ZÁRÓDOLGOZAT

Lázár Levente

2024

Magyar Agrár- és Élettudományi Egyetem

Károly Róbert Campus

Felsőoktatási szakképzés – programtervező informatikus szak

API hívások tesztjeinek átvitele Postman-ből ABAP környezetbe

Belső konzulens:	Dr. Pántya Róbert
	Egyetemi adjunktus
Belső konzulens intézete/tanszéke:	**MATE Műszaki Intézet**
Készítette:	**Lázár Levente**

Gyöngyös

2024

Tartalomjegyzék

1. Bevezetés

A záródolgozatom célja, hogy bemutasson egy fejlesztést a munkahelyemen, amely szakmai szempontból megfelel az egyetemi tanulmányaimhoz is. A projekt fő célja egy adott programban (*Postman*) íródott API unit tesztek (ezek a program legkisebb egységei, amik egyetlen konkrét feladatot látnak el) "átköltöztetése" egy másik programozási környezetbe, mivel a nevezett programot a cég a tavalyi évben kivezette az elérhető szoftverek közül. Így az én feladatom az lett, hogy ezeket a teszteket átvigyem a cég saját programozási nyelvébe, és hogy ugyanazt az eredményt produkálva sikeresen futtathatóak legyenek ezek a tesztek, amik a teszt-lefedettségben nagyon sokat segítenek, hiszen ezen esetek a vállalat ügyfeleinek számos rendszerében is hívásra kerülnek. A dolgozatban érinteni fogom az összes felmerülő kihívást és nehézséget, amik a projekt során felléptek, hiszen két különböző rendszer, illetve két különböző programozási nyelv esetéről van szó, eltérő megoldási lehetőségekkel, de a végeredménynek pontosan ugyanolyannak kell lennie, mint az eredeti tesztek esetében. Ezen felül szerepelni fog jelentős mennyiségű forráskód is, mindkét környezetből, hogy pontosan és teljes mértékben be tudjam mutatni az átköltözés menetét, illetve a programozási nyelvek különféle megoldásait egy-egy felmerülő problémára, hiszen amíg a Postman egy kifejezetten API tesztek írására és alkalmazására való szoftver, addig az ABAP egy üzleti programozási környezetet biztosító nyelv, amelyben ezen tesztek átköltöztetése nem is olyan egyszerű, mint azt elsőre gondolnánk.

2. Célkitűzések

A projekt egy ambiciózus útra indult, hogy áthidalja a szakadékot a Postman program és az ABAP környezet között. A Postman az API-fejlesztéshez és teszteléshez széles körben használt platform, az ABAP pedig központi szerepet játszik a cégünk rendszereiben.

A projekt lényege az eredetileg Postman programban végrehajtott API-hívástesztek replikálása volt az ABAP ökoszisztémában, hogy kihasználja az ABAP robusztus tesztelési keretrendszerét, és zökkenőmentesen integrálja ezeket a teszteket az ABAP-alkalmazások fejlesztési életciklusába.

Az API-hívások kulcsfontosságúak az ABAP-rendszerek számtalan külső alkalmazással és szolgáltatással való integrálásához. A projekt megoldotta a hívások Postman és ABAP rendszerben történő lebonyolítása és kezelése közötti különbségeket, beleértve a hitelesítést, a kérés formázását, a válaszelemzést és a hibakezelést.

A projekt egyik jelentős vívmánya az egyéni deszerializáló metódusok kifejlesztése volt az ABAP-ban a *JSON* válaszok kezelésére, amely a *RESTful API* kommunikáció általános formátuma. A *RESTful API* egy interfész, amelyet két számítógépes rendszer használ információk biztonságos cseréjére az interneten keresztül. Egy alkalmazás-programozási interfész (API, vagyis Application Programming Interface) meghatározza azokat a szabályokat, amelyeket követni kell más szoftverrendszerekkel való kommunikáció során. Egy webes API-t úgy is lehet tekinteni, mint egy kaput a kliensek és a weben elérhető erőforrások között. A Representational State Transfer (REST) egy szoftver-architektúra, amely meghatározott feltételeket szab az API működésére. Azokat az API-kat, amelyek követik a REST architekturális stílust, REST API-knak nevezzük. Többféle HTTP metódust tudunk alkalmazni ennek segítségével, de a dolgozatban a GET kérés kerül bemutatásra. A kliensek a GET metódust használják az erőforrások eléréséhez, amelyek a megadott URL-en találhatóak a szerveren. Lehet gyorsítótárazni (cache) a GET kéréseket és paramétereket küldhetünk a RESTful API kérésben, amelyek utasíthatják a szervert az adatok szűrésére a küldés előtt.

Forrás: https://aws.amazon.com/what-is/restful-api/

A *JSON* vagyis *JavaScript Object Notation* egy egyszerű adatcsere formátum, amely emberek számára könnyen olvasható és írható. A nevében szerepel a JavaScript, de ennek ellenére a JSON nyelvfüggetlen, sőt a legtöbb nyelvhez van is értelmezője.

A JSON név/érték párok gyűjteménye, melyek a különböző nyelveken objektumként, rekordként, struktúraként, kulcsos listaként vagy asszociatív tömbként valósulnak meg. Ezek univerzális adatszerkezetek és szinte az összes modern programozási nyelv támogatja őket. Egy ilyen objektum rendezetlen név/érték párokat tartalmazó halmaz. Egy objektum bal kapcsos zárójellel kezdődik (’{’) és jobb kapcsos zárójellel végződik (’}’), minden nevet kettőspont követ (’:’) és a név/érték párokat vessző (’,’) választja el.

Forrás: https://www.json.org/json-en.html

A deszerializálás folyamata a JSON formátumban érkező adatokat alakítja át programozási nyelv specifikus objektumokká vagy adatstruktúrákká, ami lehetővé teszi a programozók számára, hogy felhasználják ezeket az adatokat és dolgozhassanak velük tovább.

Forrás: https://stackoverflow.com/questions/3316762/what-is-deserialize-and-serialize-in-json

Ez az erőfeszítés az ABAP adatkezelési képességeinek mélyreható elmélyülését tette szükségessé, ami egy sor segédprogramot eredményezett, amelyek képesek JSON-karakterláncokat ABAP belső táblákká és struktúrákká alakítani. Ez az átalakítás kritikus fontosságú a tesztfeltételek érvényesítése és az API-válaszok várt értékekhez viszonyított érvényesítése szempontjából.

Ezen túlmenően a projekt innovatív módon kezelte a Postman dinamikus környezetének ABAP-ban való replikálásának kihívását, rugalmas és újrafelhasználható függvények létrehozásával az API-hívások indításához. Ezek a funkciók dinamikusan alkalmazkodhatnak a különféle API-végpontokhoz, kérésparaméterekhez és fejlécekhez, szorosan tükrözve a Postman által kínált rugalmasságot.

3. A GET típusú kérések bemutatása

A feladat a GET kérések kezelésével indult és pontosan ugyanúgy kellett megoldani a teszteseteket, mint ahogyan azok a Postman programban, JavaScript nyelven történtek. Kezelni kellett a GET kérés státusz kódját, illetve egy megadott szűrő szerinti JSON válasz vizsgálatát, hogy az elvárt értékek megfelelnek-e a visszakapott értékekkel.

A Postman egy API platform API-k építésére és használatára, amely egyszerűsíti az API életciklusának minden lépését és optimalizálja az együttműködést, így gyorsabban készíthetünk jobb API-kat. A Postman API platform egy szoftverrendszer integrált eszközökkel és folyamatokkal, amelyek lehetővé teszik a csapatok számára, hogy hatékonyan építsenek, kezeljenek, publikáljanak és használjanak API-kat. Az API platformok beépített együttműködési funkciókat tartalmaznak, mint például munkaterületek, sorok közötti megjegyzések és verziókövetés, amelyek támogatják a hatékony munkafolyamatokat és segítik a csapatokat a magas minőségű API-k szállításában. A legjobb API platformok irányítási és biztonsági ellenőrzéseket biztosítanak, amelyek könnyen végrehajthatók az API életciklusa során, és lehetővé teszik a készítők számára, hogy nyomon kövessék szabályzataik hatását, így szükség esetén módosíthatják azokat.

Forrás: https://www.postman.com/api-platform/

A GET az egyik leggyakoribb HTTP metódus, amelyet arra használunk, hogy lekérdezzünk adatokat egy megadott erőforrásból, rend szerint egy szerverről. A HTTP GET módszerrel történő kéréseknek csak adatot kell lekérniük, nem lehet adatot mellékelni a GET üzenet törzsében, és nem szabad más hatást gyakorolniuk az adatokra a szerveren. A Hypertext Transfer Protocol, vagyis http a World Wide Web (WWW) alap protokollja. Támogatja a kommunikációt egy böngésző vagy egy alkalmazás és a szerverek között. A HTTP protokollt arra használják, hogy információkat küldjenek olyan formátumban, amelyet mind a kliens, mind a szerver megért. Az adatok küldése a szerverre HTTP-n keresztül több HTTP kérési módszerrel történhet. A HTTP GET az egyik ilyen módszer. Ez a kérés tehát arra szolgál, hogy erőforrást kérjen a szervertől, csak adatokat fogadhat.

Forrás: https://reqbin.com/Article/HttpGet

Az alábbi ábrán látható az első kód, ami a Postman környezetben használható és JavaScript szintaxist követ:

1. ÁBRA: POSTMAN GET METÓDUS A PLANNING AREA LEKÉRDEZÉSÉRE

(Forrás: Saját szerkesztés Postman program alapján)

Mielőtt megvizsgáljuk a kódot, vegyük észre, hogy a felső sorban találhatunk egy legördülő menüpontot, ahol a már említett HTTP kéréseket tudjuk kiválasztani, ami jelen esetben a GET kérés. Mellette maga az URL található, amit küldünk a szervernek és a megadott szűrők alapján adja vissza a JSON választ. Ebben a specifikus példában a periódusokat vizsgáljuk (*PeriodSet*), a szűrőnk pedig a tervezési területek (*Plarea*, vagyis *Planning Area*) közül csak a '**CHINTNORM4**' nevűhöz tartozó periódusokat jeleníti meg, JSON formátumban (*$format=json*).

Az ábrán látható három függvényhívás, valamint egy konzolra való kiírás a visszaadott eredményekből. Sorról-sorra haladva vizsgáltam meg és értelmeztem a kód működését.

pm.test("Status code is 200", function () { pm.response.to.have.status(200); });

Ez a sor egy tesztet definiál a Postman programban, amely ellenőrzi, hogy a HTTP válaszunk státuszkódja 200-as e. A 200-as kód egy standard válasz a sikeres HTTP kérésre. A GET kérésben a válasz a kért erőforrásnak megfelelő entitásokat fogja tartalmazni, amit a kód feletti URL-ben adtunk meg.

A *pm.test* egy függvény, amely lehetővé teszi tesztesetek definiálását a Postman környezetben. Első paramétere a teszt neve ("*Status code is 200*"), a második pedig egy úgynevezett callback függvény, ami leírja a teszt logikáját.

A *pm.response.to.have.status(200)* kifejezés azt ellenőrzi, hogy a válasz státuszkódja 200-as e. Ha igen, a teszt sikeres, ha nem, akkor viszont sikertelen kérésről beszélünk.

pm.test("The number of periods are correct with no filter", function () { pm.expect(pm.response.json().d.results.length).to.eql(170); });

Ezekben a sorokban egy másik tesztet indítunk, amely azt ellenőrzi, hogy a válaszban lévő adatok száma megfelelő e (170 elemnek kell lennie a megadott szűrési feltételek mellett).

A *pm.test* ismételten egy tesztdefiníció, hasonlóan az előzőhöz.

A *pm.response.json()* kifejezés a JSON formátumú válaszadatot adja vissza. Ebben az esetben a JSON objektum ***d*** kulcsa alatti ***results*** tömb hosszát vizsgálja és ellenőrzi, hogy az 170 e. A to.eql(170) azt jelenti, hogy pontosan 170-nek kell lennie az értéknek.

console.log(pm.response.json().d.results[0]);

Ez a funkció kiírja a konzolra a válasz első elemét a results tömbből. A *console.log* a JavaScript-ben szokásos konzolra írás függvény, a legtöbb esetben hibakezelésre használjuk, jelen esetben csak bizonyosságot szeretnénk nyerni afelől, hogy a tömbünk tartalmaz elemeket, abból is az elsőt kiírjuk magunknak a konzolra. Magából a JSON válaszból való első elem kinyerését a *pm.response.json().d.results[0]* végzi, és a *console.log* írja azt ki.

postman.setEnvironmentVariable('url', pm.request.url);

Ez a függvény beállít egy környezeti változót (‘**url**’) a Postman programban, értéke pedig a jelenlegi kérés URL-je lesz, amit a kód feletti sorban adtunk meg. A *setEnvironmentVariable* egy régebbi függvény, amely egy környezeti változót állít be, vagy módosítja azt. Ez a függvény a Postman újabb verzióiban már elavult, helyette a *pm.environment.set* függvény használata javasolt. Az ‘*url*’ a változónk neve, a *pm.request.url* pedig az aktuális HTTP kérés URL-jét adja vissza, amit változóban tárolunk.

Forrás: R. M. Shah (2021): Decoding JavaScript. New Delhi: BPB Publications, India.

Forrás: A. Ranjan, A. Sinha, R. Battewad (2020): JavaScript for Modern Web Development. New Delhi: BPB Publications, India.

Az ábrán láthatjuk a GET kérésünk válaszának egy kivonatát.

2. ÁBRA: GET KÉRÉS VÁLASZ KIVONAT

(Forrás: Saját szerkesztés böngészőből való futtatás után)

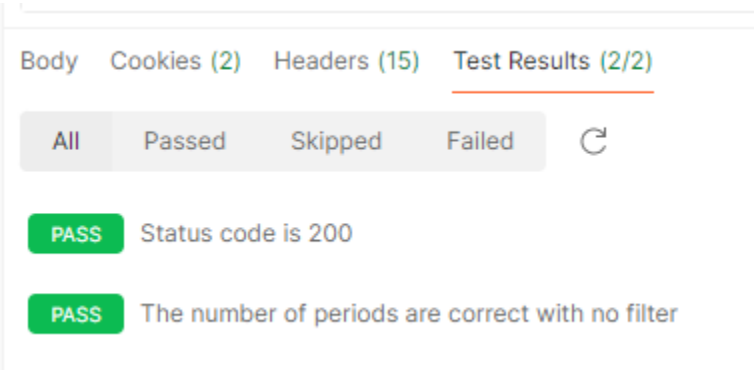

```
{
    "d": {
        "results": [
            {
                "__metadata": {
                    "id": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70553)",
                    "uri": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70553)",
                    "type": "IBP.API_CHANGEHISTORY_READ_SRV.Period"
                },
                "Periodid": 70553,
                "Plarea": "CHINTNORM4",
                "PeriodDescription": "JAN 2016",
                "PeriodLevelId": "PERIODID0",
                "TimeLevelDescription": "Month"
            },
            {
                "__metadata": {
                    "id": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70554)",
                    "uri": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70554)",
                    "type": "IBP.API_CHANGEHISTORY_READ_SRV.Period"
                },
                "Periodid": 70554,
                "Plarea": "CHINTNORM4",
                "PeriodDescription": "FEB 2016",
                "PeriodLevelId": "PERIODID0",
                "TimeLevelDescription": "Month"
            },
            {
                "__metadata": {
                    "id": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70555)",
                    "uri": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70555)",
                    "type": "IBP.API_CHANGEHISTORY_READ_SRV.Period"
                },
                "Periodid": 70555,
                "Plarea": "CHINTNORM4",
                "PeriodDescription": "MAR 2016",
                "PeriodLevelId": "PERIODID0",
                "TimeLevelDescription": "Month"
            },

            {
                "__metadata": {
                    "id": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(90055)",
                    "uri": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(90055)",
                    "type": "IBP.API_CHANGEHISTORY_READ_SRV.Period"
                },
                "Periodid": 90055,
                "Plarea": "CHINTNORM4",
                "PeriodDescription": "2024",
                "PeriodLevelId": "PERIODID1",
                "TimeLevelDescription": "Year"
            },
            {
                "__metadata": {
                    "id": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(90056)",
                    "uri": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(90056)",
                    "type": "IBP.API_CHANGEHISTORY_READ_SRV.Period"
                },
                "Periodid": 90056,
                "Plarea": "CHINTNORM4",
                "PeriodDescription": "2025",
                "PeriodLevelId": "PERIODID1",
                "TimeLevelDescription": "Year"
            }
        ]
    }
}
```


Az ábra nem teljes, mivel több, mint 2000 sort tartalmaz, így a válasz tömb első és utolsó sorai vannak feltűntetve. Jól látható, hogy a Postman szkript alapján a JSON objektumnak van egy ***d*** kulcsa, abban egy ***results*** nevű tömb, amiben a konkrét adatok szerepelnek. A szűrési feltétel az volt, hogy a Plarea legyen egyenlő *CHINTNORM4-gyel*, így ha szövegszerkesztőbe másoljuk a teljes válasz tartalmát és megszámoljuk, hogy ez a terület hányszor szerepel, akkor 170-et fogunk kapni.

Ez a manuális ellenőrzés, de természetesen a Postman programban is el tudjuk küldeni a kérést, és ha az sikeres, akkor maga a program is tájékoztat minket erről, ahogyan az alábbi ábrán láthatjuk:

3. ÁBRA: SIKERES POSTMAN TESZT

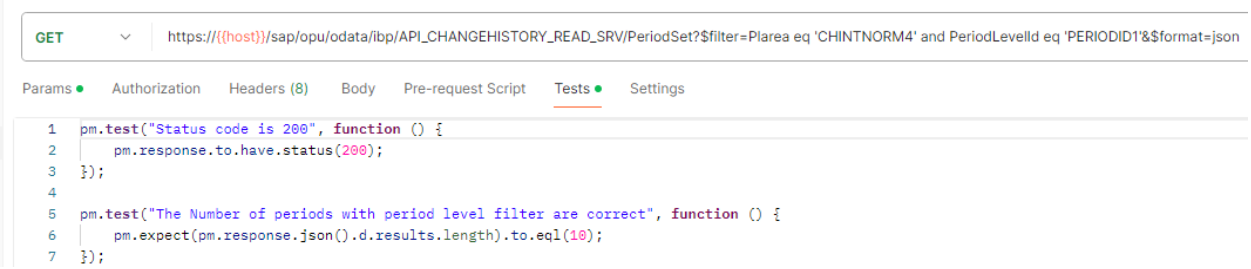
(Forrás: Saját szerkesztés Postman program alapján)



A következő példákban maradunk a Periódusoknál, továbbra is csak GET metódusokat tesztelünk, de itt már nem térünk ki a kód részletes ismertetésére, hiszen azok szinte megegyeznek az első példában szereplő teszttel, csak a szűrési feltétel, ezáltal pedig az URL változik. Ahol szignifikáns különbség szerepel az első példához képest, ott az ábrához külön kódismertetés is kerül.

4. ÁBRA: POSTMAN GET METÓDUS, PERIODLEVELID='PERIODID1' FELTÉTELLEL

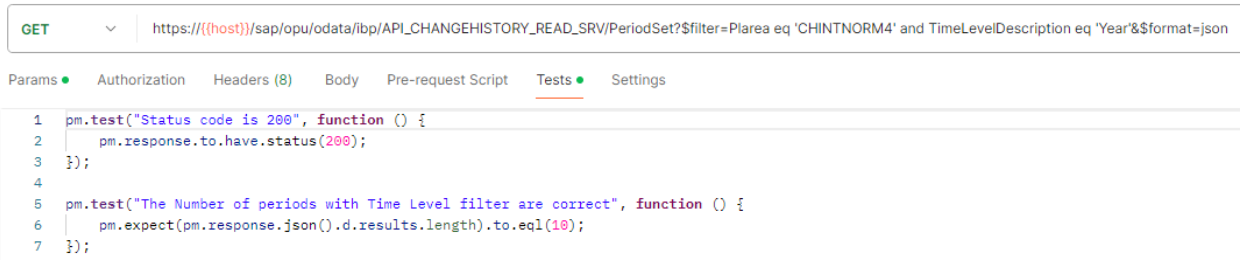
(Forrás: Saját szerkesztés Postman programban)



Az URL alapján láthatjuk, hogy most a feltételünk az, hogy a periódus szintjének azonosítója (*PeriodLevelId*) legyen egyenlő **'PERIODID1'-gyel**. Megvizsgáljuk, hogy a HTTP hívás sikeres volt e, erre a 200-as státuszkód ad sikeres választ, valamint megvizsgáljuk az eredmény tömbben lévő értékeket, amiknek jelen esetben 10-nek kell lennie.

5. ÁBRA: POSTMAN GET METÓDUS, TIMELEVELDESCRIPTION='YEAR' FELTÉTELLEL

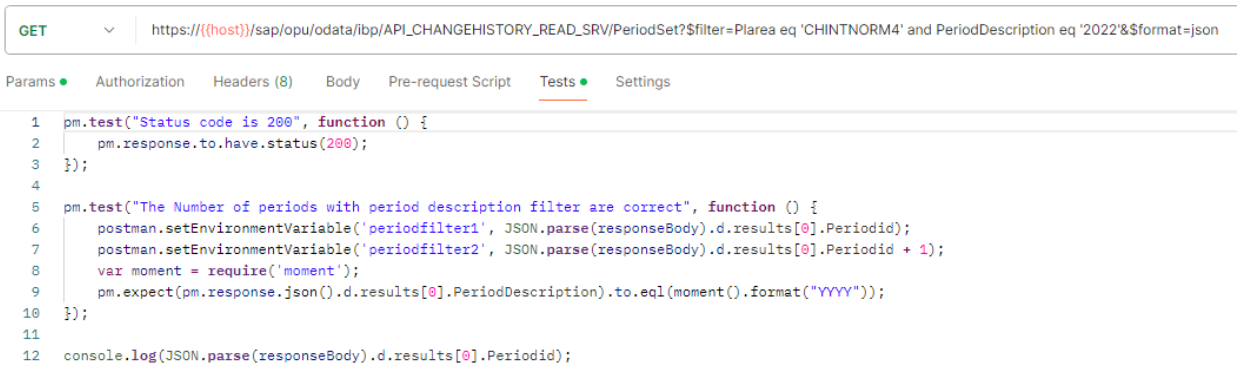
(Forrás: Saját szerkesztés Postman programban)



Új feltétel, mely szerint az időszint leírása (*TimeLevelDescription*) legyen egyenlő éves szinttel (*'Year'*). HTTP kérés sikerességének vizsgálata, az eredmény tömbben 10 elemnek kell szerepelnie, amikre igaz a megadott feltétel. A Planning Area (*Plarea*) még mindig a '**CHINTNORM4'** nevű.

6. ÁBRA: POSTMAN GET METÓDUS, PERIODDESCRIPTION='2022' FELTÉTELLEL

(Forrás: Saját szerkesztés Postman programban)



Itt a feltételünk az, hogy a periódus leírása egyezzen meg azzal, hogy '*2022'*, vagyis az ebben az évben létrejött termékeket szeretnénk lekérni. Ez a teszt azt ellenőrzi, hogy a válaszadatokban szereplő időszak leírása megfelel-e a jelenlegi évnek (a teszt 2022-ben íródott és azóta nem futott le sikeresen, többek között ezért kaptam a feladatot, hogy ezt az egészet átköltöztessem ABAP nyelvbe). A *postman.setEnvironmentVariable* segítségével két környezeti változót állítunk be, amelyek az API válaszában található első '**Periodid**' értékét, valamint annak eggyel növelt értékét tároljuk. A *JSON.parse(responseBody)* magát a '**responseBody**' vagyis a visszakapott válasz törzsének szöveges változatát alakítja egy JavaScript objektummá, amiből könnyen kiolvashatjuk a benne lévő adatokat.

A *require(moment)* esetében a '**moment'** egy JavaScript könyvtár, amelyet dátumok és idők kezelésére használhatunk. Itt a *moment().format("YYYY")* a jelenlegi évet adta vissza négy számjegyben. A *pm.expect* függvény ellenőrzi, hogy a válasz első elemének *PeriodDescription* mezője megegyezik-e a jelenlegi évvel.

4. A GET kérések átköltöztetése ABAP környezetbe

Az ABAP egy programozási nyelv, amelyet az SAP fejlesztett ki az SAP környezetben működő üzleti alkalmazások fejlesztésére. Támogatja az objektum-orientált programozási modellt osztályokon és interfészeken alapulva, illetve az eljárás-orientált modellt függvényeken és szubrutinokon alapulva. Az ABAP mint üzleti alkalmazásokra szánt nyelv egyik alapvető tulajdonsága, hogy az adatbázis táblákhoz való hozzáférés teljes mértékben integrált a nyelvbe. Az ABAP adatmodellezés lehetővé teszi üzleti alkalmazások adatmodelljeinek létrehozását. Ilyen például az ABAP Dictionary, amely egy állandó tároló az adattípusokhoz és azok függőségeihez. Ezek láthatóak és használhatóak minden más fejlesztési objektumban. Érdemes még megemlíteni, hogy az Open SQL lehetővé teszi az ABAP Dictionary-ban definiált adatbázis-objektumokhoz való hozzáférést automatikus klienskezeléssel, ami platformfüggetlen és integrálva van a nyelvbe.

Forrás: https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-us/abenabap_overview.htm

Most pedig térjünk át a projekt és a záródolgozat érdemi részére, magára a fejlesztésre, amelynek keretein belül az előző fejezetben említett GET metódusokat átvisszük az ABAP programozási nyelvbe. Mint említettem, a fenti Postman tesztek már meg voltak írva évekkel ezelőtt, de mivel magát a programot már nem lehet használni automatizált tesztekre, ezért ezek nem is futnak.

Ekkor jött a feladat, hogy ültessem át ezeket a teszteket ABAP-ba, ami szintén tud kezelni JSON objektumokat, tud kezelni ezekből képzett válaszokat, de alapvetően nem feltétlenül erre lett kitalálva, így maga a projekt igen nehéz volt és a legnehezebb részeket egy kicsit később részletezni is fogom.

Előtte azonban tekintsük meg az alábbi ábrát, melyen az első GET kérés ABAP-ban való megvalósítása látható:

7. ÁBRA: GET METÓDUS A PLANNING AREA LEKÉRDEZÉSÉRE ABAP-BAN

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
METHOD test_periods_plarea_filter.
  DATA(lv_response) = get_request_method( EXPORTING iv_uri = |PeriodSet?$filter=Plarea%20eq%20%27{ f_cut->gc_plarea }%27&$format=json| ).
  f_cut->check_http_response_code( lv_response ).
  DATA(lt_periods) = f_cut->parse_json_periods_to_table( iv_json_string = lv_response ).
  cl_abap_unit_assert=>assert_equals( act = lines( lt_periods )
                                      exp = 170
                                      msg = 'The number of periods are not correct with no filter' ).
ENDMETHOD.
```

Először elemezzük magát a kódot, majd részletesen belenézünk a meghívott függvények funkcióiba is, hogy egy teljes átfogó képet kapjunk a teszt működéséről. Ez az ABAP kód egy tesztmetódust tartalmaz. A metódus az API hívások tesztelésére szolgál egy adott forgatókönyv szerint. Ebben az esetben megegyezik a Postman programban bemutatott első példánkkal, ahol egy specifikus területet (*Planning Area*, vagyis *Plarea*) vizsgáló időszakok (*Periods*) szűrését teszteljük.

DATA(lv_response) = get_request_method(). Egy HTTP kérést indít, amelynek URI paramétere dinamikusan van összeállítva. A '*Plarea*' mező értéke megegyezik az '*f_cut->gc_plarea*' globális változó értékével, tehát ez a szűrőfeltételünk. A kérés eredménye JSON formátumban érkezik vissza. Az *lv_response* változóban tároljuk a HTTP kérés válaszát, amely tartalmazza a JSON formátumban kapott adatokat. Ezt a fajta deklarálási formát *in-line declaration-nek*, vagyis sor közbeni deklarálásnak nevezzük, ilyenkor a változónk típusa automatikusan van megadva és fordítási időben dől el, hogy pontosan mi is legyen az. Az '*f_cut*' egy osztályhoz tartozó adat az ABAP nyelvben (**CLASS-DATA**), ami egy referencia a globális osztályunkra. Az f '*prefix*' azt jelzi, hogy '*field*', vagyis mező, a *cut* pedig a *class under test* rövidítése, vagyis olyan osztály, ami épp tesztelés alatt van, tesztelésre szolgál. Jelenleg az *f_cut->gc_plarea* értéke *'CHINTNORM4'-re* van állítva a programban és ez nem is fog változni, de mindenképp fontos ezt is eltárolni egy globális változóba, nem pedig beleégetni az URL-be, hiszen ha egy másik Planning Area-t szeretnénk vizsgálni, akkor csak egy helyen, a deklaráció helyén kell átírnunk, nem pedig az adott esetben akár több 100 tesztfüggvényben. Érdemes még megemlíteni a prefix-ek használatát ABAP nyelvben.

Az *lv_response* nevű változóban például az *lv* azt jelöli, hogy lokális változó (*local variable*), a *gc_plarea* változóban a *gc* a globális konstans (*global constant*) jelölése.

f_cut->check_http_response_code(lv_response). Ez a metódus vizsgálja a HTTP válasz státuszkódját. Ellenőrzi, hogy a kérés sikeres volt e és hibás válasz esetén hibaüzenetet ad.

DATA(lt_periods) = f_cut->parse_json_periods_to_table(iv_json_string = lv_response). Az *lv_response* változóban tárolt JSON sztringet átalakítja egy ABAP táblába, amely az időszakokat (*periods*) tartalmazza. A JSON formátumból ABAP adatstruktúrába konvertáljuk az adatokat, hogy azokat további teszteléshez használjuk. Az *lt_periods* esetében az *lt* a lokális táblára utal (*local table*), az *iv_json_string* esetében pedig az *iv* az importálandó változóra utal (*importing variable*).

cl_abap_unit_assert=>assert_equals(act = lines(lt_periods) exp = 170 msg = 'The number of periods are not correct with no filter'). Az '*assert_equals*' metódust használjuk, amely összehasonlítja az '*lt_periods*' tábla sorainak számát az elvárt '170' értékkel. Ellenőrizzük, hogy a lekérdezés eredményeként visszaadott időszakok száma pontosan 170. Ha nem, akkor a teszt sikertelen és hibaüzenet jelenik meg ("*The number of periods are not correct with no filter*" vagyis a periódusok száma nem egyezik meg 170-nel). Az *act* az aktuális értékeket jelöli, ami jelen esetben egyenlő az *lt_periods* táblánk soraival. Az *exp* az *expected*, vagyis a várt eredményt jelöli. Az *msg* ami a *message*, vagyis az üzenet akkor jön, amikor az aktuális érték nem egyezik meg az elvárttal.

A kódblokk tehát egy adott terület szerinti szűrést tesztel, a válaszadatok helyességét ellenőrzi, majd a kapott adatok számát hasonlítja össze az elvárt értékekkel.

Forrás: W. Schwarzmann (N.a): Designing Testable ABAP Classes And Packages. SAP Press.

Most bontsuk apróbb részekre a kódot és vizsgáljuk meg az egyes meghívott metódusokat funkciójuk szerint. A *get_request_method* egy beépített, nem általam létrehozott függvény, így azt csak nagyvonalakban mutatom be, mivel nem az én érdemi munkám. Ez a metódus egy HTTP GET kérést indít egy speciális módon, használva az Odata szolgáltatásokat támogató batch kéréseket.

Beállítja az URI-t (Uniform Resource Identifier), ami megadja a kérés célpontját, hasonló módon, amint azt a Postman programban is láthattunk a kódsor feletti mezőben. Ha ezek mind megvannak, utána lezárjuk a teljes kérést a *close* függvények segítségével.

Ebben a függvényben történik magának a HTTP kérésnek a küldése is a szerver felé. Még mindig batch kérésről van szó, tehát háttérben futó alkalmazásról. Egy paramétert *'application/json'-ra* állítjuk, amely szerint JSON formátumú választ várunk. A státusz kódot tároló paramétert ebben a pillanatban 202-re állítjuk, ezt várjuk válaszként, ami azt jelzi, hogy a kérés elfogadva, de még nem hajtódott végre. Összefoglalva ez a metódus egy batch kérés keretében küld el egy GET kérést az adott URI-ra, majd ellenőrzi, hogy a válasz státusza a vártnak megfelel e. Ahogy láthattuk korábban, ez a fajta kérés hasznos a nagy mennyiségű adat lekérdezésekor, mint például ebben az esetben is, hogy a kapott válasz 170 féle objektumot tartalmaz, több mint 2000 sorban.

A következő függvényünk az ábrán is látható HTTP státuszkód ellenőrző:

8. ÁBRA: A CHECK_HTTP_RESPONSE_CODE METÓDUS

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
127
128  METHOD check_http_response_code.
129    FIND REGEX 'HTTP/1.1 200 OK' IN iv_response.
130
131    IF sy-subrc NE 0.
132      cl_abap_unit_assert=>fail( msg = 'GET REQUESTED FAILED.' ).
133    ENDIF.
134  ENDMETHOD.
```

Ez a metódus arra szolgál, hogy ellenőrizze egy HTTP válasz státuszkódját. A cél, hogy megerősítse, a kérés sikeres volt e. A funkciója a **FIND REGEX** kulcsszavakkal kezdődik, ami egy reguláris kifejezést használ (*Regular Expressions*, vagyis *REGEX*) az *iv_response* változóban, amely a HTTP válasz szövegét tartalmazza.

A regex kifejezés a "**HTTP/1.1 200 OK**" mintát keresi a válasz fejlécében, ami jelzi, hogy a kérés sikeresen teljesült. Ha lefuttatjuk az eredeti teszt függvényünket, amit a 8-as ábrán láthatunk és egy *breakpoint-ot* teszünk a HTTP státuszkód ellenőrző sorba, akkor azt láthatjuk, hogy futás közben a válaszunk a következő fejléccel van ellátva:

Content-Type: application/http
Content-Length: 68882
content-transfer-encoding: binary

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 68689
dataserviceversion: 2.0
metadata-last-modified: Wed, 17 Apr 2024 17:14:59 GMT
cache-control: no-store, no-cache

Itt tehát látható, hogy az a kifejezés, hogy "**HTTP/1.1 200 OK**" szerepel a fejlécben, tehát a kérés és annak ellenőrzése sikeres volt. A kérés eredménye a '*sy-subrc*' rendszerváltozóban tárolódik. Ha a kérés sikeres, akkor a '*sy-subrc*' értéke 0 lesz. Minden más esetben sikertelen kérésről beszélünk. Ezt a következő sorokban vizsgáljuk. Amennyiben a '*sy-subrc*' értéke nem egyenlő a nullával (**IF sy-subrc NE 0**, ahol az NE a *not equals*, vagyis nem egyenlő jelölése), tehát ha a '*sy-subrc*' nem nulla, azaz a kérés sikertelen volt, akkor belép az IF blokkba. Itt meghívjuk a *cl_abap_unit_assert=>fail* metódust, ami egy teszt hibát jelent. A '*fail*' megállítja a teszt futását és hibaüzenetet ("**GET REQUESTED FAILED**") ad ki, ami jelzi, hogy a GET kérés nem volt sikeres.

4.1. A JSON válasz feldolgozása

A következőkben bemutatásra kerül a 8-as ábra negyedik sora: **DATA(lt_periods) = f_cut->parse_json_periods_to_table(iv_json_string = lv_response).**

A *parse_json_periods_to_table* metódus feladata egy JSON formátumú szöveg (*string*) feldolgozása, majd annak elemzése és átalakítása egy adattáblába, amely időszakokat (*periods*) reprezentál. Ez a folyamat tipikusan két részre bontható: JSON objektumok kinyerése, majd az egyes objektumok elemzése.

Az alábbi ábrán láthatjuk ennek a függvénynek a kódját:

9. ÁBRA: A PARSE_JSON_PERIODS_TO_TABLE FÜGGVÉNY

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
METHOD parse_json_periods_to_table.
  DATA lt_json_objects TYPE TABLE OF string.

  " Extract JSON objects
  lt_json_objects = extract_json_objects( iv_json_string = iv_json_string ).

  " Parse JSON objects
  LOOP AT lt_json_objects INTO DATA(lv_json_object).
    DATA(ls_period) = parse_json_object_period( lv_json_object ).
    APPEND ls_period TO rt_periods.
  ENDLOOP.
ENDMETHOD.
```

DATA lt_json_objects TYPE TABLE OF string. Egy lokális táblát hozunk létre (*lt: local table*), amely karakterláncokat tartalmaz, és ez fogja tartalmazni az egyes JSON objektumokat szöveges formában. Funkciója az ABAP adatstruktúra előkészítése a JSON objektumok tárolására.

lt_json_objects = extract_json_objects(iv_json_string = iv_json_string). Itt meghívunk egy másik függvényt, az *extract_json_objects* nevűt, ami szintén bemutatásra fog kerülni hamarosan. A függvény megkapja az *iv_json_string* bemeneti paramétert, ami egy JSON formátumú szöveget tartalmaz, majd feldolgozza és kinyeri belőle az egyes JSON objektumokat. Ezeket a sztringeket eltároljuk az *lt_json_objects* táblában, amit az előző sorban hoztunk létre. Ennek funkciója, hogy a JSON sztringből kinyert objektumokat külön-külön lehessen feldolgozni.

Az utolsó sorokban egy ciklus található, ami végigmegy az *lt_json_objects* táblában található összes JSON objektumon. Mindegyik iterációban egy *lv_json_object* változóban tároljuk az aktuálisan kinyert JSON objektumot. Újabb segédfüggvényt hívunk, ***parse_json_object_period***. Az *lv_json_object* változóban lévő JSON objektumot itt alakítjuk át ABAP struktúrába (*ls_period*, lokális struktúra, vagyis *local structure*). Majd az *ls_period* struktúrát hozzáadjuk az *rt_periods* táblához, ami egy olyan tábla, amely az összes feldolgozott időszak adatát tartalmazza. Itt tehát a JSON objektumok egyenkénti feldolgozása és ABAP struktúrába való átalakítása történik, majd ezeket gyűjtjük egy táblába, például a tábla sorainak megszámlálására.

Forrás: K. Bandari (2022): Complete ABAP. SAP Press.

4.2. ABAP típusok és struktúrák deklarálása

Az alábbi ábrán láthatjuk az előzőekben említett táblát, illetve ABAP struktúrát, amit előre kellett definiálni a periódusok lekérdezésére és a JSON válasz objektumainak tárolására:

10. ÁBRA: TY_METADATA ÉS TY_PERIOD STRUKTÚRA LÉTREHOZÁSA

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
TYPES:
  BEGIN OF ty_metadata,
    id   TYPE string,
    uri  TYPE string,
    type TYPE string,
  END OF ty_metadata,

  BEGIN OF ty_period,
    metadata             TYPE ty_metadata,
    Periodid             TYPE /ibp/period_id,
    Plarea               TYPE /ibp/dm_plarea,
    PeriodDescription    TYPE /ibp/period_descr,
    PeriodLevelId        TYPE /ibp/period_attr,
    TimeLevelDescription TYPE /ibp/dm_tplevel_descr,
  END OF ty_period,
  tt_periods TYPE STANDARD TABLE OF ty_period WITH EMPTY KEY,
```

Ha megfigyeljük a korábban bemutatott választ (**2-es ábra**), amit visszakaptunk egy böngészőben indított GET kérés után, a struktúra ugyanaz, mint ami a fenti ábrán definiálva van. Egy külön struktúrában tárolom a *metadata* adatokat (*id, uri, type*) és a fő struktúránk pedig a periódusok típusa, a *ty_period*, ami áll egy már említett *metadata* tagból, aminek *ty_metadata* a típusa, valamint *Periodid, Plarea, PeriodDescription, PeriodLevelId és TimeLevelDescription* tagokból, melyek egyedi típusú változók, de mind szöveges alapúak.

A típus deklarálása végén, az utolsó sorban látható, hogy létrehozok egy típusos táblát *tt_periods* néven (*tt: type table*), ami *ty_period* típusok táblája, lényegében tömbje, viszont kulccsal van ellátva, aminek a segítségével egyedivé tudjuk tenni a tábla egyes sorait.

Az alábbi ábrán újra megnézhetünk egy apró kivonatot a GET kérésből kapott válaszból:

11. ÁBRA: GET KÉRÉS KIVONAT

(Forrás: Saját szerkesztés böngészőből)


```
[
  {
      "__metadata": {
          "id": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70553)",
          "uri": "https://pd6-001.wdf.sap.corp:443/sap/opu/odata/ibp/API_CHANGEHISTORY_READ_SRV/PeriodSet(70553)",
          "type": "IBP.API_CHANGEHISTORY_READ_SRV.Period"
      },
      "Periodid": 70553,
      "Plarea": "CHINTNORM4",
      "PeriodDescription": "JAN 2016",
      "PeriodLevelId": "PERIODID0",
      "TimeLevelDescription": "Month"
  },
```


4.3. A projekt legnehezebb feladata: objektumok kinyerése a JSON válaszból

Az alábbi ábrán a projekt legfontosabb függvényét láthatjuk, amelynek ***extract_json_objects*** a neve, és ami képes bonyolult JSON válaszokból releváns adatokat kinyerni és előkészíteni azokat további feldolgozásra:

12. ÁBRA: AZ EXTRACT_JSON_OBJECTS METÓDUS

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
METHOD extract_json_objects.
  DATA: lv_json_array        TYPE string,
        lt_json_objects      TYPE TABLE OF string,
        lv_start_pos         TYPE i,
        lv_end_pos           TYPE i,
        lv_reversed_response TYPE string,
        lv_json_extracted    TYPE string,
        lv_length            TYPE i.

  " Extract JSON content from the response
  FIND FIRST OCCURRENCE OF gc_opening_bracket IN iv_json_string MATCH OFFSET lv_start_pos.
  lv_reversed_response = reverse( iv_json_string ).
  FIND FIRST OCCURRENCE OF gc_closing_bracket IN lv_reversed_response MATCH OFFSET lv_end_pos.
  lv_end_pos = strlen( iv_json_string ) - lv_end_pos.

  IF lv_start_pos > 0 AND lv_end_pos > lv_start_pos.
    lv_length = lv_end_pos - lv_start_pos + 1.
    lv_json_extracted = iv_json_string+lv_start_pos(lv_length).
  ELSE.
    RETURN. " JSON not found
  ENDIF.

  " Find start and end of the JSON array within the extracted JSON
  FIND FIRST OCCURRENCE OF '[' IN lv_json_extracted MATCH OFFSET lv_start_pos.
  FIND FIRST OCCURRENCE OF ']' IN lv_json_extracted MATCH OFFSET lv_end_pos.
  IF lv_start_pos > 0 AND lv_end_pos > lv_start_pos.
    lv_length = lv_end_pos - lv_start_pos + 1.
    lv_json_array = lv_json_extracted+lv_start_pos(lv_length).
  ELSE.
    RETURN. " JSON array not found
  ENDIF.

  " Split into individual JSON objects
  SPLIT lv_json_array AT '},{' INTO TABLE lt_json_objects.

  rt_json_objects = lt_json_objects.
ENDMETHOD.
```

Ez az eljárás alapvetően egy JSON tömb elemeit különálló JSON objektumokként gyűjti össze és tárolja el egy táblában. A függvény a szükséges változók deklarálásával kezdődik, létrehozunk többek között sztringeket az adatok tárolására és indexeket a szövegben való kereséshez.

A következőkben a JSON tartalom válaszból való kinyerése történik:

FIND FIRST OCCURRENCE OF gc_opening_bracket IN iv_json_string MATCH OFFSET lv_start_pos.

lv_reversed_response = reverse(iv_json_string).

FIND FIRST OCCURRENCE OF gc_closing_bracket IN lv_reversed_response MATCH OFFSET lv_end_pos.

lv_end_pos = strlen(iv_json_string) - lv_end_pos.

Első lépésként meghatározzuk a JSON objektum kezdő *'{'* és záró *'}'* kapcsos zárójelek pozícióját a szövegben. A *gc_opening_bracket* és *gc_closing_bracket* változók globális konstansok és értékük a nyitó és záró kapcsos zárójelek:

13. ÁBRA: NYITÓ ÉS ZÁRÓ KAPCSOS ZÁRÓJELEK DEFINIÁLÁSA GLOBÁLIS KONSTANSKÉNT

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
gc_opening_bracket     TYPE c VALUE '{',
gc_closing_bracket     TYPE c VALUE '}',
```

A záró kapcsos zárójel megtalálásához megfordítjuk a szöveget a *reverse()* függvény segítségével, hogy a zárójel első előfordulását megtalálva, a sztring végétől számoljuk a pozíciót. Ehhez a **FIND FIRST OCCURRENCE OF** kulcsszavakat használjuk, ami megkeresi egy karakter első előfordulását a szövegben. Érdekesség, hogy a megfordítást ki lehetett volna hagyni, ha lenne az ABAP nyelvben **FIND LAST OCCURRENCE OF** funkció, vagyis ami egy karakter utolsó előfordulását találná meg. Mivel ilyen funkció nincs, ezért először megkeressük az első nyitó zárójelet, innen kezdődik a szöveg feldolgozása, majd a megfordított szöveg egy záró zárójellel fog kezdődni, ahol az eredeti szövegünk végződik, addig tart a feldolgozás.

Az ezt követő **IF-ELSE** elágazásban ellenőrizzük, hogy érvényes pozíciókat találtunk-e és ha igen, kivágjuk a teljes JSON szakaszt a válaszból. Ellenkező esetben a **RETURN** kulcsszóval egyszerűen kilépünk az elágazásból, ez azt jelenti, hogy nem található megfelelően visszaadott JSON válasz.

```
FIND FIRST OCCURRENCE OF '[' IN lv_json_extracted MATCH OFFSET lv_start_pos.
FIND FIRST OCCURRENCE OF ']' IN lv_json_extracted MATCH OFFSET lv_end_pos.
IF lv_start_pos > 0 AND lv_end_pos > lv_start_pos.
  lv_length = lv_end_pos - lv_start_pos + 1.
  lv_json_array = lv_json_extracted+lv_start_pos(lv_length).
ELSE.
  RETURN. " JSON array not found
ENDIF.
```

Ebben a szakaszban azonosítjuk a tömböt jelző *'['* és *']'* zárójeleket a kinyert JSON szakaszon belül a már ismert módon ugyanúgy, mint a kapcsos zárójelek esetében és ki is vágjuk ezt a részt. Ezzel magát a JSON tömböt tudjuk kinyerni, ami a különálló JSON objektumokat tartalmazza.

SPLIT lv_json_array AT '},{' INTO TABLE lt_json_objects. Itt az egyes JSON objektumokra való bontás történik. A JSON tömböt a vessző (‘,’) karakterek mentén szétbontjuk, pontosabban a ‘},{‘ részeken, mivel az objektumokat így választjuk el egymástól, ezáltal különálló JSON objektumokat kapunk. Az egyes JSON objektumokat külön sztringenként tároljuk az *lt_json_objects* nevű táblában.

Majd az utolsó művelet az eredmény visszaadása az **rt_json_objects = lt_json_objects** utasítással, tehát az összegyűjtött JSON objektumokat tartalmazó tábla értékét átadjuk a metódus kimeneti paraméterének, a feldolgozott adatokat továbbítjuk, hogy más metódusok felhasználhassák őket.

Forrás: J. Wood, J. Rupert (2016): Object-Oriented Programming with ABAP Objects. SAP Press.

4.4. Az aktuális értékek kinyerése a JSON válaszból REGEX használatával

Ha visszatekintünk a **9-es ábrára**, található azon egy sor, ami még nem került ez idáig bemutatásra: **DATA(ls_period) = parse_json_object_period(lv_json_object).** Az alábbi ábrán megtekinthetjük a *parse_json_object_period* metódust, ami egy másik segítő függvényt is használ, *extract_value_by_regex* néven. A következőben ezt a két, egymással szoros kapcsolatban álló függvényt fogjuk elemezni.

14. ÁBRA: A PARSE_JSON_OBJECT_PERIOD METÓDUS

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
METHOD parse_json_object_period.
  DATA ls_period TYPE ty_period.

  ls_period-periodid = extract_value_by_regex(
    iv_json_string = iv_json_object
    iv_pattern = '"Periodid":\s*(\d+)' ).

  ls_period-periodlevelid = extract_value_by_regex(
    iv_json_string = iv_json_object
    iv_pattern = '"PeriodLevelId":\s*"([^"]*)"' ).

  ls_period-timeleveldescription = extract_value_by_regex(
    iv_json_string = iv_json_object
    iv_pattern = '"TimeLevelDescription":\s*"([^"]*)"' ).

  ls_period-perioddescription = extract_value_by_regex(
    iv_json_string = iv_json_object
    iv_pattern = '"PeriodDescription":\s*"([^"]*)"' ).

  rs_period = ls_period.
ENDMETHOD.
```

Ennek a metódusnak a feladata egy JSON objektum elemzése és adatok kinyerése egy specifikus típusú struktúrába (*ty_period*). A kód használ reguláris kifejezéseket (*regex*) a JSON formátumban szereplő adatok kinyerésére. Első lépésként definiálunk egy lokális struktúrát *ls_period néven* (*ls: local structure*), amely a *ty_period* típust használja, amit korábban láthattunk, hogy többek között olyan, számunkra aktuálisan releváns mezőket tartalmaz, mint '*periodid*', '*periodlevelid*', '*timeleveldescription'* és '*perioddescription'*.

A következő sorok az adatok kinyerésére szolgálnak regex segítségével:

ls_period-periodid = extract_value_by_regex(iv_json_string = iv_json_object iv_pattern = '"Periodid":\s*(\d+)'). Minden további sor hasonló mintát követ: adatok kinyerése az *iv_json_object* változóból a különféle mezők számára.

Az *extract_value_by_regex* funkciót meghívjuk, amely két paramétert kap: *iv_json_string*, ami a JSON objektumot tartalmazó string, és *iv_pattern*, ami a keresett adatot meghatározó reguláris kifejezés. Ebben az esetben a kifejezés a **'"Periodid":\s*(\d+)'** mintát keresi, ahol **'\s*'** bármennyi szóközt jelent és a **'(\d+)'** egy vagy több számjegyet jelentő csoportot.

Ezzel tehát kinyerjük a *periodid* mező értékét és el is tároljuk az *ls_period* struktúra *periodid* attribútumában. A további mezők (*periodlevelid, timeleveldescription, perioddescription*) hasonló módon van feldolgozva, ahol a minták a következők:

- **PeriodLevelId**: "PeriodLevelId":\s*"([^"]*)"
- **TimeLevelDescription**: "TimeLevelDescription":\s*"([^"]*)"
- **PeriodDescription**: "PeriodDescription":\s*"([^"]*)"

Ezek a regex kifejezések az adott mező értékét keresik meg, ahol a **'[*"]*'** kifejezés azt jelenti, hogy bármennyi, idézőjel kivételével bármilyen karakter előfordulhat. Az utolsó sorban **rs_period = ls_period.**, az előző lépésekben betöltött *ls_period* struktúrát hozzárendeli a metódus visszatérési paraméteréhez (*rs_period*). Ezzel a feldolgozott adatokat visszaadjuk, hogy más metódusok felhasználhassák.

A következő ábrán tekinthetjük meg az *extract_value_by_regex* függvényt:

15. ÁBRA: AZ EXTRACT_VALUE_BY_REGEX METÓDUS

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
METHOD extract_value_by_regex.
  DATA: lv_match_offset   TYPE i,
        lv_match_length   TYPE i,
        lv_matched_string TYPE string,
        lv_result_string  TYPE string.

  FIND FIRST OCCURRENCE OF REGEX iv_pattern IN iv_json_string MATCH OFFSET lv_match_offset MATCH LENGTH lv_match_length
    SUBMATCHES lv_result_string.

  IF sy-subrc = 0 AND lv_result_string IS NOT INITIAL.
    TRY.
        rv_value = lv_result_string.
      CATCH cx_sy_conversion_no_number.
        CLEAR rv_value.
    ENDTRY.
  ENDIF.
ENDMETHOD.
```

A fenti metódus feladata egy megadott reguláris kifejezés (*regex*) alapján értékek kinyerése egy JSON formátumú sztringből. Ez a metódus igen fontos az adatfeldolgozás során, különösen akkor, amikor adatokat kell kinyerni szöveges formátumokból. Magát a konkrét mintát az előző függvényben definiáltuk, itt csak az adatkinyerés történik. Az első sorokban a szükséges változókat definiáljuk, amelyeket a reguláris kifejezés keresésének során fogunk használni. Ezek tartalmazzák a találat kezdő pozícióját (*lv_match_offset*), a találat hosszát (*lv_match_length*), a megtalált szöveget (*lv_matched_string*), valamint a kinyert értéket (*lv_result_string*).

FIND FIRST OCCURRENCE OF REGEX iv_pattern IN iv_json_string MATCH OFFSET lv_match_offset MATCH LENGTH lv_match_length SUBMATCHES lv_result_string. Itt történik a regex keresés, a **FIND** utasítás az *iv_pattern* reguláris kifejezést keresi meg az *iv_json_string* változóban. A mintát (*pattern*) a már említett előző függvényben definiáltuk, hiszen az egy specifikus függvény, ez pedig egy segéd metódus, ami magát az adatok kinyerését végzi.

A **MATCH OFFSET** és a **MATCH LENGTH** opciókkal meghatározzuk a találat pozícióját és hosszát, míg a **SUBMATCHES** opcióval a zárójelekkel megjelölt alcsoportok értékét nyerhetjük ki közvetlenül.

Az utolsó sorokban egy **IF** egyirányú elágazás és azon belül **TRY-CATCH** kivételkezelés történik. Először is ellenőrizzük, hogy a keresés sikeres volt e (*sy-subrc = 0*) és hogy az *lv_result_string* nem üres (**IS NOT INITIAL**, tehát nem kezdeti üres érték). Ha ezek teljesülnek, akkor megpróbáljuk (**TRY**) hozzárendelni a kinyert értéket az *rv_value* visszatérési változóhoz (*rv: returning value*). A **TRY-CATCH** blokk arra szolgál, hogy kezelje az esetleges típuskonverziós hibákat, például ha a kinyert adat nem konvertálható a célváltozó típusára (*cx_sy_conversion_no_number* beépített hibakezelő függvény). Ebben az esetben az *rv_value* törlésre kerül (**CLEAR rv_value**), jelezve, hogy a konverzió nem sikerült.

Az *extract_value_by_regex* metódus így egy nagyon hasznos eszköz a strukturálatlan szöveges adatokból való strukturált információk kinyerésére, különösen JSON formátumú adatok esetén. A reguláris kifejezések rugalmassága és az ABAP nyelv beépített kezelési lehetősége lehetővé teszik, hogy pontosan és hatékonyan dolgozhassuk fel az adatokat.

Forrás: K. Haeuptle, F. Hoffmann, R. Jordao, M. Martin, A. Ravinarayan, K. Westerholz (2021): Clean ABAP – A Style Guide for Developers. SAP Press.

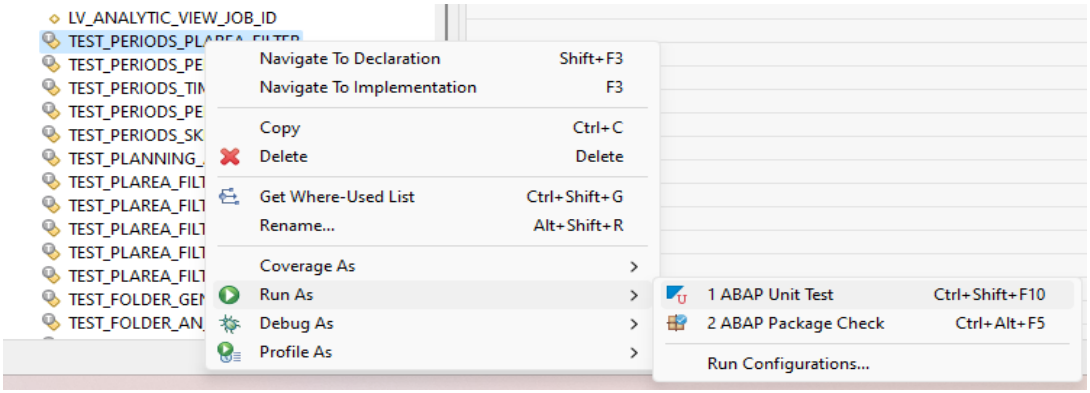
4.5. Sikeres futtatás után az eredmény ugyanaz, mint a Postman programban

Az utolsó ábrákon láthatjuk, hogy a teszt metódus futtatása után az eredményünk pontosan megegyezik azzal, amit a Postman programban, illetve a böngészőben indítottunk. A GET kérés ugyanolyan struktúrában adta vissza az adatokat ABAP környezetben, így a teszt sikeresnek tekinthető és innentől lehet alkalmazni éles rendszerekben, automatizált módon, hogy naprakész információink legyenek a termékeink periódusainak sikeres létrejöttéről és tárolásáról.

Első lépésként indítsuk el a teszt függvényt ABAP unit tesztként:

16. ÁBRA: TESZT INDÍTÁSA

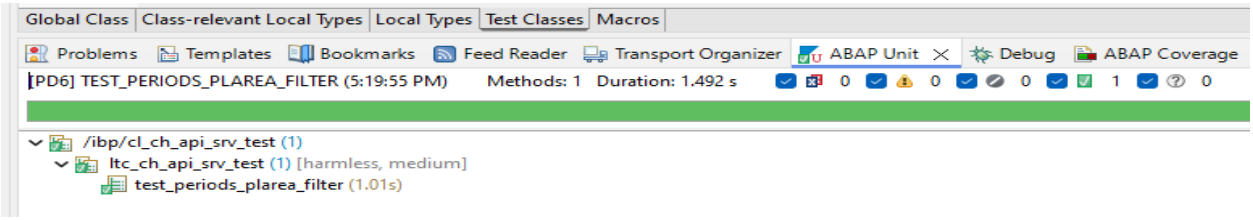
(Forrás: Saját szerkesztés ABAP Development Tools programban)



A következő ábrán a teszt sikeres futását láthatjuk az ABAP fejlesztői környezetében. Kiegészítésként hozzá kell tenni, hogy a fejlesztés pillanatában egy teszt rendszerben dolgoztam és a cél, hogy a teszt hiba nélkül lefusson, hogy utána ki lehessen ajánlani ügyfelek éles rendszereibe.

17. ÁBRA: A TEST_PERIODS_PLAREA_FILTER FÜGGVÉNY FUTTATÁSA

(Forrás: Saját szerkesztés ABAP Development Tools programban)



Végezetül az alábbi ábrán látható, hogy az *lt_periods* táblába a JSON deszerializáló függvényem bepakolja az egyes adatokat a JSON válaszból, hogy utána tudjunk ezen adatokkal műveleteket végezni, mint például megszámolni és ellenőrizni, hogy valóban 170 külön egységet tartalmaz-e:

18. ÁBRA: AZ LT_PERIODS TÁBLA TARTALMÁNAK KIVONATA

(Forrás: Saját szerkesztés ABAP Development Tools programban)

```
223  METHOD test_periods_plarea_filter.
224    DATA(lv_response) = get_request_method( EXPORTING iv_uri = |Pe
225    f_cut->check_http_response_code( lv_response ).
226    DATA(lt_periods) = f_cut->parse_json_periods_to_table( iv_json
227
228    cl_abap_unit_assert=>assert_equals( act = lines( lt_periods )
229                                        exp = 170
230                                        msg = 'The number of perio
231  ENDMETHOD.
```

Global Class | Class-relevant Local Types | Local Types | Test Classes | Macros

Console | Problems | ABAP Unit | ABAP Internal Table (Debugger) | Search

Table Data | Columns | LT_PERIODS | [170x6(812)]Standard Table

Filter pattern

Row	METADATA	PERIODID	PLAREA	PERIODDESCRIPTION	PERIODLEVELID	TIMELEVELDESCRIPTION
1	Structure: deep	70553		JAN 2016	PERIODID0	Month
2	Structure: deep	70554		FEB 2016	PERIODID0	Month
3	Structure: deep	70555		MAR 2016	PERIODID0	Month
4	Structure: deep	70556		APR 2016	PERIODID0	Month
5	Structure: deep	70557		MAY 2016	PERIODID0	Month
6	Structure: deep	70558		JUN 2016	PERIODID0	Month
7	Structure: deep	70559		JUL 2016	PERIODID0	Month
8	Structure: deep	70560		AUG 2016	PERIODID0	Month
9	Structure: deep	70561		SEP 2016	PERIODID0	Month
10	Structure: deep	70562		OCT 2016	PERIODID0	Month
11	Structure: deep	70563		NOV 2016	PERIODID0	Month
12	Structure: deep	70564		DEC 2016	PERIODID0	Month
13	Structure: deep	70565		JAN 2017	PERIODID0	Month
14	Structure: deep	70566		FEB 2017	PERIODID0	Month
15	Structure: deep	70567		MAR 2017	PERIODID0	Month

5. Következtetések

Ez a projekt további automatizálási lehetőségeket nyit a cég tesztelési gyakorlatain belül. Az ABAP-on belüli API-tesztelés alapjaival a hasonló módszertanok kiterjeszthetők más típusú külső integrációkra is, tovább csökkentve a kézi tesztelési terheket és növelve az automatizált tesztek hatókörét. Ez a kibővített tesztlefedettség létfontosságú annak biztosításához, hogy az ABAP-rendszerek, amelyek gyakran a vállalati IT-környezet gerincét képezik, zökkenőmentesen kommunikálhassanak a gyorsan változó digitális ökoszisztémával.

Oktatási szempontból a projekt esettanulmányként szolgál a több tudományágat átfogó tanulásban, bemutatva, hogy a modern webfejlesztés világából származó eszközök és technikák hogyan segíthetik elő és javíthatják a hagyományos ABAP fejlesztési gyakorlatokat. Ösztönzi a folyamatos tanulás gondolkodásmódját és az ötletek keresztfejlesztését a különböző technológiai halmazokon, ami értékes pozíció az állandó innovációval jellemezhető iparágban.

Összefoglalva, a projekt nemcsak elérte közvetlen célját, a Postman API-tesztek integrálását az ABAP-környezetbe, hanem hozzájárult a cégem fejlesztési gyakorlatok modernizálásának tágabb narratívájához is. Ez az erőfeszítés rugalmasabb, megbízhatóbb és robusztusabb ABAP-alkalmazások előtt nyitja meg az utat, biztosítva, hogy továbbra is kritikus vállalati eszközként szolgálhassanak egy digitálisan összekapcsolt világban.

6. Összefoglalás

Összefoglalva, ez a projekt rávilágított a Postman agilis tesztelési képességei és az ABAP robusztus és integrált tesztelési keretrendszere közötti szinergiára. Egyedi deszerializáló metódusok, dinamikus API-hívási funkciók kifejlesztésével és ezen elemek ABAP tesztelési keretrendszerbe történő zökkenőmentes integrálásával a projekt sikeresen átültette a Postman API-tesztek agilitását az ABAP fejlesztés strukturált világába. Ezzel javítva a külső interfészekkel összekapcsolódó ABAP-alkalmazások minőségét és megbízhatóságát. A saját deszerializáló metódusok írására azért volt szükség, mert bár a dolgozatban egy fajta struktúrát, a periódusokat mutattam be, de a projekt során felmerültek egyéb GET kérések is, amelyek már más eredményt adtak vissza egy teljesen más struktúrával, különféle attribútumokkal. Az ABAP nyelvben egy beépített osztályban található ilyen JSON deszerializáló metódus, ahol bár különféle attribútumok megadásával mi is ki tudunk nyerni egyszerűen információt, de mivel a feladatban az értékekkel számolást is kellett végezni, így problémák merültek fel a különféle struktúrák miatt, hiszen nem lehetett alkalmazni ezt a függvényt, mivel az nem pakolta be a kinyert információt egy belső táblába. A belső tábla létrehozására azért volt szükség, hogy az egyes név/érték párokat ideiglenesen fel tudjam tölteni ilyen táblákba és számítási műveleteket tudjak rajtuk végezni. Ilyen volt a részletesen bemutatott első teszteset, ahol annak kellett visszajönnie, hogy a JSON válaszban 170 különféle név/érték pár szerepel, nem több és nem is kevesebb. A JSON deszerializáló függvényem nem csupán szétválasztva értelmezte ezen név/érték párokat, hanem egy másik segítő függvény használatával fel is töltötte az előre létrehozott struktúrát. Az ideiglenes tábla pontosan olyan attribútumokat tartalmazott, mint a JSON válaszban látható értékek, minden a megfelelő helyen került feltöltésre, innentől pedig a számítási műveletek nagyon leegyszerűsödtek, hiszen csak azt kellett vizsgálni, hogy annyi elem van-e a táblában, mint ahányat elvárunk a GET kérés válaszából. Innentől ha egy teljesen más struktúrát kapunk válaszul, akkor is tudjuk használni ezen függvényeket, csupán az ideiglenesen létrehozott táblánk struktúráját kell módosítani olyan mezőkkel, amiket a válaszban is kapunk, vagy adott esetben új struktúrát is létre tudunk hozni és onnantól kezdve azzal dolgozhatunk tovább, a deszerializálás és a táblafeltöltés működni fog.

A projekt sikere a Postman API tesztek ABAP-környezetben történő replikálásában nem csak technikai vívmány, hanem stratégiai továbbfejlesztése is a cég fejlesztési életciklusának. Az API-tesztek ABAP környezetbe való beépítésével a fejlesztők és tesztelők mostantól átfogó tesztcsomagot hajthatnak végre egyetlen, egységes környezetben, jelentősen csökkentve a környezetváltást és a manuális többletterhelést, amely korábban a külső API-integrációk érvényesítéséhez szükséges volt.

Ezen kívül ez a törekvés hangsúlyozza az alkalmazkodóképesség fontosságát a szoftverfejlesztésben. Az a képesség, hogy a teszteket le lehet fordítani egy modern API-tesztelési platformról, például a Postman programról az ABAP-környezetbe, amely hosszabb múltra tekint vissza és más fókuszú, azt a sokoldalúságot mutatja, amely szükséges ahhoz, hogy a régi rendszerek naprakészek és teljes mértékben működőképesek legyenek a fejlődő külső szolgáltatások és API-k mellett.

Irodalomjegyzék

1. Amazon Web Services. Letöltés dátuma: 2024. 04. 20.

   Forrás: https://aws.amazon.com/what-is/restful-api/

2. Introducing JSON. Letöltés dátuma: 2024. 04. 20.

   Forrás: https://www.json.org/json-en.html

3. Stack Overflow: What is deserialize and serialize in JSON? Letöltés dátuma: 2024. 04. 20.

   Forrás: https://stackoverflow.com/questions/3316762/what-is-deserialize-and-serialize-in-json

4. Reqbin: HTTP GET Request Method. Letöltés dátuma: 2024. 04. 20.

   Forrás: https://stackoverflow.com/questions/3316762/what-is-deserialize-and-serialize-in-json

5. Postman: API Platform. Letöltés dátuma: 2024. 04. 20.

   Forrás: https://www.postman.com/api-platform/

6. SAP Help Portal: ABAP Programming Language – Overview. Letöltés dátuma: 2024. 04. 20.

   Forrás: https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-us/abenabap_overview.htm

7. W. Schwarzmann (N.a): *Designing Testable ABAP Classes And Packages.* SAP Press.
   Forrás: https://www.sap-press.com/designing-testable-abap-classes-and-packages_5466/
   91 oldal.
8. K. Bandari (2022): *Complete ABAP.* SAP Press.
   Forrás: https://www.sap-press.com/complete-abap_5567/
   912 oldal.
9. J. Wood, J. Rupert (2016): *Object-Oriented Programming with ABAP Objects.* SAP Press.
   Forrás: https://www.sap-press.com/object-oriented-programming-with-abap-objects_3597/ 470 oldal.
10. K. Haeuptle, F. Hoffmann, R. Jordao, M. Martin, A. Ravinarayan, K. Westerholz (2021): *Clean ABAP – A Style Guide for Developers.* SAP Press.
    Forrás: https://www.sap-press.com/clean-abap_5190/
    351 oldal.

Ábrajegyzék

Hallgatói nyilatkozat

NYILATKOZAT

a záródolgozat nyilvános hozzáféréséről és eredetiségéről

A hallgató neve: LÁZÁR LEVENTÉ

A Hallgató Neptun kódja: MRWFPP

A dolgozat címe: API hívások tesztjeinek távoli Postman-ből ABAP környezetbe

A megjelenés éve: 2024.

A konzulens intézetének neve: MŰSZAKI INTÉZET

A konzulens tanszékének a neve: ALKALMAZOTT INFORMATIKA TANSZÉK

Kijelentem, hogy az általam benyújtott záródolgozat egyéni, eredeti jellegű, saját szellemi alkotásom. Azon részeket, melyeket más szerzők munkájából vettem át, egyértelműen megjelöltem, és az irodalomjegyzékben szerepeltettem.

Ha a fenti nyilatkozattal valótlant állítottam, tudomásul veszem, hogy a záróvizsga-bizottság a záróvizsgából kizár és a záróvizsgát csak új dolgozat készítése után tehetek.

A leadott dolgozat, mely PDF dokumentum, szerkesztését nem, megtekintését és nyomtatását engedélyezem.

Tudomásul veszem, hogy az általam készített dolgozatra, mint szellemi alkotás felhasználására, hasznosítására a Magyar Agrár- és Élettudományi Egyetem mindenkori szellemitulajdon-kezelési szabályzatában megfogalmazottak érvényesek.

Tudomásul veszem, hogy dolgozatom elektronikus változata feltöltésre kerül a Magyar Agrár- és Élettudományi Egyetem MATER Hallgatói Dolgozatok repozitoriumába. Tudomásul veszem, hogy a megvédett és

- nem titkosított dolgozat a védést követően
- titkosításra engedélyezett dolgozat a benyújtásától számított 5 év eltelte után

nyilvánosan elérhető és kereshető lesz az Egyetem MATER Hallgatói Dolgozatok repozitoriumában.

Kelt: 2024 év ÁPRILIS hó 20 nap

Hallgató aláírása

Konzulensi nyilatkozat

NYILATKOZAT

LÁZÁR LEVENTE (név) (hallgató Neptun azonosítója: MRWFFP) konzulenseként nyilatkozom arról, hogy a záródolgozatot áttekintettem, a hallgatót az irodalmi források korrekt kezelésének követelményeiről, jogi és etikai szabályairól tájékoztattam.

A záródolgozatot a záróvizsgán történő védésre **javaslom** / **nem javaslom**[1].

A dolgozat állam- vagy szolgálati titkot tartalmaz: igen nem*[2]

Kelt: 2024 év ÁPRILIS hó 16 nap

belső konzulens