RPC. Spôsob realizácie je však pomerne odlišný, preto je každej kategórii venovaná samostatná kapitola.

Podľa toho, či sa aká na výsledok volania:

- Synchronné volanie – Volajúci čaká, pokiaľ volaný kód neskoní vykonávanie a nevráti výsledok.
- Asynchronné volanie – Volajúci odošle príkaz na vykonanie zvolenej RPC aj s parametrami, ale neaká na jej skončenie. Volaný o výsledku vôbec neinformuje.

V prostredí ESL a v prostredí internej Javy je synchrónne aj asynchrónne RPC nasmerované do tej istej procedúry (metódy). Spôsob volania vyberá volajúci na základe dohody, lebo volaný kód toto nevie ovplyvni ani zisti. V prostredí *JAPI* je vo volanom kóde RPC vidie, i bol zavolaný synchrónne alebo asynchrónne. Pokroilými anotáciami sa dVolajúci okonca volaný kód oznaí tak, aby spracúval iba jeden konkrétny z dvoch typov volania.

4.3.1. Zoznam parametrov pre volanie RPC

Volanie RPC je systémom D2000 prenášané ako správa zložená z viacerých astí:

- adresa volajúceho
 - HOBj procesu (proces typu Event Handler, HI, *Session*),
 - dynamické HOBj vykonávaného objektu (objekt typu Event, Schéma alebo 0 ak je volajúci *Session*),
 - príznak, i RPC volanie pochádza z prostredia interného Java run-time,
- adresa volaného
 - HOBj procesu (proces typu Event Handler, HI, *Session*),
 - HOBj vykonávaného objektu (objekt typu Event, Schéma alebo 0 ak je volajúci *Session*),
 - príznak, i RPC volanie smeruje do prostredia interného Java run-time,
 - íslo inštancie vykonávaného objektu (ak ide o „inštanu" vytvorený Event alebo Schému a HOBj objektu bolo bázové, inak 0),
- identifikátor volanej RPC
 - meno
 - HOBj objektu ESL Interface, ak ide o implementáciu RPC definovanej v ESL Interface, inak 0,
- hodnoty parametrov volania,
- príznak, i ide o synchrónne⁵ alebo asynchrónne⁶ volanie.

Klasický spôsob volania RPC prostredníctvom JAPI vyžadoval, aby volajúci pri každom volaní uviedol všetky položky okrem adresy volajúceho (ktorú doplnil *D2Connector*). Nevýhodou tohto prístupu je predovšetkým chýbajúca typová kontrola hodnôt parametrov volania ako aj ich práčne vytváranie. Nepohodlná je tiež nutnosť získa a uklada HOBj objektov a procesov.

4.3.2. Volanie z *JAPI* do ESL – príklad s komentármi

```
RPC PROCEDURE Parse(IN TEXT _text, INT _result)
  _result := %StrToI(_text)
END Parse

RPC PROCEDURE Square(REAL _value)
  _value := _value * _value
END Square

RPC PROCEDURE Redim(IN INT _n, RECORD NOALIAS(SD.Person) _data)
  REDIM _data[_n]
END Redim

RPC PROCEDURE SetValue(IN TIME _value)
  U.Value := _value TIME _value\TIM
END SetValue

RPC PROCEDURE GetDataAndMetadata(RECORD NOALIAS(SD.Person) _data,
                                  RECORD NOALIAS(SD.Metadata) _metadata)
  ...
END Redim

BEGIN
END
```

Uvedený ESL kód predstavuje zdrojový kód server event-u E.Service, ktorého rodi je proces SELF.EVH. Väšina procedúr má pre ilustratívny implementované jednoduché telo. RPC *GetDataAndMetadata* telo nemá, pretože by bolo príliš zložitú. Ilustruje však príklad procedúry, ktorá má viac ako jeden výstupný parameter⁷.

V nasledujúcich podkapitolách bude postupne po astiach uvedený zdrojový kód, ktorý by za normálnych okolností tvoril jeden súbor.

4.3.2.1 Jednoduché mapovanie vstupného a výstupného parametra

```

import java.util.Date;
import java.util.List;
import sk.ipesoft.d2000.d2japi.annotations.InOut;
import sk.ipesoft.d2000.d2japi.annotations.ParameterDirectionType;
import sk.ipesoft.d2000.d2japi.annotations.ParameterType;
import sk.ipesoft.d2000.d2japi.annotations.UnivalAttributeType;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.Event;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.Parameter;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.ParameterAttribute;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.ParameterValue;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.RPC;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.ReturnValue;

@Event(name = "E.Service")
public interface Service
{
    @RPC(name = "Parse",
        asynchronous = false,
        parameters =
        {
            @Parameter(name = "value", type = ParameterType.text, inOut = false),
            @Parameter(name = "result", type = ParameterType.integer, inOut = true)
        })
    @ReturnValue(name = "result")
    public Integer parse(
        @ParameterValue(name = "value") String value);
}

```

Anotácia `@Event` indikuje, že rozhranie `Service` slúži na mapovanie RPC volaní z *JAPI*. Nepovinný parameter `name` v tomto prípade indikuje, že je mapovaný objekt `E.Service`.

Rozhranie má jednu metódu – `parse`. Anotácia `@RPC` na tejto metóde indikuje, že volanie tejto metódy má by premenené na odchádzajúce RPC volanie. Parametre anotácie majú nasledovný význam:

- `name = "Parse"` definuje identifikátor volanej RPC, ktorý musí by zhodný s identifikátorom v ESL.
- `asynchronous = false` definuje, že volanie má prebehnú synchronne a teda, že sa po zavolaní má poka na výsledok.
- `parameters = { ... }` definuje zoznam „formálnych“ parametrov RPC. Tento zoznam musí v presnom poradí uvádzať, aké parametre sú v ESL deklarované.
 - `name` lokálna identifikácia parametra. Tento identifikátor sa môže líšiť s názvom parametra v ESL, lebo rozsah platnosti tohto identifikátora je len v rámci mapovania tejto RPC. (Odkazujú sa na alšie anotácie, ale inde sa nepoužíva.)
 - `type` deklarovaný typ hodnoty parametra.
 - `inOut = false`, resp. `inOut = true` indikuje, či je parameter deklarovaný ako vstupný (v ESL je pred typom kúové slovo `IN`) alebo vstupno-výstupný a teda jeho výsledná hodnota bude prenesená naspäť.

Anotácia `@ReturnValue` slúži na previazanie návratovej hodnoty metódy s návratovou hodnotou niektorého z parametrov. Parameter anotácie `name = "result"` indikuje, že to má by návratová hodnota 2. parametra zo zoznamu formálnych parametrov kvôli zhode v identifikátoroch. Typ návratovej hodnoty `Integer` zodpovedá typu formálneho parametra podľa tabučky v kapitole 4.1.1. Dôležitou podmienkou je, aby bol formálny parameter oznaený ako vstupno-výstupný.

Anotácia 1. parametra metódy `@ParameterValue` indikuje, že hodnota tohto parametra má by premenená na vstupnú hodnotu 1. formálneho parametra (podľa parametra anotácie `name = "value"`). Typ hodnoty `String` zodpovedá typu formálneho parametra podľa tabučky v kapitole 4.1.1.

Zaujímavý je tiež fakt, že pre 2. formálny parameter (`result`) nie je definované mapovanie vstupnej hodnoty. V takomto prípade je pri volaní RPC procedúry nastavený 2. parameter na neplatnú (invalidnú) hodnotu.

4.3.2.2. Mapovanie vstupno-výstupného parametra

```

@RPC(name = "Square",
    parameters =
    {
        @Parameter(name = "v", type = ParameterType.real)
    })
@ReturnValue(name = "v")
public Double square_1(
    @ParameterValue(name = "v") Double value);

@RPC(name = "Square",
    parameters =
    {
        @Parameter(name = "v", type = ParameterType.real)
    })
public void square_2(
    @ParameterValue(name = "v") InOut<Double> value);

```

Metódy `square_1` a `square_2` mapujú obidve tú istú RPC (je to dovolené). V porovnaní s predošlým príkladom je však niekoľko odlišností v použití anotácie `@RPC` a `@Parameter`:

- parameter `RPC.asynchronous` nie je uvedený, lebo je nepovinný a v takom prípade nadobúda automaticky hodnotu `false`.
- parameter `Parameter.inOut` nie je uvedený, lebo je nepovinný a v takom prípade nadobúda automaticky hodnotu `true`.

Metódy `square_1` a `square_2` sa vzájomne odlišujú spôsobom, akým mapujú parameter volania:

- `square_1` rozdeľuje vstupnú a výstupnú čas parametra RPC procedúry na vstupný parameter a návratovú hodnotu metódy.
- `square_2` ponecháva vstupnú aj výstupnú čas parametra RPC procedúry v parametri metódy. Pretože Java ako jazyk nemá vstupno-výstupné parametre, parameter metódy `value` je deklarovaný s typom `InOut<Double>`.

Knižnica *JAPI* použije parameter metódy ako vstupno-výstupný (pri volaní preíde jeho vstupnú hodnotu a po skončení nastaví jeho výstupnú hodnotu) vtedy, ak sú splnené nasledovné podmienky.

- formálny parameter RPC procedúry je definovaný ako vstupno-výstupný
- parameter metódy je deklarovaný s typom `InOut<>`. Generický parameter typu musí zodpovedať typu formálneho parametra.
- nepovinný parameter anotácie `@ParameterValue.direction` nebol uvedený, iže automaticky nadobudol hodnotu `ParameterDirectionType.e.derived`. (Rovnaké správanie by bolo dosiahnuté, keby bol explicitne nastavený na hodnotu `derived` alebo `inout`.)

4.3.2.3. Mapovanie synchrónnej procedúry s odloženým vyhodnotením výsledku

```

RPC(name = "Square",
    parameters =
    {
        @Parameter(name = "v", type = ParameterType.real)
    })
@ReturnValue(name = "v")
public Future<Double> square_3(
    @ParameterValue(name = "v") Double value);

```

Metóda `square_3` sa od metódy `square_1` odlišuje typom návratovej hodnoty. Ke je typ návratovej hodnoty deklarovaný ako `Future`, volanie metódy nie je blokujúce, tak ako pri bežnej synchrónnej RPC, ale skoní hne. Výsledok je uložený vo `Future` objekte, ktorého `.get()` metóda zabezpečí synchrónnos volania.

4.3.2.4. Mapovanie parametra so štruktúrovaným typom

```

@RPC(name = "Redim",
    parameters =
    {
        @Parameter(name = "n", type = ParameterType.integer, inOut = false),
        @Parameter(name = "r", type = ParameterType.record, recordType = Person.class)
    })
@ReturnValue(name = "r")
public List<Person> redim(
    @ParameterValue(name = "n") Integer length,
    @ParameterValue(name = "r") List<Person> data);

```

Metóda `redim` mapuje RPC, ktorej druhým formálnym parametrom je štruktúrovaná hodnota. Pre správne mapovanie je typ parametra v anotácii nastavený na `@Parameter.type = ParameterType.record` a nepovinný parameter anotácie `@Parameter.recordType = Person.class`. Použitá definícia štruktúry `SD.Person` a jej mapovanie triedou `Person` pochádza z kapitoly 5.2.1.

Pri použití metódy `redim` bude pre úely návratovej hodnoty vytvorená nová inštancia typu `List<Person>` a jej prvky budú nové inštancie typu `Person` napriek tomu, že majú rovnaký obsah ako inštancie, ktoré boli použité ako parametre.

4.3.2.5. Mapovanie asynchrónneho volania a asovej znaky hodnoty

```
@RPC(name = "SetValue",
      asynchronous = true,
      parameters =
      {
          @Parameter(name = "v", type = ParameterType.time, inOut = false)
      })
public void setValue(
    @ParameterValue(name = "v") Date value,
    @ParameterAttribute(name = "v", attributeType = UnivalAttributeType.valueTime)
    Long valueTime);
```

Metóda `setValue` sa od predošlých príkladov odlišuje tým, že mapuje asynchrónne volanie (`RPC.asynchronous = true`). To znamená, že z pohľadu používateľa skoní jej volanie okamžite (po odoslaní správy) a neaká sa na doručenie odpovede o výsledku. Preto tiež asynchrónna metóda nemôže pristupovať ku výstupnejasti parametrov volania a teda nemôže použiť anotáciu `@ReturnValue` alebo `@ReturnAttribute` a takisto nemôže nastaviť parametre anotácií `ParameterValue.direction` a `ParameterAttribute.direction` na hodnoty `inout` alebo `out`.

Nová anotácia `@ParameterAttribute` mapuje vybraný *unival* atribút – v tomto prípade asovú znaku hodnoty (`attributeType = UnivalAttributeType.valueTime`). Pozor, nie je dovolené zavolať metódu s `null` hodnotou tohto parametra – mapovanie `null` na atribút *univalu* nie je možné.

V príklade je zámerne použitý iný typ parametra `value` (`java.util.Date`) a parametra `valueTime` (`java.lang.Long`). Obidva typy je možné rovnako dobre použiť na mapovanie hodnoty vyjadrujúcej „D2000 absolútny as“, ich výber je na tvorcovi mapovania.

4.3.2.6. Mapovanie procedúry s dvomi výstupnými štruktúrovanými parametrami

```
@RPC(name = "GetDataAndMetadata",
      asynchronous = false,
      parameters =
      {
          @Parameter(
              name = "data",
              inOut = true,
              type = ParameterType.record,
              recordType = Person.class),
          @Parameter(
              name = "meta",
              inOut = true,
              type = ParameterType.record,
              recordType = Metadata.class)})
public void getDataAndMetadata(
    @ParameterValue(name = "data") InOut<List<Person>> data,
    @ParameterValue(name = "meta") InOut<List<Metadata>> meta);
```

Metóda `getDataAndMetadata` mapuje procedúru s dvomi vstupno-výstupnými parametrami (2 parametre s `inOut = true`). V porovnaní s predošlými metódami stojí za povšimnutie:

- Typ parametra `data` je `InOut<List<Person>>`. Generickým parametrom kontajnera `InOut` je generický `List`, ktorého parametrom je trieda `Person`, ktorej mapovanie na `SD.Person` je popísané v kapitole 4.2.1.
- Trieda `Metadata` mapuje definíciu štruktúry `SD.Metadata`. Jej definíciu v príkladoch neuvádzame.
- Inštancia objektu typu `List`, ako aj inštancie predstavujúce jeho prvky, ktoré sa nachádzajú v kontajneri typu `InOut` po skončení synchrónneho volania sú vždy iné, ako inštancie, ktoré sa v kontajneri nachádzali na začiatku volania. Toto správanie je vlastnosť knižnice *JAPI*.

4.3.3. Volanie z ESL do JAPI – príklad s komentármi

V nasledujúcej ukážke bude uvedený zdrojový kód triedy `ExampleHandler`. Je to aplikane definovaná trieda, ktorej metódy môžu byť volané ako RPC z prostredia ESL.


```

import sk.ipesoft.d2000.d2japi.annotations.ParameterType;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.Parameter;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.ParameterValue;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.RPC;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.ReturnValue;

public class ExampleHandler
{
    @RPC(name = "Parse",
        parameters =
        {
            @Parameter(name = "text", type = ParameterType.text, inOut = false),
            @Parameter(name = "result", type = ParameterType.integer)
        })
    @ReturnValue(name = "result")
    public Integer parseInt(
        @ParameterValue(name = "text") String text)
    {
        try
        {
            return Integer.parseInt(text);
        }
        catch (NumberFormatException ex)
        {
            return null;
        }
    }
}

```

Metóda `parseInt` má rovnaké formálne rozhranie a ja tiež funkčne zhodná s `RPC PROCEDURE E.Service.Parse` uvedenou v kapitole 4.3.2. Pri jej mapovaní boli použité rovnaké anotácie s rovnakým významom ako na metódu `Service.parse`.

Rozdiel medzi `RPC Parse` napísanej v ESL a metódou `ExampleHandler.parseInt` je v tom, že metódu `parseInt` nie je možné zavolať asynchrónne. Pre takéto volanie by sa JAPI pokúsilo nájsť inú metódu s rovnakým formálnym rozhraním, ktoré by malo parameter `RPC.asynchronous = true`.

4.3.4. Použitie anotácie *CallerInformation*

```

import sk.ipesoft.d2000.d2japi.annotations.ParameterType;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.Parameter;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.ParameterValue;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.RPC;
import sk.ipesoft.d2000.d2japi.annotations.eventBinding.ReturnValue;

public class HelloHandler
{
    @RPC(name = "Hello")
    public void hello(
        @CallerInformation(type = CallerInformationType.processHobj) Integer processHobj,
        @CallerInformation(type = CallerInformationType.eventHobj) Integer eventHobj,
        @CallerInformation(type = CallerInformationType.internalJava) Boolean java)
    {
        ...
    }
}

```

Anotácia parametra metódy `@CallerInformation` indikuje, že za hodnotu parametra má byť dosadená informácia o volajúcom. Hodnota parametra `type` určuje, aká informácia bude dosadená:

- `processHobj` – HOBJ procesu (EVH, HIP, DCC), z ktorého bola RPC zavolaná
- `eventHobj` – HOBJ objektu typu `Event`, z ktorého bola RPC zavolaná.
- `internalJava` – nadobúda hodnotu `true`, ak bola RPC zavolaná z prostredia internej Javy.

4.4. Použitie anotovaných objektov

V kapitolách 4.2. a 4.3. a ich podkapitolách boli vytvorené definície mapovania štruktúrovaných hodnôt (`SD.Person` → `Person`) a definície mapovania volania RPC (`Service` → `E.Service`) a späť (trieda `ExampleHandler`). Ich použitie zjednodušuje mechanickú prácu vytvárania a konverzie *unival* hodnôt ako aj potrebu manažova a opakovane používa množstvo `HOBj` pre volanie RPC.

Pri používaní anotácií v jazyku Java je dôležité si uvedomiť, že označenie nejakého objektu anotáciou nemá žiadne priame funkčné dôsledky. Napríklad, ak vytvoríme inštanciu triedy `ExampleHandler` a následne zavoláme jeho metódu `parseInt`, jej vykonanie prebehne vždy rovnako, bez ohľadu na prítomnosť alebo neprítomnosť anotácie `@RPC`. Anotácie slúžia iba na to, aby sa kód, ktorý skúma iné asti kódu, vedel lepšie orientovať.

Nasledujúce príklady sa budú odkazovať na triedy vytvorené v príkladoch v kapitolách 4.2.1, 4.3.2. a 4.3.3. Okrem toho budú použité nasledovné objekty:

```
D2Connector connector = ... // aktívne spojenie
D2Session session = connector.createSession(... // aktívna session
```

Na konverziu štruktúrovaných hodnôt bude použitý objekt typu `UnivalConverter` prístupný cez `connector.getDefaultUnivalConverter()`.

4.4.1. Príklad volania z JAPI do ESL

V uvedenom príklade je ukážka volania RPC z prostredia JAPI. Príklad sa skladá z inicializovanej asti (kroky 1 a 2), ktorú je potrebné spraviť raz, pri štarte aplikácie, po pripojení sa ku kernelu. Ďalšie kroky (odrážky), sú príklady samotného volania, je možné použiť v ubovonom poradí a opakovať podľa potreby.

1. Vytvorenie tzv. Event Proxy Factory objektu. V tomto kroku je trieda `Service` analyzovaná. Nový objekt (factory) v sebe implementuje mapovanie RPC volaní podľa nájdených anotácií. Parametre volania majú nasledujúci význam:
 - `Service.class` referencia na triedu, ktorá definuje mapovanie
 - `session` získanie `HOBj` a parent `HOBj` objektu `E.Service`

```
EventProxyFactory<Service> factory =EventProxyFactory.createFactory(Service.class, session);
```

2. Vytvorenie tzv. Event Proxy objektu s využitím factory z predchádzajúceho kroku. V tomto kroku vznikne inštancia anonymnej triedy (odvodenej od `java.lang.reflect.Proxy`), ktorá implementuje rozhranie `Service`, aby bolo možné volať anotované metódy. Implementáciu tejto triedy generuje JAPI. Parameter volania má nasledovný význam:
 - `session` asociuje vytvorený objekt proxy s touto inštanciou `Session` – v jej mene budú RPC volané. Môže to byť iná inštancia, ako v kroku 1.

```
Service proxy = factory.createDefault(session);
```

Nasledujú príklady samotného volania:

- Jednoduché volanie RPC `Parse`:

```
Integer result;
result = proxy.parse("12"); // Integer.valueOf(12)
result = proxy.parse("ab"); // null
```

- Použitie kontajnera `InOut<>` vo volaní RPC `Square`:

```
Double value = 5.0;
InOut<Double> valueContainer = new InOut<>(value);
proxy.square_2(valueContainer);
Double result = valueContainer.getValue();
```

V aplikáciách, kde dochádza k viacnásobnému pripájaniu a odpájaniu počas životného cyklu aplikácie:

- Krok 1 stačí spraviť raz, po vytvorení 1. `session`. Vzniknutú `factory` môžete považovať za platnú, pokiaľ je aktívny `connector`. Pre nový `connector` je potrebné vytvoriť novú `factory`.
- Krok 2 je potrebné raz zopakovať pre každú novú `session`, z ktorej majú byť volané RPC.

4.4.2. Príklad volania z ESL do JAPI

V uvedenom príklade je ukážka volania RPC z prostredia ESL do prostredia JAPI a najmä spôsob spracovania a odpovedania na toto volanie. Príklad sa skladá z inicializovanej asti (kroky 1 a 2), po ktorej bude možné adresovať RPC volania do zvolenej `session`. Krok 3 je inicializácia v ESL. Krok 4 je samotné volanie, ktorý možno podľa potreby opakovať s ubovými parametrami.

1. Vytvorenie tzv. Event Wrapper Factory objektu. V tomto kroku je trieda `ExampleHandler` analyzovaná. Nový objekt `factory` v sebe implementuje spracovanie prichádzajúcich RPC volaní a ich mapovanie na volania metód triedy `ExampleHandler`. Význam použitých parametrov je nasledovný:

- `ExampleHandler.class` referenciu na triedu, ktorá definuje mapovanie
- `session` preklad mien

```
EventWrapperFactory<ExampleHandler> factory = EventWrapperFactory.createFactory(ExampleHandler.class, session);
```

2. Registrácia inštancie triedy `ExampleHandler` pre prijímanie RPC volaní do konkrétnej `session`. Po tomto kroku bude možné adresovať RPC `Parse` aj do použitej `session`.

```
ExampleHandler handler = new ExampleHandler();
factory.registerNewHandler(session, handler);
```

3. Uloženie HOBJ dynamického objektu `session`, aby bolo možné neskôr volať späť. V tele ubovenej RPC v ESL je možné zistiť adresu (HOBJ procesu a objektu) volajúceho nasledovným spôsobom:

```
INT _sessionHobj
RPC PROCEDURE Register
_sessionHobj := %GetRPCCallerProcess()
END Register
```

4. Volanie RPC z prostredia ESL. Namiesto mena objektu je použité `[]` pretože v *JAPI* neexistuje ekvivalent objektu typu `Event`. Namiesto mena procesu je použité `(_sessionHobj)` pretože `session` je v D2000 DODM dynamický objekt typu `proces` a na jeho meno sa nedá použiť ako identifikátor v zdrojovom kóde.

```
INT _r
CALL [] Parse("12", _r) ON (_sessionHobj)
```

¹ D2000 logická hodnota je štandardne mapovaná vymenovaným typom `sk.ipesoft.d2000.base.VBool`. Pre zjednodušenie použitia je umožnené mapovať typom `java.lang.Boolean`, pričom je hodnota `vOscillate` mapovaná ako `false`.

² Poet milisekúnd od UNIX epochy (1. 1. 1970 00:00:00 GMT). Pozor, systém D2000 používa inú epochu, jej hodnota je uložená v objekte `sk.ipesoft.d2000.utils.D2Time.d2TimeEpoch`.

³ Generický parameter `T` musí zodpovedať použitej *definícii štruktúry*.

⁴ Podľa konvencie JavaBeans tvoria metódy `getName` a `setName` spolu *property* `Name`.

⁵ Volajúci aká na dokončenie vykonávania RPC a môže tak získať návratové hodnoty parametrov.

⁶ Volajúci pokračuje vo vykonávaní hne po odoslaní správy a nemá žiadnu spätnú väzbu o vykonaní.

⁷ Rovnako v dokumente neuvádzame definíciu a mapovanie pre `SD.Metadata`, lebo nie je potrebné.

⁸ Môžu byť volané aj z prostredia internej Javy a tiež z prostredia *JAPI*.