



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK

TANSZÉK

## RefactorErl elemző képességeinek integrálása az Erlang LS nyelvi kiszolgálóba

*Témavezető:*

Tóth Melinda, Bozó István

Egyetemi docens, Egyetemi adjunktus

*Szerző:*

Fikó Róbert

programtervező informatikus BSc

*Budapest, 2022*

## SZAKDOLGOZAT TÉMABEJELENTŐ

**Hallgató adatai:**

Név: Fikó Róbert

Neptun kód: G55OFZ

**Képzési adatok:**

Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)

Tagozat : Nappali

Belső témavezetővel rendelkezem

*Témavezető neve: Dr. Tóth Melinda, Bozó István*

*munkahelyének neve, tanszéke: ELTE-IK, Programozási nyelvek és Fordítóprogramok Tanszék*

*munkahelyének címe: 1117, Budapest, Pázmány Péter sétány 1/C.*

*beosztás és iskolai végzettsége: Egyetemi docens (PhD), egyetemi adjunktus (PhD)*

**A szakdolgozat címe:** RefactorErl elemző képességeinek integrálása az Erlang LS nyelvi kiszolgálóba

**A szakdolgozat témája:**

*(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását )*

Az Erlang egy futási idejű szemétygyűjtővel ellátott, konkurens funkcionális programozási nyelv. A nyelv széleskörben elterjedt a telekommunikációs alkalmazásokban, köszönhetően a gazdag OTP (Open Telecom Platform) futási-idejű környezetnek, ami rengeteg használatra kész komponenst tartalmaz. Az Erlang alkalmas nagymennyiségű adatfeldolgozásra, mert képes elosztott módban futni, továbbá hibákkal is igen toleráns.

A RefactorErl az ELTE Informatikai karán futó projekt, mely a Erlang nyelven írt források elemzésével és átalakításával foglalkozik. Legfontosabb célkitűzések a kódmegértés támogatása, a programozó segítése biztosabb, stabilabb Erlang programok írásában. Az eszköz erőssége a hatékony gráf-reprezentációban és a széleskörű szemantikus lekérdező nyelvben rejlik.

Az Erlang Language Server, vagyis nyelvi kiszolgáló (továbbiakban ELS), a Microsoft Language Server Protokolljára épülő cross-platform és cross-editor elemző rendszer, ami a legtöbb népszerű kódszerkesztővel kompatibilis. Annak ellenére, hogy az ELS egy hasznos elterjedt eszköz, a RefactorErl elemző lehetőségeivel még többre lenne képes.

Szakdolgozatomban a RefactorErl diagnosztikai és információ nyújtási képességeit kívánom integrálni az ELS nyelvi kiszolgálóba. A Language Server, az LSP protokollnak köszönhetően már sok népszerű fejlesztői környezetben támogatott, továbbá a protokollban jól definiált megjelenítési és bemenet adási lehetőségek rejlenek, aminek segítségével a programozó a szerkesztőjéből közvetlenül tud egyedi lekérdezéseket adni interaktív módon, továbbá az esetleges hibáit is egy helyen látja a forrásban.

Munkám során elsősorban a VSCode fejlesztői környezetben való alkalmazhatóságra koncentrálok. Azonban az ELS már támogatja az Emacs, SublimeText, Vim környezeteket is, így a RefactorErl funkcionalitásai itt is elérhetőek lesznek a későbbiekben.

Budapest, 2021. 11. 16.

# Tartalomjegyzék

# Köszönetnyilvánítás

I like to acknowledge ...

# 1. fejezet

## Bevezetés

### 1.1. Statikus kódelemzés, illetve refaktorálás

Mi az a kód elemzés? Nem a programozó feladata a kódot megérteni? Ezen kérdések mind felmerülhetnek bennünk. A kódelemző egy olyan szoftver, ami képes a forráskód elemzésére, annak futtatása nélkül. Ugyan egy kicsi projektben egy elemző program használata még nem feltétlen szükséges, azonban egy ipari méretű szoftver esetében a programozók feladatát nagyban megkönnyítik.

Egy statikus elemző szoftver segítségével rengeteg információt tudhatunk meg még a fejlesztés során, mint például: *kód-metrikákat* és *függőségi információkat*, melyek olyan kódrészek tekintében felettébb fontosak, amiket nem mi írtunk.

Szintén nagy méretű projekteknél fontos a refaktoráló, azaz kód átalakító eszközök használata is, hiszen manuálisan egy függvény vagy egy változó átnevezése igen nagy kihívást jelentene. Azonban nem csak ilyen méretű szoftvereknél hasznos, amit esetlegesen több ember is használt, hanem egy projekt fejlesztésében, fejlődésében is jelentősége van egy ilyen, refaktoráló eszköznek. Gondoljunk bele: elkezdünk egy projektet, (ami lehet egy független szoftver, vagy akár egy új komponens egy már meglévő projektben) először egy prototípust készítünk belőle, majd az alapján folytatjuk a fejlesztést, jellemzően már egy tervet követve, azonban a legalaposabb tervezés ellenére is előfordulhat, hogy egy kódot át kell alakítani.

## 1.2. RefactorErl

A RefactorErl az ELTE Informatikai Karán futó kutatás-fejlesztési projekt melynek fő célja az, hogy Erlang forrásokhoz statikus elemző szoftvert készítsen. Az eszközzel az elemzés mellett kódátalakítást, refaktorálást is elvégezhezünk. Főbb funkcionálisáiként kiemelném a: függőségi gráf vizualizációját, illetve a szemantikus lekérdező nyelvét, amelyen keresztül többek között olyan információk érhetőek el, mint nem használt makró definíciók, változó lehetséges értékei vagy biztonsági sérülékenységek [1]. Az eszköznek jelenleg több felhasználói felülete van, amelyből kiemelném a webes felületet és a Erlang Shell-en keresztül elérhető parancssori felületét.

## 1.3. Fejlesztő környezeti nyelvi támogatás

Napjainkban nagyon elterjedtek az integrált fejlesztői környezetek, avagy IDE<sup>1</sup>-k. Az ilyen fejlesztői környezetben egy szoftveren belül tudjuk szerkeszteni a forráskódot, konzolt kezelni, verziót kezelni, s a kódukról diagnosztikákat is kapunk, szinte valós időben, amennyiben az adott programozási nyelvhez rendelkezésre áll ilyen IDE-kompatibilis rendszer. (*Például: Visual Studio, IntelliJ, Emacs*)

A nyelvi támogatás nem jelent mást, mint egy a fejlesztői környezetbe integrált szoftvert, ami segíti a programozó munkáját a hibák és figyelmeztetések vizualizációjában, kód-kiegészítési javaslatokat is tesz, illetve a dokumentáció egy részét is elérhetővé teszi a szerkesztőn belül (*Például: az adott függvény paraméterezése, leírása*)

Sok esetben az ilyen nyelvi támogató rendszerek kifejezetten egy rendszerhez készültek, egy másik szerkesztőben pedig nem használhatóak, pedig a forrást ugyanúgy kell elemezni és a fejlesztőnek is közel azonos diagnosztikákat, visszajelzéseket kell nyújtani. Erre ad megoldást a Microsoft Language Server Protocolja (LSP), ami egy fejlesztői környezet független nyelvi kiszolgáló protokollt jellemez. Ennek az előnye, hogy a kiszolgálót csak egyszer kell megírni, és onnantól kezdve könnyedén integrálható, bármely kliensbe (amennyiben az megvalósítja a protokollt)

---

<sup>1</sup>*Integrated Development Environment*

## 1.4. Erlang LS

Az Erlang Language Server (röviden: Erlang LS) a nyelvi kiszolgáló az Erlang programozási nyelvhez. *Roberto Aloï* vezetésével indult el a nyílt forráskódú projekt, aminek mára már jelentős karbantartói és fejlesztői csapata van, a világ minden pontjáról, Magyarországról is. Az Erlang LS alapvetően egy szerkesztő független megvalósítás, azonban a csapat a Visual Studio Code-hoz fejlesztett egy interfész kiegészítőt is, ami segítségével az LS, szépen betud épülni a szerkesztőbe. Az elemző eszköz elérhető továbbá például Emacs, IntelliJ és még sok más IDE-ben is. Az Erlang LS támogatja a DAP<sup>2</sup>-ot és a legfrissebb kiadása már a RefactorErl diagnosztikák integrációját is támogatja.

## 1.5. Visual Studio Code

A Visual Studio Code manapság az egyik legelterjedtebb kódszerkesztő szoftver, ami köszönhető nyílt forrásának, illetve annak, hogy elérhető macOS, Linux és Windows operációs rendszereken, sőt akár (béta verzióban) az interneten is. Az eszköz beépített grafikus verziókezelést és integrált terminált is biztosít, a plugin<sup>3</sup> architektúrája végett pedig szinte a kedvünkre bővíthető és személyre szabható<sup>4</sup>.

## 1.6. Feladat

A feladat egy olyan alkalmazás csomagot készíteni, melynek komponensei egymással képesek együtt működni, s ezáltal az Erlang nyelven fejlesztők munkáját megkönnyíteni. A funkcionálisokat két felé kell csoportosítani: amiket meg lehet valósítani az Erlang LS-ben és amiket nem. Sajnos amiket nem tudunk megvalósítani a Language Server Protocol felett, azok csak a Visual Studio Code szerkesztőben fognak működni, egy önálló bővítmény segítségével.

Az alkalmazás jelenítsen meg diagnosztikákat és kód vizualizációkat a forrásról. Lehesse egyszerűen függőségi információkat kinyerni modul és függvény szinten. Továbbá legyen lehetőség beépített és egyedi lekérdezések futtatására, illetve azok megjelenítésére.

---

<sup>2</sup>Debugger Adapter Protocol

<sup>3</sup>kiegészítő, beépülő modul

<sup>4</sup>Ezen kiegészítőket az alkalmazáson belül elérhető Marketplace-ből tudjuk letölteni.

## 2. fejezet

# Felhasználói dokumentáció

A következő fejezetben bemutatom, hogy az általam fejlesztett kiegészítőt, illetve a kiegészített szoftvereket hogyan kell telepíteni, beüzemelni, illetve használni, képekkel illusztrálva.

### 2.1. Rendszerkövetelmények

- Hardver
  - 2 GHz vagy nagyobb órajelű processzor
  - 2 GB RAM memória
  - 1 GB lemezterület a RefactorErl, Visual Studio Code, Erlang LS, illetve a bővítmény számára (forráskódok betöltéséhez további tárhely és memória szükséges)
- Szoftver
  - mac OS El Capitan, Windows 10 vagy Linux operációs rendszer
  - Telepített RefactorErl elemző eszköz
  - GraphViz 2.38 (a függőségi gráf rajzolásához)
  - Erlang/OTP 22 vagy újabb
  - Visual Studio Code 1.67.0 (Universal)
  - GCC 4.7.2 vagy újabb fordítóprogram



## 2.2. Telepítés

A fejlesztés mac OS operációs rendszeren történt, így a legtöbb helyen a telepítési útmutató részletesebb leírást tartalmaz ezen operációs rendszerhez. Azonban a mind az újonnan fejlesztett komponensek (Visualiser kiegészítő) és a RefactorErl, illetve az Erlang LS működik Windows és Linux operációs rendszerek alatt is.

### 2.2.1. Erlang/OTP telepítése

Mind a RefactorErl futtatásához, mind az Erlang LS futtatásához szükségünk van az Erlang virtuális gép telepítéséhez. Ezt macOS operációs rendszer legegyszerűbben a Homebrew [2] nevű csomagkezelővel tehetjük meg az alábbi parancs kiadásával:

```
brew install erlang
```

Linux és Windows rendszerekhez az Erlang honlapjáról [3] tájékozódhatunk a telepítő parancsokról, illetve innen tölthetjük le a telepítő állományt is.

### 2.2.2. GCC telepítése

A GNU/GCC fordító szükséges az Erlang és a RefactorErl egyes komponenseinek fordításához, így telepítése szükséges. Ezt legegyszerűbben szintén az operációs rendszerünk csomagkezelőjével megtehetjük. Mac OS operációs rendszer alatt, itt is használhatjuk a Homebrew [2] csomagkezelőt, az alábbi paranccsal: `brew install gcc`

További információért keressük fel a GNU/GCC hivatalos oldalát. [4].

### 2.2.3. Graphviz telepítése

A GraphViz telepítése opcionális, azonban a gráfok kirajzolásához szükséges. Ezt szintén legegyszerűbben az operációs rendszerünk csomagkezelőjével tehetjük meg. Amennyiben további információra van szükségünk, keressük fel a GraphViz oldalát [5], ahol telepítő csomagok és parancsok is rendelkezésre állnak. Mac OS operációs rendszer alatt, használhatjuk az alábbi parancsot: `brew install graphviz`

### 2.2.4. Yaws webservertelepítése

A YAWS<sup>1</sup> forráskódja letölthető és telepíthető a GitHub oldalukról [6], ahol a README fájlban megtalálhatóak a telepítéshez szükséges utasítások.

Mac OS operációs rendszer alatt használhatjuk a Homebrew csomagkezelőt [2] is, az alábbi parancs kiadásával: `brew install yaws`

### 2.2.5. RefactorErl telepítése

A RefactorErl hivatalos kiadása letölthető az eszköz honlapjáról [7], azonban a *szakdolgozat készítésének időpontjában a publikus verzióban a szükséges modulok még nem elérhetőek, de a következő kiadásban jelen lesznek*. Természetesen a szükséges modulokat tartalmazó verzió megtalálható a szakdolgozat mellékleteként.

Telepítéshez csomagoljuk ki a RefactorErl forrását tartalmazó tömörített állományt, majd lépünk be a gyökérkönyvtárába.

Telepítés Windowson: `bin\referl -build tool -yaws_path path\to\yaws` Telepítés

Linuxon: `bin/referl -build tool -yaws_path path/to/yaws`

Ahol a `-yaws_path` kapcsoló utána a YAWS ebin könyvtárának elérhetőségét kell megadni.

### 2.2.6. Visual Studio Code telepítése

A Visual Studio Code egy cross-platform<sup>2</sup> fejlesztői környezet, ami igen gazdag fejlesztői interfésszel rendelkezik, ami lehetővé tette ezt a fejlesztést is. Megfelelő operációs rendszer kiválasztása után letölthető a termék honlapjáról [8].

### 2.2.7. RefactorErl Visualiser telepítése

A Visualiser bővítmény letölthető a bővítmény GitHub oldaláról: <https://github.com/robertfiko/refactorerl-visualiser>. Ezután a `make visualiser` paranccsal fordítható és telepíthető, de javasolt inkább a mellékelt állományok között is megtalálható a `.vsix` kiterjesztésű telepítő fájl használata.

A telepítéshez menjünk a bővítmények menübe a Visual Studio Code-on belül. Ezt a bal oldali oldalmenüben találjuk meg. Majd a megjelenő oldalsáv jobb felső

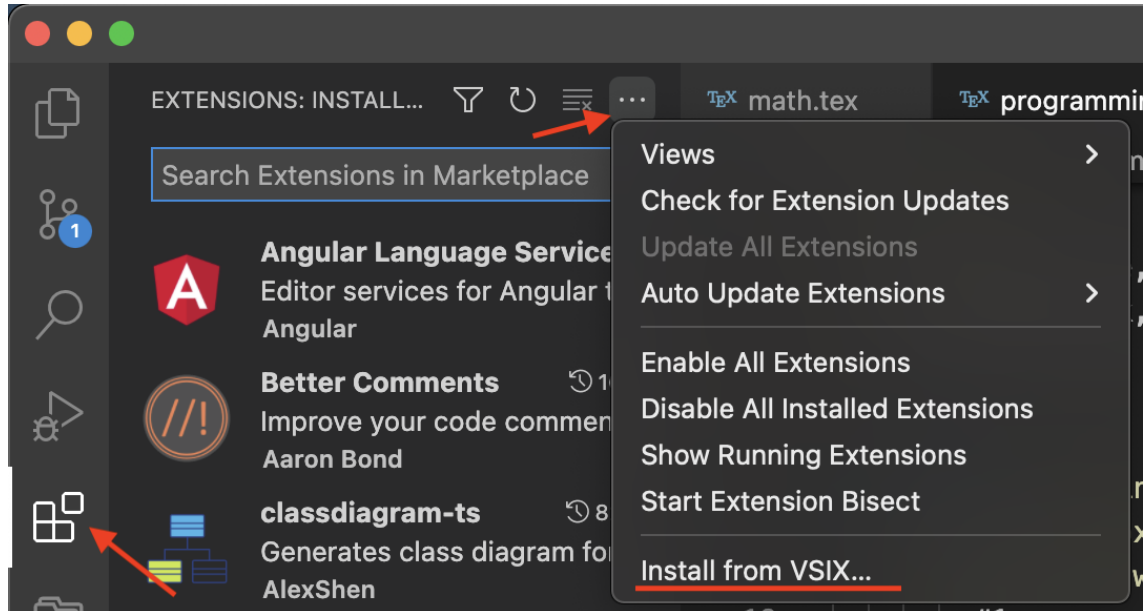
---

<sup>1</sup>Yet Another Web Server

<sup>2</sup>többféle rendszeren képes futni

sarkából a három pontra (...) kattintva válasszuk az "Install from VSIX ..." opciót, majd a felugró ablakban keressük ki a VSIX fájlt. (lásd: 2.1 ábrán)

*Megjegyzés: amennyiben rendelkezünk GNU/MAKE-el abban az esetben a `make refels` paranccsal az Erlang LS bővített verziója is telepíthető a Git tárolójából, de nem ezt az eljárás módot fogjuk követni.*



2.1. ábra. Telepítés .vsix fájlból

## 2.2.8. Erlang LS bővített változatának telepítése

Az Erlang Language Server *eredeti verziója* megtekinthető a GitHub tárolójában<sup>3</sup>, illetve letölthető a Visual Studio Code Marketplace<sup>4</sup>-ről.

A bővített verzió forráskódja megtalálható a dolgozat csatolmányai között, s az ebből előálló .vsix fájl is. Ezt a már korábban a 2.1 ábrán mutatott módon telepíthetjük.

## 2.3. Erlang LS használata

### 2.3.1. Első elindításkor szükséges beállítási lehetőségek

Az első elindulás előtt az Erlang LS konfigurációs [9] fájljában néhány módosítást kell végeznünk, hogy az megfelelően illeszkedjen a RefactorErlhez. A konfigurációs

<sup>3</sup>[https://github.com/erlang-ls/erlang\\_ls](https://github.com/erlang-ls/erlang_ls)

<sup>4</sup><https://marketplace.visualstudio.com/items?itemName=erlang-ls.erlang-ls>

fájl YAML formátumot követ. A diagnosztikák futtatásához vegyük fel a `refactorer1` elemet az engedélyezettek közé.

```
1 diagnostics:
2   enabled:
3     - refactorer1
```

### 2.1. forráskód. RefactorErl diagnosztikák engedélyezése Erlang LS-ben

*Erlang node-nak nevezzük az Erlang virtuális gép egy példányát. Gyakran csak egyszerűen node-ként fogok rá hivatkozni. A RefactorErl node alatt pedig azt az Erlang virtuális gép példányt értjük, amin az eszköz fut.*

Ezzel a beállítással az Erlang LS már futtatni fogja a diagnosztikákat, azonban alapértelmezetten egyetlen diagnosztikai sem fog futni, ezeket manuális állíthatjuk be. Ehhez vegyük fel egy `refactorer1` kulcsot a fájl gyökerébe. Ez alá két *alkulcs* kerülhet:

- **node:** ide annak az Erlang node-nak a nevét kell megadni szöveges karakterláncként, ahol a RefactorErl fut.
- **diagnostics:** azon diagnosztikák azonosítói, amelyeket futtatni szeretnénk listaként (felsoroláskén) megadva. A diagnosztikák azonosítói szintén szöveges karakterláncok.

Példa egy ilyen konfigurációs fájl részletre:

```
1 diagnostics:
2   enabled:
3     ...
4     - refactorer1
5
6   ...
7
8 refactorer1:
9   node: "nodeName@hostName"
10  diagnostics:
11    - "unused_macros"
12    - "unsecure_os_call"
```

### 2.2. forráskód. RefactorErl konfigurációs példa Erlang LS-ben

Az alábbi diagnosztikákból tudunk válogatni jelenleg:

Lekérdezés neve	Lekérdezés rövid ismertetése
<code>unused_macro</code>	Nem használt makró definíciók megjelenítése
<code>unsecure_calls</code>	Az összes lehetséges támadási forma megjelenítése
<code>unsecure_interoperability</code>	Az együttműködési képességből fakadó sebezhetőségek azonosítása.
<code>unsecure_concurrency</code>	A konkurens programozásból eredő hibalehetőségek feltérképezése.
<code>unsecure_os_call</code>	Az ismeretlen helyről származó paraméterekkel meghívott OS szintű utasítások ellenőrzése.
<code>unsecure_port_creation</code>	A portok létrehozásával kapcsolatos sebezhetőségek megjelenítése.
<code>unsecure_file_operation</code>	Az ismeretlen bemenettel meghívott fájlkezeléssel kapcsolatos műveletek megjelenítése.
<code>unstable_call</code>	Az atomok dinamikus létrehozásával kapcsolatos függvények feltérképezése.
<code>nif_calls</code>	A NIF függvények használatából fakadó sebezhetőségek ellenőrzése.
<code>unsecure_port_drivers</code>	A dinamikusan betölthető könyvtárak használatából fakadó veszélyek azonosítása.
<code>decommissioned_crypto</code>	Az elavultnak számító kriptográfiai műveletek megjelenítése.
<code>unsecure_compile_operations</code>	Az ismeretlen helyről származó programkód fordításának és betöltésének ellenőrzése.
<code>unsecure_process_linkage</code>	A folyamatok nem megfelelő összekapcsolásából adódó sebezhetőségek megjelenítése.

<code>unsecure_prioritization</code>	A folyamatok prioritásának módosításából fakadó veszélyek ellenőrzése.
<code>unsecure_ets_traversal</code>	Az ETS tábla rögzítés nélküli bejárásának ellenőrzésére szolgáló megjelenítése.
<code>unsafe_network</code>	A hálózati rendszermaggal kapcsolatos műveletek feltérképezése.
<code>unsecure_xml_usage</code>	Az ismeretlen helyről származó xml paraméterek elemzésével kapcsolatos függvények azonosítása.
<code>unsecure_communication</code>	Az elosztott hálózat szereplői között zajló kommunikációs beállítások ellenőrzése.

2.1. táblázat. Elérhető diagnosztikai azonosítók listája.

### 2.3.2. A RefactorErl konfigurálása

Ahhoz megfelelő válaszdíóvel és teljesítménnyel tudjunk dolgozni a Erlang LS-ben RefactorErl-el vagy a Visualiser bővítménnyel érdemes már egy konfigurált adatbázist készíteni, hogy az elemzések zömét mér elvégezze az eszköz, s használat közben csak esetleges szinkronizációkat kelljen végezni, amikor csak a változtatásokat kell leelemezni. A RefactorErl használatának részletei megtekinthetők az eszköz Wiki oldalán. [10]

Az adatbázishoz az `ri:add` függvény segítségével adhatunk fájlokat. Megadhatunk paraméterként egy fájlnevet karakterláncként, vagy úgynevezett atomként<sup>5</sup>. Ebben az esetben a lokális útvonal a jelenlegi könyvtár, amiben dolgozunk. Természetesen globális útvonalként is megadhatunk fájlokat. Amennyiben könyvtárat adunk meg, akkor az adott könyvtárat rekurzívan bejárva az összes fájl hozzá lesz adva. Néhány példa a használatra [11]:

```
cd(dir), ri:add(modname).  
  
ri:add('dir/modname').  
  
ri:add("dir/modname.erl").
```

---

<sup>5</sup>Az Erlang nyelvben atomnak nevezünk egy olyan literált, aminek értéke megegyezik a nevével, amit egyfajta szövegkonstansnak is nevezhetünk.

```
ri:add("path_to_dir/dir").
```

Ezen kívül az `ri:ls()`. paranccsal lekérhetjük a beolvasott fájlok listáját, az `ri:reset()`. segítségével pedig törölhetjük az adatbázist.

### 2.3.3. Diagnosztikák használata

Az Erlang LS a diagnosztikákat az adott szerkesztőben figyelmeztetésként jeleníti meg. Ugyan ez a funkció **minden ELS-el kompatibilis szerkesztőben meg fog jelenni**, most a példák során a Visual Studio Code szerkesztőre fogunk fókuszálni. Miután a konfigurációs fájlban mindent beállítottunk (ld. ?? bekezdés) nincs más dolgunk, mint betölteni egy forráskódot és megtekinteni az eredményt.

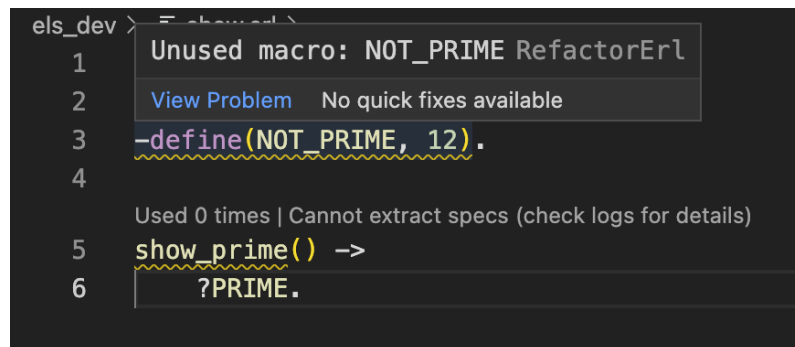
#### 2.3.3.1. Példa: Nem használt makró definíciók

Az alábbiakban a `unused_macros` azonosítójú diagnosztikát fogjuk áttekinteni. Ehhez nézzük meg alábbi a forrást:

```
1 -module(show).  
2 -define(PRIME, 7).  
3 -define(NOT_PRIME, 12).  
4  
5 show_prime() ->  
6     ?PRIME.
```

2.3. forráskód. Nem használt makró definíciókkal ellátott szemléltető kód

Itt láthatjuk, hogy a `NOT_PRIME` makró nincs használatban. Ha megnézzük, milyen diagnosztikákat kaptunk a RefactorErltől, az Erlang LS-en keresztül akkor láthatjuk, hogy figyelmesztetés szintű diagnosztikát jelez nekünk.



2.2. ábra. Nem használt makró diagnosztika Visual Studio Code-ban

2.3.3.2. Példa: Nem biztonságos `os` hívás diagnosztika

```

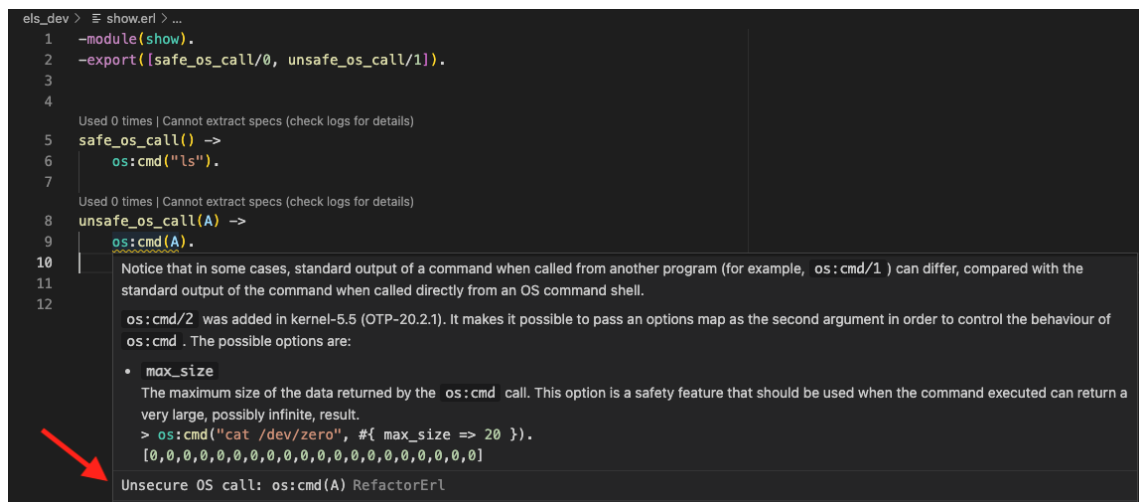
1 -module(show).
2 -export([safe_os_call/0, unsafe_os_call/1]).
3
4 safe_os_call() ->
5     os:cmd("ls").
6
7 unsafe_os_call(A) ->
8     os:cmd(A).

```

2.4. forráskód. Nem biztonságos `os` hívást szemléltető kód

A fenti forrásban ötödik sorban lévő `os:cmd` hívást nem tekintjük veszélyforrásnak, hiszen a fejlesztő maga paraméterezi fel. Azonban hetedik sorban kezdődő `unsafe_os_call` definíciót sérülékenységi pontnak tekintjük, hiszen paramétere ismeretlen helyről származik [1].

Ennek megfelelően a 2.3 ábrán láthatjuk is, hogy a szerkesztő program figyelmeztetést is adott a szóbanforgó kódrészletre.

2.3. ábra. Nem biztonságos `os` hívás diagnosztika Visual Studio Code-ban

Ezen fejezetnek nem célja bemutatni az összes lehetséges diagnosztikát, csupán egy átfogó képet adni, azok használatáról. Továbbiakban az felhasználóra van bízva, hogy mely diagnosztikát szeretné használni. Az integrált diagnosztikák jelentős része Baranyai Brigitta TDK dolgozatából származik, ahol további részletekről is olvashatunk [1].



### 2.3.4. Kódaakció parancsok kiadása

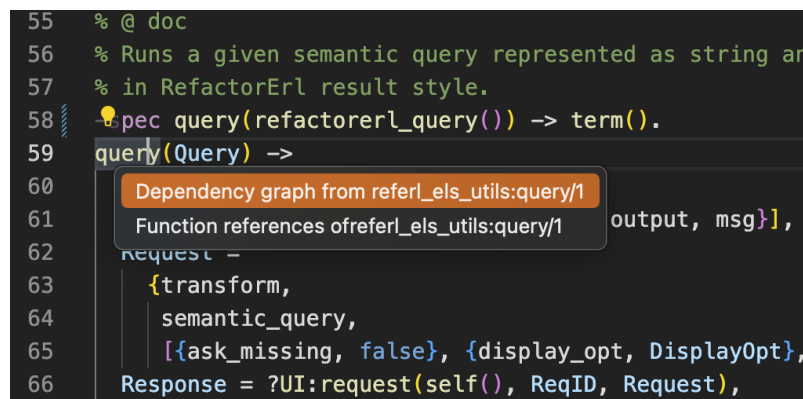
Egy változó *Origin* értékének vagy eredetének azt mondjuk, hogy melyek azok a lehetséges értékek, amelyek a kiértékelés során belefolyhatnak. [12] [13]

Egy változó *Reach* értékén azt értjük, hogy az adott változó értéke mely helyekre folyhat be, ahol az ki lesz értékelve. [12] [13]

A kódaakciók, avagy angolul *Code Actions* a fejlesztői környezetben megjelenő gyors javítások, refaktorálások és tanácsok. Jelenleg inkább az első kettő megközelítés az elterjedt, ahogyan azt a Visual Studio Code felhasználói útmutatója is írja. [14]. Ebben a dolgozatban ezen funkciót kissé kiterjesztve, kód értelmezési funkcionalitást kapott. Kódaakcióként elérhetőek az alábbi funkciók:

- Függőségi gráf lekérése adott függvényből vagy modulból kiindulva
- Változó *Origin* értékének lekérése
- Változó *Reach* értékének lekérése
- Függvény (dinamikus) hivatkozásainak lekérése

Ugyan a fent felsorolt funkcionalitások az Erlang LS felületéről indulnak, azonban sajnos a *Language Server Protocol* limitációi miatt, ezen funkciók, lekérdezések megjelenítésére nincsen lehetőség [15]. Ezen limitációk áthidalására készült a RefactorErl Visualiser bővítmény. A lekérdezések eredményének megjelenítéséről gráf rajzolása esetében 2.4.3 fejezetben, míg a Reach, Origin és függvények hivatkozás esetében a 2.4.4.2 fejezetben olvashatunk.

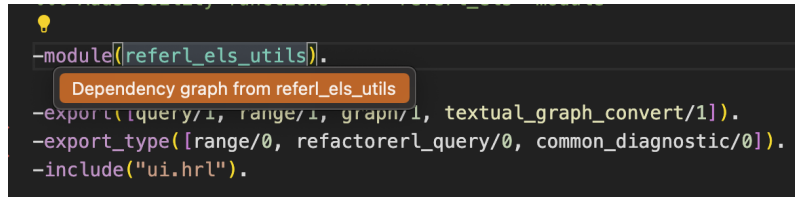


```

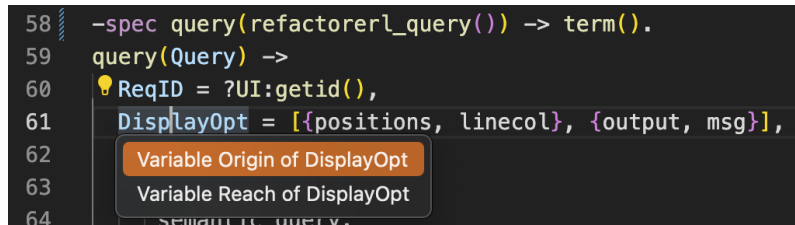
55  % @ doc
56  % Runs a given semantic query represented as string and
57  % in RefactorErl result style.
58  -spec query(refactorerl_query()) -> term().
59  query(Query) ->
60      {
61          Dependency graph from referl_els_utils:query/1
62          Function references of referl_els_utils:query/1
63          request =
64              {transform,
65               semantic_query,
66               [{ask_missing, false}, {display_opt, DisplayOpt}],
67               Response = ?UI:request(self(), ReqID, Request),

```

2.4. ábra. Függvényhez kapcsolódó kódaakciók



2.5. ábra. Modulhoz kapcsolódó kódakciók



2.6. ábra. Változóhoz kapcsolódó kódakciók

A kódakció elvégzéséhez a kurzorral álljunk a változó, függvény vagy a modul definíció fölé és a megjelenő lámpa ikonra kattintva válasszuk ki a megfelelő opciót a helyi menüből. (lásd: 2.4, 2.5 és 2.6 ábrák)

*Dependenc graph from Example* opció a függőségi gráfot fogja kirajzolni a megfelelő kiindulási pontból.

*Variable Origin/Reach of Example* opciók rendre a változó Origin, illetve Reach értékét adják vissza.

Amikor kódakcióból indítunk beépített lekérdezést, akkor az adott fájl amiből indítjuk, hozzá lesz adva az adatbázishoz, vagy szinkronizáció el lesz végezve, de csak arra az adott fájlra, ahonnan a parancsot kiadjuk.

## 2.4. Visualiser kiegészítő használata

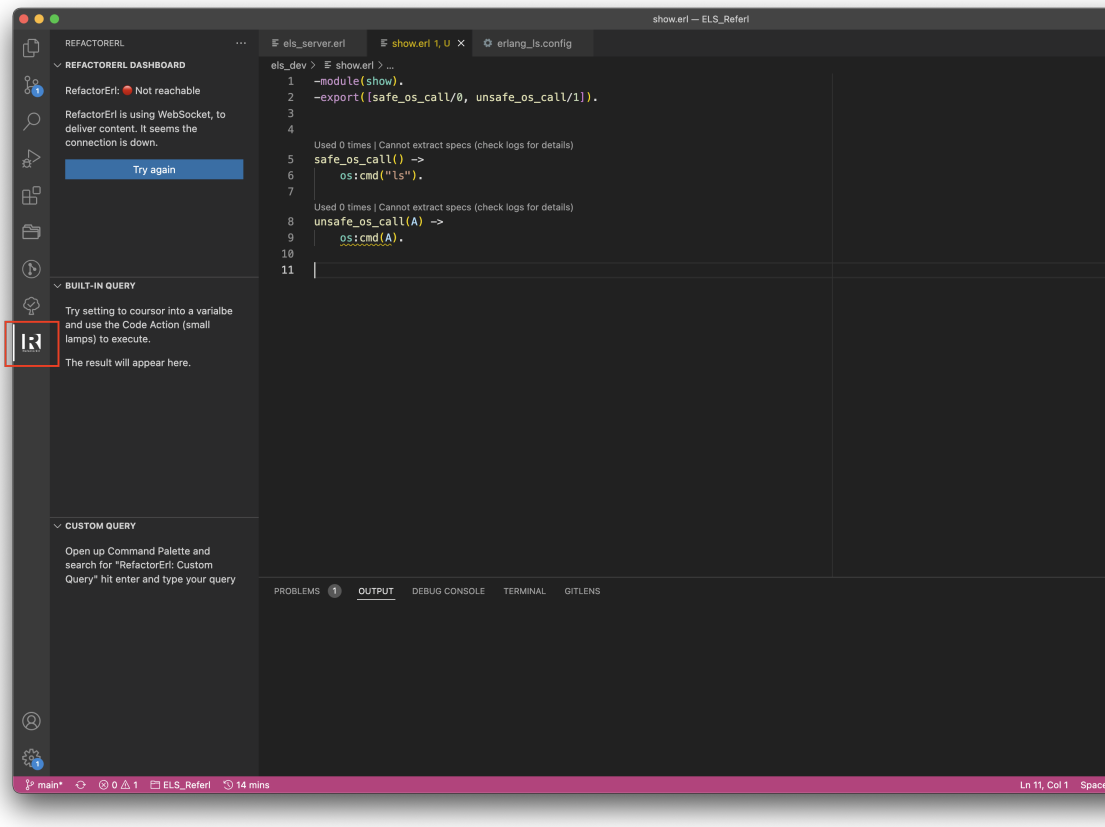
### 2.4.1. Elindítása

Az elindításhoz nincsen más teendőnk, mint az oldalsávon megjelenő RefactorErl logóra kattintani (lásd: 2.7 ábra), továbbá a RefactorErl node-on az alábbi parancsot kiadni:

```
referl_vsc:start(). vagy ri:start_vsc().
```

```
referl_vsc:stop(). vagy ri:stop_vsc().
```

 parancsokkal pedig leállíthatjuk.

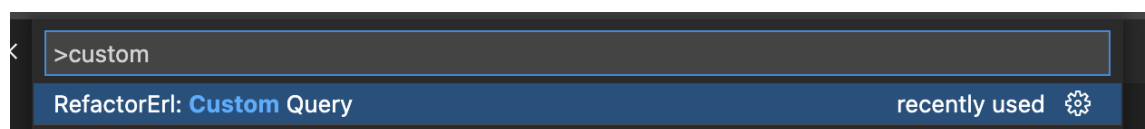


2.7. ábra. RefactorErl Visualiser indítása

### 2.4.2. Egyéni szemantikus lekérdezések

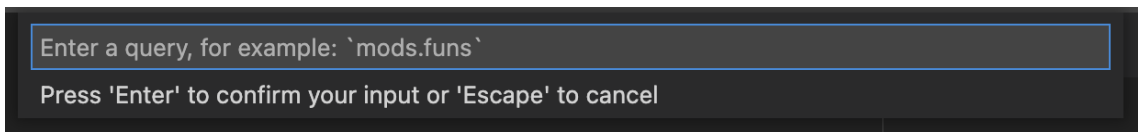
A kiegészítő segítségével egyedi szemantikus lekérdezéseket is futtathatunk a RefactorErl adatbázisában tárolt fájlokon. A szemantikus lekérdezéseket az eszköz lekérdező nyelvének segítségével foglalmazhatjuk meg, melyről bővebb információt a projekt wiki oldalán találhatunk [16]. [17]

Ahhoz, hogy egy ilyen lekérdezést futtassunk az ún. *Command Palette*-et nyissuk meg, amit mac OS-en a `CMD + SHIFT + P`, Windows-on és Linux operációs rendszeren pedig a `CTRL + SHIFT + P` billentyűk egyidejű lenyomásával tudunk elő hozni. Itt kezdjük el gépelni a parancs nevét: *Custom Query*, majd a megjelenő lehetőségek közül válasszuk ki a *RefactorErl* prefixxel ellátott opciót. (lásd: 2.8 ábra)



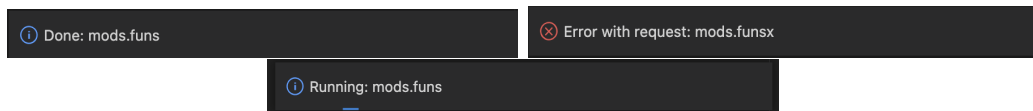
2.8. ábra. Custom Query megnyitása a Command Palette felületéről

Ez után a *Command Palette* helyén megjelenik egy beviteli mező, ahova beírhatjuk a lekérdezést. (lásd: 2.9 ábra)



2.9. ábra. Beviteli mező a Query számára

A lekérdezést az *Esc* billentyű lenyomásával megszakíthatjuk, azt *Enter* segítségével pedig elküldhetjük azt. A lekérdezés állapotáról a jobb alsó sarokban kaphatunk információt. (lásd: 2.10 ábra)



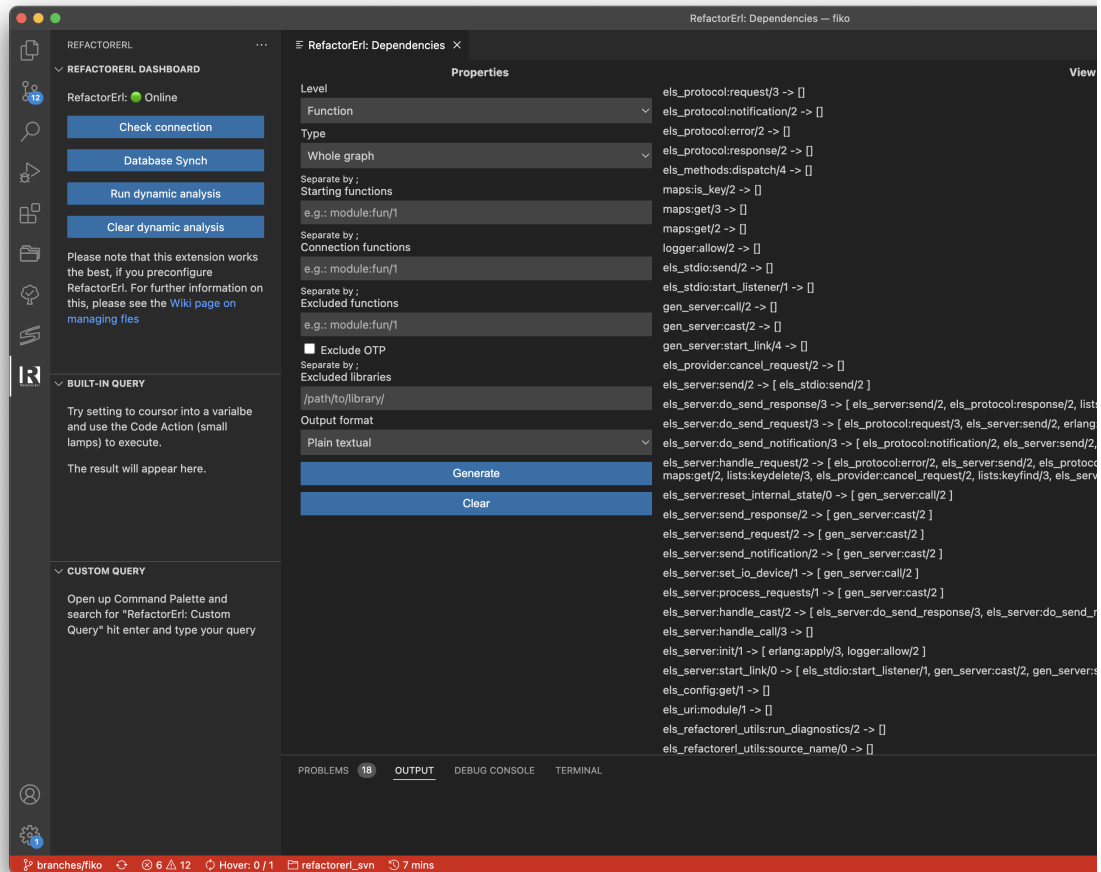
2.10. ábra. Lehetséges értesítések lekérdezés esetén

Fontos kiemteni, hogy a kódakciókkal ellentétben, az egyedi lekérdezéseknél semmiféle adatbázis szinkronizáció nem történik. Amennyiben szeretnénk egy fájlt hozzáadni, úgy a 2.3.2 fejezetben tárgyaltak szerint megtehetjük, vagy használhatjuk a bővítmény szinkronizáló funkcióját a 2.4.4.1 fejezetben leírtak alapján. Az egyedi lekérdezések megjelenítését a 2.4.4.3 fejezetben fogjuk tárgyalni.

### 2.4.3. Függőségi gráf rajzolása

Függőségi gráfot tudunk rajzolni kódakciók segítségével, illetve a *RefactorErl: Dependencies* nézet segítségével, itt az utóbbit fogjuk megtekinteni.

A nézet előhozásához a *Command Palette*-n írjuk be, hogy: *Dependenc Graph Visualiser*. Majd meg is jelenik a nézet. (lásd 2.11 ábra)



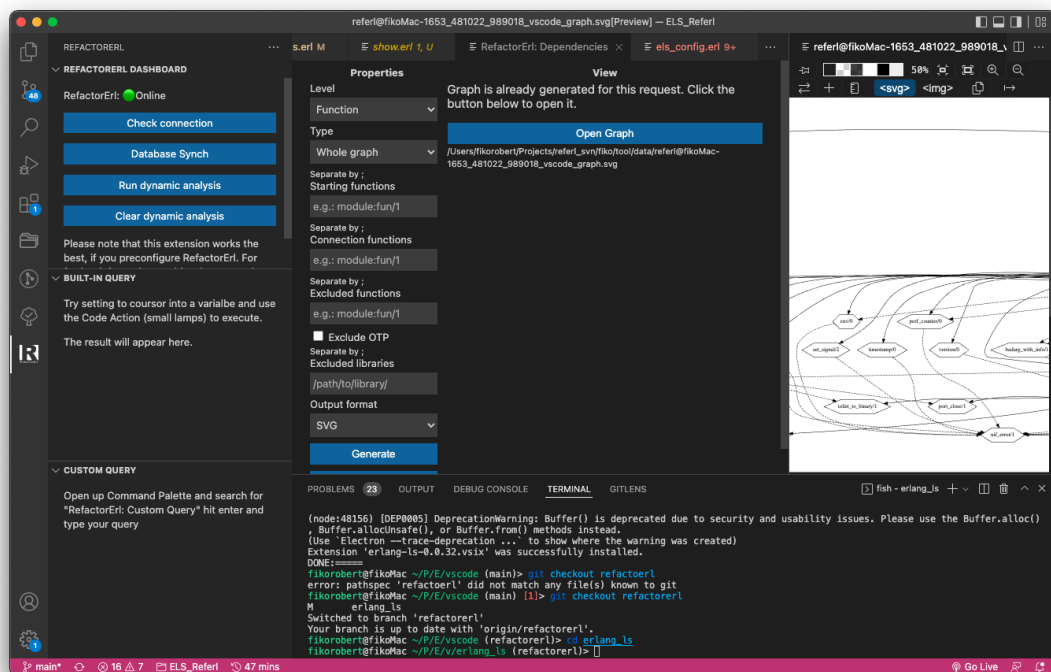
2.11. ábra. Függőségi gráf nézete a bővítményben

Ammennyiben a nézetből szeretnénk generálni a gráfot, úgy az alábbi lehetőségek állnak rendelkezésre az oldalsó részen: (bővebben a Wiki oldalon olvashatunk a szűrési opciókról (Filtering options) [18])

- **Level:** típusa, lehet *Function* (függvény) vagy *Module* (modul). Azt tudjuk meghatározni, hogy milyen szinten legyen a gráf generálva
- **Type:** típusa, lehet *Whole graph* (teljes gráf) vagy *Cyclic sub-graph* (ciklikus részgráf).
- **Starting functions/modules:** megadhatjuk, hogy mely függvények vagy modulok legyenek a kezdő pontok. Ha többet szeretnénk megadni, akkor pontos vesszővel válasszuk el.
- **Connection functions/modules:** megadhatjuk, hogy mely függvények vagy modulok legyenek a kapcsolódási pontok. Ha többet szeretnénk megadni, akkor pontos vesszővel válasszuk el.

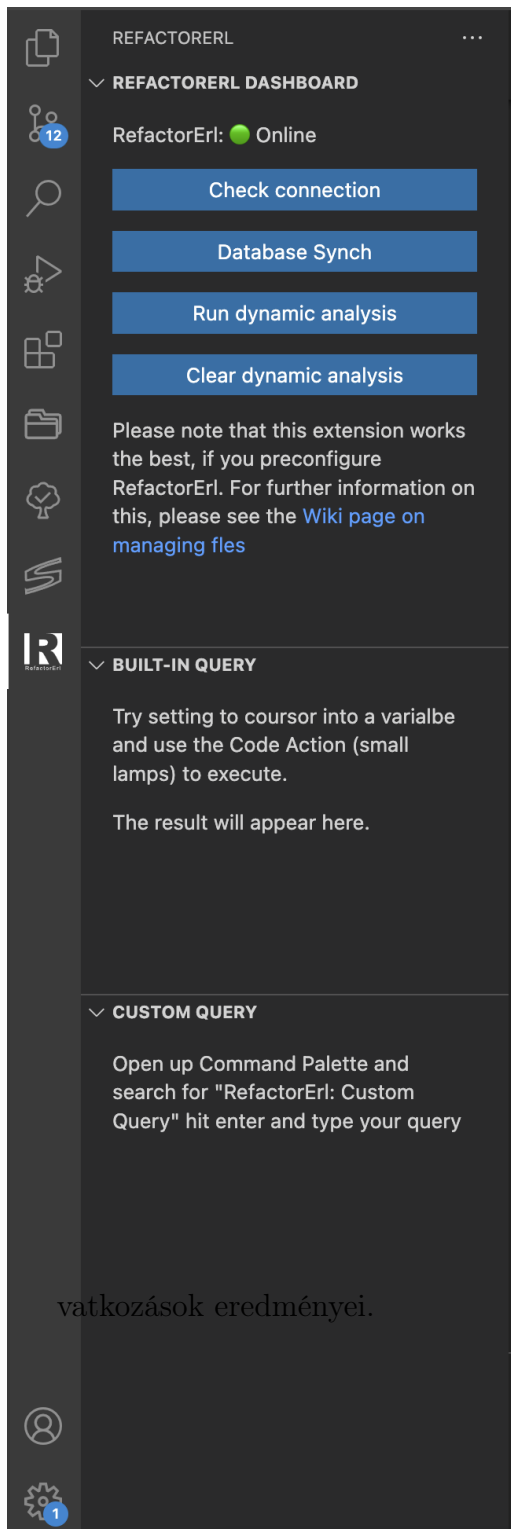
- **Excluded functions/modules:** megadhatjuk, hogy mely függvények vagy modulok legyenek kizárva. Ha többet szeretnénk megadni, akkor pontos vesszővel választjuk el.
- **Excluded OTP:** Amennyiben ezt bejelöljük, úgy az OTP függvényei, moduljai ki lesznek zárva a gráfból.
- **Excluded libraries:** megadhatjuk, hogy mely könyvtárak legyenek kizárva. Ha többet szeretnénk megadni, akkor pontos vesszővel választjuk el.
- **Output format:** kimeneti formátum, lehet *Plain textual* (szöveges reprezentáció) vagy *SVG*

Végül rendelkezésre áll egy *Generate* és egy *Clear* feliratú gomb. A feladatuk rendere a gráf generálása, létrehozása illetve a generált gráf törlése. Szöveges reprezentáció esetén az eredmény jobb oldalt a nagyobb munkaterületen jelenik meg. SVG opció esetén egy új fülön fog megjelenni, s a munkaterületen egy hivatkozás, gomb fog megjelenni. (lásd: 2.12 ábra)



2.12. ábra. Függőségi gráf SVG opcióval

#### 2.4.4. Az oldalsáv funkcionálisai



2.13. ábra. Oldalsáv funkcionálisai

Az 2.13 ábra alapján tekintsük át az oldalsávon elérhető funkcionálisokat.

*Megjegyzés: A 2.13 ábrán látott sorrendtől a felhasználó sorrendje eltérhet, hiszen ezen panelek szabadon átrendezhetőek. A bemutatásnál a 2.13 ábrán látható sorrendet követjük.*

Fő panelek áttekintése:

- **RefactorErl Dashboard**<sup>6</sup> Itt láthatjuk a RefactorErl node állapotát, illetve néhány vezérlő gomb segítségével ellenőrizhetjük a kapcsolatot, szinkronizálhatjuk az adatbázist, illetve lefuttathatjuk a dinamikus elemzést vagy éppen törölhetjük annak eredményét.
- **Built-In Query** Ezen panelen fognak megjelenni a beépített lekérdezések eredményei, mint például: *Variable Reach Variable Origin*, vagyis változó lehetséges értékei és dinamikus függvény hivatkozások eredményei.
- **Custom Query** Ebben a nézetben jelennek meg a *Command Palette*-n beírt egyedi lekérdezések eredményei.

<sup>6</sup>Vezérlőpult

#### 2.4.4.1. RefactorErl Dashboard

##### Kapcsolat állapota

Az első információ amit a vezérlőpult jelez nekünk, a kapcsolat állapota. Ennek két állapota lehet:

- Zöld jelzővel, *Online*, vagyis elérhető
- Piros jelzővel, *Not reachable*, vagyis nem elérhető

Az elérhető esetben a WebSocket alapú kapcsolat teljes mértékben létrejött és a bővítmény használható, az utóbbi esetben sajnos ez nem mondható el. Azt fontos kiemelni, hogy ha kiegészítő számára nem elérhető a RefactorErl node, az nem azt jelenti, hogy a teljes eszköz nem fut vagy nem működik. Erről további információt a 2.5.1 részben olvashatunk.

##### Kapcsolat ellenőrzése

A *Check connection* feliratú gomb megnyomásával tudjuk ellenőrizni a kapcsolat állapotát. Az alábbi eredményekkel zárulhat ez a művelet:

- **Sikeres kapcsolat.** Ebben az esetben a *Connected to RefactorErl via WS* üzenet jelenik meg a jobb alsó sarkban. Ez esetben a kapcsolat az eszköz és a bővítmény között WebSocketen keresztül ki van építve.
- **Nem lehet csatlakozni.** Ebben az esetben a *Cannot connect to RefactorErl via WS* üzenetet kapjuk jobb alul. Ebben az esetben a kapcsolat kiépítése közben hiba lépett fel.

**Adatbázis szinkronizálása** A *Database Synch* feliratú gombra kattintva a már betöltött fájlok frissítését tudjuk kérvényezni, amennyiben azokban történt változás. A folyamat állapotáról a jobb alsó sarokban a rendszer tájékoztat.

**Dinamikus függvényhívás elemzés elvégzése** A *Run dynamic analysis* címkéjű gomb segítségével a betöltött állományokon a dinamikus elemzést végezhetünk el. A folyamat állapotáról a jobb alsó sarokban a rendszer tájékoztat.

**Dinamikus függvényhívás elemzés eredményének törlése** A *Clear dynamic analysis* gomb megnyomásával a már elvégzett dinamikus elemzést törölhetjük. Ez



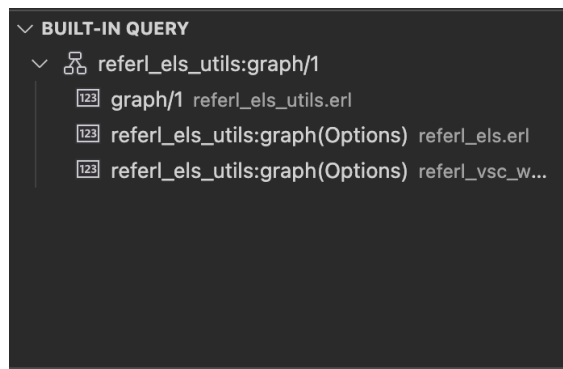
akkor lehet fontos, amikor az adatbázisban változás lépett fel. A folyamat állapotáról a jobb alsó sarokban a rendszer tájékoztat.

#### 2.4.4.2. Built-In Query megjelenítője

A második panelen a beépített lekérdezések megjelenítését találhatjuk. Amennyiben az Erlang LS-ből kódakciók segítségével indítjuk a lekérdezést, abban az esetben ebben a középső<sup>7</sup> panelben jelenik meg az eredmény, egy *fa szerkezetű listában*. Ha a listában egy értékre kattintunk, akkor bővítmény az adott eredményre visz, amennyiben van pozíció adat hozzá a RefactorErl adatbázisban.

Itt jelennek meg az alábbi beépített lekérdezések

- Variable Origin
- Variable Reach
- Dinamikus függvény hívások



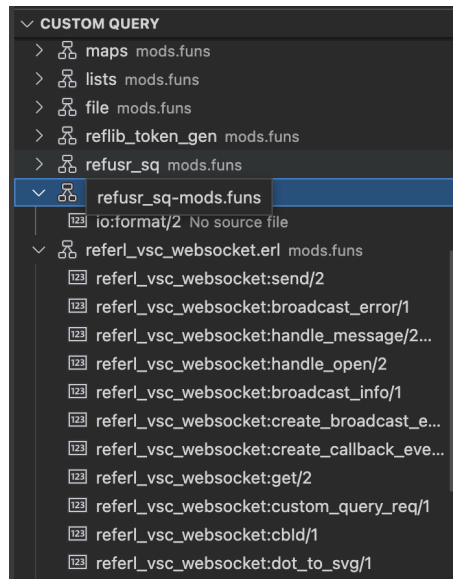
2.14. ábra. Dinamikus függvény hívások a Built-In Query megjelenítőben

#### 2.4.4.3. Custom Query megjelenítője

Az egyedi lekérdezések megjelenítője nagyon hasonló a beépítettéhez. Itt is egy *fa szerkezetű listában* jelennek meg az adatok, és kattintásra ugyaúgy az eredményhez vihetnek. (lásd: 2.15 ábra)

---

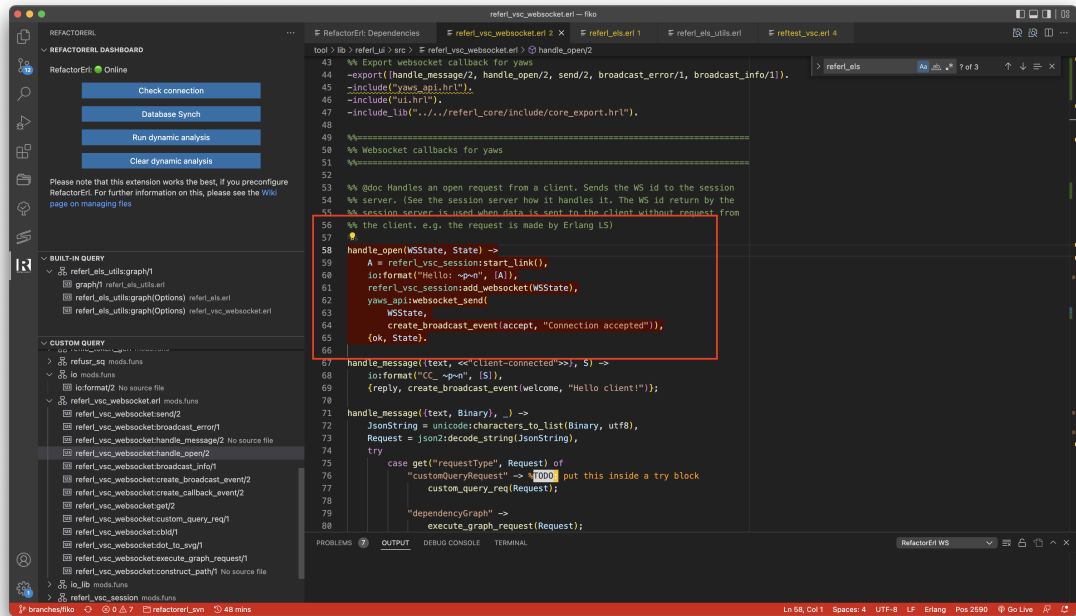
<sup>7</sup>A panelek tetszőlegesen átrendezhetőek, de az ábrán és alapértelmezetten középső



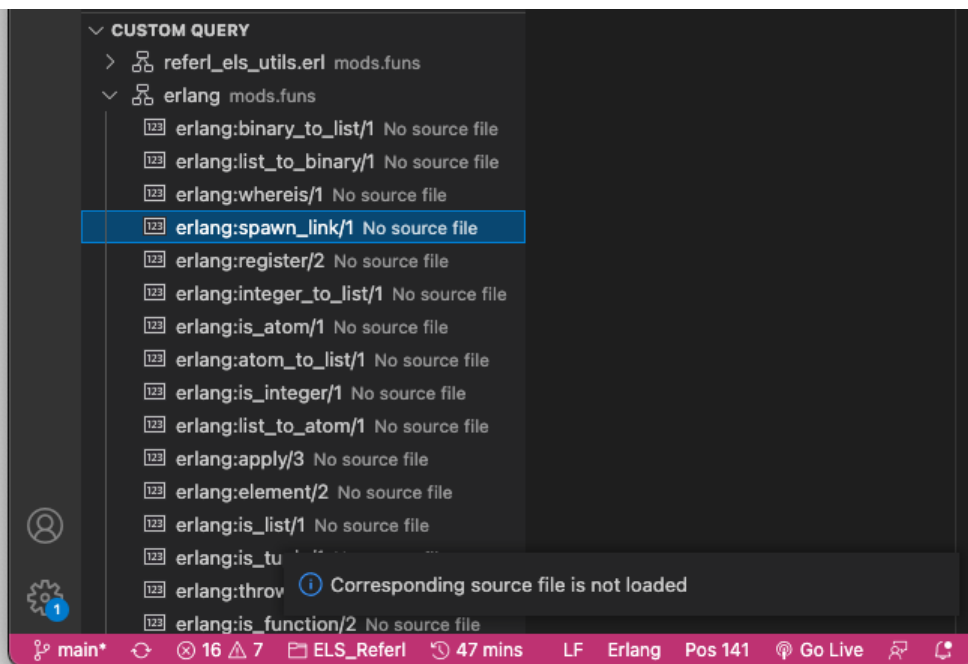
2.15. ábra. Egyedi `mods.funs` lekérdezés a Custom Query megjelenítőben

Egyedi lekérdezések esetében gyakrabban belefuthatunk a pozíció nélküli eredménybe. (lásd: 2.17 ábra) Az ilyen eredmény mellett látjuk a *No source file* alcímet, ami ezt jelzi nekünk. Erre való kattintás esetén hiba üzenet jelzi, hogy nem tudunk semmilyen pozícióra menni. Ez olyankor fordulhat elő, amikor meghívunk egy olyan függvényt, ami ugyan egy másik modulban jelene van, de annak a modulnak a forrása nincsen betöltve a RefactorErl-be, így nem tudunk hozzá lokációt adni.

Ammenyiben a pozíció adat rendelkezésre áll, úgy a megfelelő helyre visz a bővítmény. (lásd: 2.16 ábra)



2.16. ábra. Pozícióval rendelkező eredmény kódkiemelése a szerkesztőben



2.17. ábra. Pozíció nélküli eredmény megjelenítése

## 2.5. Probléma elhárítás

### 2.5.1. Ha a node nem elérhető a bővítmény szerint

Ez esetben győződjünk meg arról, hogy az Erlang node, amin a RefactorErl fut, az elérhető-e. Ehhez használhatjuk a `epmd -names` parancsot, ami listázza a futó Erlang node-okat. Valami hasonló kimenetet kell várnunk, mint a 2.5.1 forrásban, amennyiben fut a RefactorErl. Itt a node nevét kell látnunk.

```
1 fikorobert@fikoMac ~> epmd -names
2 epmd: up and running on port 4369 with data:
3 name erlang_ls_fiko_121933337 at port 55854
4 name referl at port 55652
5 ...
```

2.5. forráskód. Erlang Port Mapper Daemon (epmd) példa kimenete

Továbbá még használható a `net_adm:ping/1`<sup>8</sup> függvénye is, ahol a paraméterül adjuk meg egy nevesített Erlang node shelljében, az adott node nevét és `pong`-al tér vissza, akkor rendben van, ha `pang`-al akkor nincs rendben.

Továbbá győződjünk meg arról, hogy a VSC applikáció fut-e. Az elindításhoz használjuk a `referl_vsc:start()`, leállításhoz, pedig a `referl_vsc:stop()` hívásokat. Ezek után a *Try again* gombbal próbáljunk újra csatlakozni.

---

<sup>8</sup>[https://www.erlang.org/doc/man/net\\_adm.html#ping-1](https://www.erlang.org/doc/man/net_adm.html#ping-1)

## 3. fejezet

# Fejlesztői dokumentáció

### 3.1. Megoldandó feladat

### 3.2. Használt eszközök és környezetek

#### 3.2.1. Erlang nyelv

Az 1980-as évek közepén az Ericsson Számítógép Laboratóriuma azt a feladatot kapta, hogy alkossa meg azt a platformot, ami képes lesz a következő generációs telekommunikációs eszközök kezelésére. Joe Armstrong, Robert Virding, és Mike Williams Bjarne Däcker felügyelete mellett végül megalkották az Erlang programozási nyelvet. Ugyan sok már meglévő nyelvet megvizsgáltak, és volt köztük néhány igen ígéretes jelölt, de egyik sem tartalmazta az összes olyan funkciót, ami elvárt volt. A nyelvre hatással volt az ML és Miranda funkcionális nyelvek, valamint olyan konkurens nyelvek, mint az ADA, Chill vagy a Module, illetve a Prolog logikai programozási nyelv. Az első Prolog alapú Erlang virtuális gép prototípus négy év alatt készült el. Végül az Ericsson 1998 decemberében a nyelvet nyílt forrásúvá tette, így bárki számára elérhető lett [19].

Az Erlang nyelv a nagy hibatűró képességének, magas szintű konkurencia támogatásának köszönhetően kedvelt választás elosztott rendszerek fejlesztésében.

#### 3.2.2. TypeScript

A TypeScript egy multi-paradigma programozási nyelv, ami lényegében a JavaScript nyelvet egészíti ki olyan szintaxis elemekkel, amik segítségével statikus

típusozást tudunk elvégezni. TypeScriptben van lehetőségünk szerver- és kliensoldalon is programozni, a különböző keretrendszereknek köszönhetően. A TypeScript kódot futtatás előtt le kell fordítanunk. Itt a fordító elvégzi a szükséges szintaktikus és szemantikus elemzéseket, majd a típus definíciók és megszorítások eltávolítása után előáll a JavaScript forrás, amit már a megszokott JavaScript környezetekben futtathatunk [20] [21].

Így egy fenntarthatóbb, könnyebben kezelhető kódbázist kapunk, illetve a hibákat is előbb kitudjuk szűrni, ha megfelelő típusokat vezetünk be és használunk.

### 3.2.3. Visual Studio Code API

A Visual Studio Code egy nyílt forráskódú editor (szekresztő), amit a Microsoft fejleszt és tart karban. A teljes szoftvert olyan elsősorban weben ismert nyelvekkel írták, mint TypeScript, JavaScript, HTML és CSS, ezzel is segítve azt, hogy a legtöbb platformon elérhető legyen [22].

A fentiekén kívül a szerkesztő program rendkívül gazdag fejlesztői dokumentációval és nagyon sok oldalú API-val, vagyis *alkalmazásprogramozási felülettel* (*Application Programming Interface*) rendelkezik, ami a bővítmények fejlesztését lehetővé teszi. Az editor szinte minden funkcióját tudjunk vezérleni az API-ján keresztül, és ezt a fejlesztést is ez tette lehetővé.

### 3.2.4. WebSocket

A WebSocket egy számítógépes kommunikációs protokoll, ami teljes duplexitást, azaz kétirányú kommunikációt biztosít egyetlen TCP kapcsolat felett. A technológiát 2011-ben szabványosították és azóta a *Web Hypertext Application Technology Working Group* karbantartásában van, ami egy konzorcium a legnagyobb böngésző gyártókból (*Microsoft, Google, Apple és Mozilla*). [23]

A protokoll segítségével a kliens és a szerver között mindkét irányban küldhetünk adatot, így nem szükséges a hagyományos kérés-válasz (*request-response*) modellhez alkalmazkodni. Ilyen WebSocket alapú kommunikáció van a Visualiser bővítmény és a RefactorErl között.

### 3.2.5. Remote Procedure Calls

A Remote Procedure Call-nak (RPC), vagyis a távoli eljáráshívásnak a lényege az, hogy egy alkalmazás számára meghívjon egy függvényt, ami egy távoli számítógépen van implementálva és végre hajtva, úgy mintha lokálisan, helyileg elérhető lenne. A fejlesztőnek megfelelően definiálnia kell a modult és a függvényt, amit végre kíván hajtani a távoli rendszeren, majd a távoli számítógép egy RPC alrendszere ezt a hívást futtatható kóddá alakítja és végrehajta a távoli hívást [24].

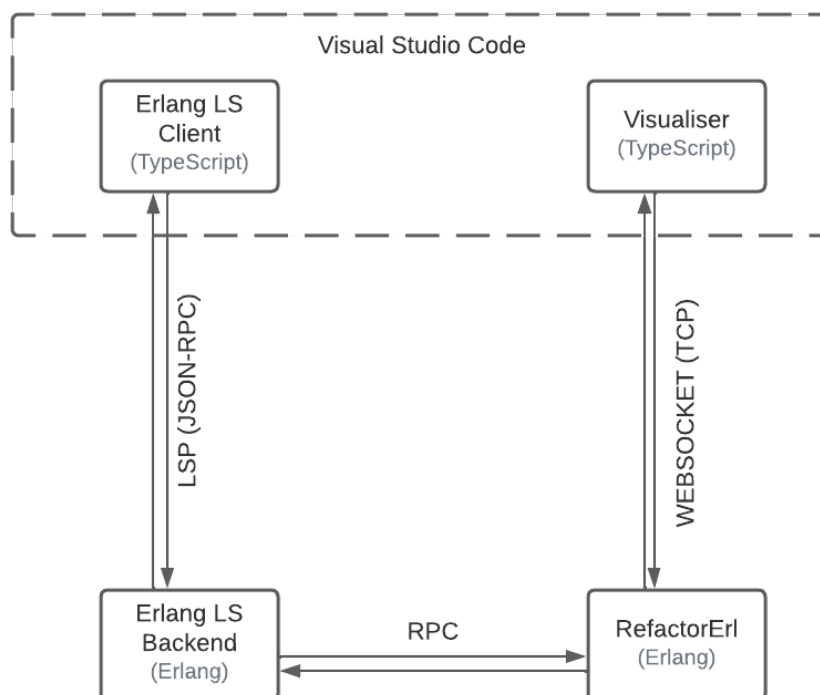
A távoli eljáráshívás igen fontos szerepet játszik az elosztott rendszerekben. Az Erlangban egy `rpc` modul segíti a fejlesztők munkáját, aminek a megfelelő felparaméterezésével elvégezhetjük a távoli hívást. Ilyen kapcsolatot alakítottam ki a `RefactorErl` és az Erlang LS node-jai között.

## 3.3. Komponensek

Az elkészült szoftver komponenseit négy fő részre lehet bontani: Erlang LS-beli illesztő modul, `RefactorErl`-beli illesztő modul, `RefactorErl` WebSocket komponense és a `Visualiser` kiegészítő. Ennek vizualizálásra tekintünk meg a 3.1 ábrát. Az ábrán fellelhető négy komponens nem teljese egészében feleltethető meg az előbb felsoroltakhoz.

A két `RefactorErl`-beli modult nem különböztettem meg az ábrán, hanem összevont *RefactorErl*-ként jellemeztem. Továbbá az Erlang LS kliensoldali forrásában nem történt változtatás a dolgozat keretein belül, azonban az architektúra megértése szempontjából fontosnak tartottam jelölni.

Az ábrán felsorolásra kerültek a használt kommunikációs protokollok is, amelyekről a 3.2 részben olvashatunk bővebben.



3.1. ábra. Komponensek viszonya egymáshoz

### 3.3.1. Erlang LS-beli illesztő modul

Az Erlang LS-ben egy meglehetősen miniális illesztő modul található, mely az üzleti logika jelentős részét a RefactorErlre bízva, hogy a projekt karbantartóinak minél kevesebb idegen kódot kelljen kezelniük. Az Erlang LS (vagy röviden: ELS) néhány helyen tartalmaz kiegészítéseket, most ezeket fogjuk sorra venni.

Az Erlang LS egy aktív projekt, akár havonta többször is jelenhet meg belőle újabb kiadás, ezért a fejlesztés során a forrást a 21.2-es verziónál rögzítettem, és azt terjesztettem ki a szükséges illesztő kódokkal

A kommunikáció alapját a `els_refactorerl_diagnostics` modul adja, ami nagyban függ a `els_refactorerl_utils` modultól. Vizsgáljuk meg közelebbről ezen modulokat.

#### 3.3.1.1. `els_refactorerl_diagnostics` modul

**Erlang Behavior:** Az Erlangban úgynevezett *behaviour* segítségével tudunk általános kód részeket kiemelni. Amennyiben több modul is többnyire azonos funkcionalitásokat lát el úgy a generikus, közös részeket *behaviour*-ba érdemes kiemelni, a specifikus részt pedig a *callback* modulban (*behaviour* megvalósító, kiterjesztő modul) implementálni.



**Callback függvény:** Az Erlang callback függvénynek nevezünk egy olyan függvényt, aminek az implementálását a *behavior* megköveteli.

Az `els_refactorerl_diagnostics` modul az Erlang LS `els_diagnostics` behavior-jét egészíti ki.

A modul az alábbi típusokat definiálja:

```
1 -type refactorerl_diagnostic_alias() :: atom().  
2 -type refactorerl_diagnostic_result() :: {range(), string()}.
```

### 3.1. forráskód. Típusdefiníciók az `els_refactorerl_diagnostics` modulban

- `refactorerl_diagnostic_alias`: szinoníma az `atom`-ra
- `refactorerl_diagnostic_result`: egy pár, aminek első tagja egy `range`, ami egy Erlang LS-beli típus, a második tagja pedig egy sztring (szöveges karakterlánc)

A modul az alábbi függvényeket definiálja:

- `-spec is_default()-> boolean()`.
  - **Bemenet:** -
  - **Kimenet:** logikai típus, hamis
  - **Leírás:** *callback függvény*, azt mondja meg, hogy alapértelmezett konfigurációval a diagnosztika fusson-e. Esetünk ez hamis értéket fog visszatéríteni.
- `-spec run(uri())-> [els_diagnostics:diagnostic()]`.
  - **Bemenet:** Uniform Resource Identifier, avagy univerzális erőforrás azonosító, röviden URI. Esetünkben ez egy elérési útvonal lesz, ami az adott elemzendő fájlra mutat.
  - **Kimenet:** ELS diagnosztikákat tartalmazó lista
  - **Leírás:** *callback függvény*, főbelépési pont a diagnosztikák tekintetében. A diagnosztikák RefactorErl elemzéseiből származik.
- `-spec source()-> binary()`.
  - **Bemenet:** -
  - **Kimenet:** A diagnosztika forrása: *RefactorErl*

- **Leírás:** *callback függvény*, minden diagnosztikának kell lennie egy forrássának, ami a diagnosztika mellett megjelenik.<sup>1</sup> Ez a függvény áthív a `els_refactorerl_utils` modulba és onnan tudja meg ezt a konstans információt.<sup>2</sup>
- `-spec enabled_diagnostics()-> [refactorerl_diagnostic_alias()]`.
  - **Bemenet:** -
  - **Kimenet:** Az engedélyezett diagnosztika azonosítók listája
  - **Leírás:** A konfigurációs fájl alapján és az `els_config` modul segítségével visszatér az engedélyezett
- `-spec make_diagnostics([refactorerl_diagnostic_result()])-> any()`.
  - **Bemenet:** `refactorerl_diagnostic_result`-ből álló lista
  - **Kimenet:** ELS-beli `diagnostic` típusú lista
  - **Leírás:** konvertáló függvény amely átalakítja a `input` (bemenet) listát, úgy, hogy az ELS megtudja jeleníteni

### 3.3.1.2. `els_refactorerl_utils` modul

A diagnosztikai modul nagyban támaszkodik a segéd moduljára, a `els_refactorerl_utils`, most nézzük meg ezt a modult, hogy milyen függvényeket definiál:

- `-spec referl_node()-> {ok, atom()} | {error, ErrorReason}`.
  - **Bemenet:** -
  - **Kimenet:**
    - \* `{ok, RefactorErlNode}`: a második tagja a párnak tartalmazza azt a `node`-ot amin a `RefactorErl` fut. Ebben az esetben az RPC alapú kapcsolat a `RefactorErl` `node`-al létre jött, használható.

---

<sup>1</sup>Ez sok esetben szokott *Compiler* (azaz fordító) azonosítót tartalmazni, hiszen a fordítóból elég sok diagnosztika származhat. Esetünkben ez *RefactorErl*, hiszen a diagnosztikákat az eszköz állítja elő.

<sup>2</sup>A későbbi fenntarthatóság és továbbfejlesztés jegyében történt ez a kialakítás

- \* `{error, ErrorReason}`: amennyiben a RefactorErl node-al bármi gond van ilyen formában kapjuk a node-ot. Az `ErrorReason` a hiba okát tartalmazza. Amennyiben `disconnected` az azt jelenti, hogy nem sikerült kapcsolódni a RefactorErl-hez, azonban amint lesz rá lehetőség a rendszer újra fogja próbálni. Amennyiben `disabled` a hiba oka, úgy a node-al a kommunikáció közben valami hiba történt, ami hatására le lett tiltva. Ezt a letiltást csak az Erlang LS újraindításával lehet feloldani. Illetve lehet még `other` amennyiben valami olyamsi hiba történt, amire nem készültünk fel korábban.
- **Leírás:** Amikor megnyitunk egy újabb fájlt vagy módosítunk egyet a lemezen, akkor az ELS lefuttatja (vagy újrafuttatja) a diagnosztikákat, s ekkor intézi az RPC hívást a RefactorErl node-ra. Ehhez ezt a függvényt használja. Amennyiben a kapcsolat még nem jött létre (és nem is lett rossznak ítélve), úgy újra próbálkozik annak létrehozásával.
- `-spec add(uri())-> error | ok.:` Amennyiben az `ok` atommal tér vissza, úgy sikeres volt, amennyiben az `error` atommal, úgy sikertelen
  - **Bemenet:** globális útvonal
  - **Kimenet:** `error` atom amennyiben a művelet nem volt sikeres, és `ok` atom amennyiben sikeres volt.
  - **Leírás:** hozzá adja a paraméterében kapott globális útvonalon elérhető fájlt a RefactorErl adatbázisához, RPC hívással
- `-spec run_diagnostics(list(), atom())-> list().:`
  - **Bemenet:** első paraméterében adjuk meg azon diagnosztika azonosítók-nak listáját amelyeket szeretnénk futtatni, a második paraméterben, pedig a modult, amire az eredmény halmazt szűkíteni kell
  - **Kimenet:** még nyers formátumú diagnosztika
  - **Leírás:** RPC hívással lefuttatja a kért diagnosztikákat a kért modulon a RefactorErl node-on
- `-spec notification(string(), number())-> atom().`

- **Bemenet:** Az egy paraméteres verzió esetében információs üzenetet küldhetünk, s magát az üzenet paraméterül adjuk meg. A két paraméteres verzióban a második paraméterrel pontosítani tudjuk azt, hogy milyen típusú üzenetet szeretnénk. Itt használhatjuk 1,2,3 vagy 4 értékeket, vagy az erre a célra definiált makrókat:

- \* `?MESSAGE_TYPE_ERROR`: hiba típusú értesítés, értéke: 1
- \* `?MESSAGE_TYPE_WARNING`: figyelmeztetés típusú értesítés, értéke: 2
- \* `?MESSAGE_TYPE_INFO`: információ típusú értesítés, értéke: 3
- \* `?MESSAGE_TYPE_LOG`: napló típusú értesítés, értéke: 4

- **Kimenet:**

- **Leírás:** egy segéd függvény amivel LSP protokoll feletti értesítéseket tudunk megjeleníteni.

- `-spec is_refactorer1(atom()) -> boolean().`

- **Bemenet:** Az Erlang node amin feltételezzük, hogy olyan verziójú RefactorErl node fut, ami képes az ELS kommunikálni
- **Kimenet:** igaz érték, amennyiben valóban megfelelő verziójú RefactorErl node fut, különben hamis.
- **Leírás:** A függvény egy RPC hívást intéz a RefactorErl node-jára. Megpróbálja meghívni az ott definiált `refer1_els:ping/0` függvényt. Amennyiben ez `{refactorer1_els, pong}`-al tér vissza, úgy megfelelő verziójú RefactorErl-t futtat a node. Ha bármi mással, melynek oka lehet az, hogy nem megfelelő a verzió, vagy az is, hogy a node-on nem fut az eszköz.

- `-spec connect_node({validate | retry, atom()} -> {error, disconnected} | atom().}`

- **Bemenet:** Az pár első tagjában a stratégiát határozzuk meg: `validate` avagy `retry`, a második pedig maga a node nevét adjuk meg
- **Kimenet:** `{error, disconnected}`, ha nem jött létre kapcsolat, ha pedig igen, akkor `{ok, Node}`, ahol a Node az Erlang node, amit megadtunk.

- **Leírás:** A stratégiát illetően `validate` opció esetében a felhasználó tájékoztatva lesz a node állapotáról. Ha korábban már kapcsolat sikeresen felállt, akkor arról, ha pedig most ez nem sikerült akkor arról lesz tájékoztatva. `retry` esetén a felhasználó csak akkor lesz tájékoztatva, ha a
- `-spec source_name()-> binary().`
  - **Bemenet:** -
  - **Kimenet:** `<<"RefactorErl">>.`
  - **Leírás:** A diagnosztika modul hívja meg ezt a függvényt. A későbbi bővíthetőség, fenntarthatóság jegyében lett ez ide kiemelve.
- `-spec variable_orgin(uri(), {number(), number()}) -> error | ok.`
  - **Bemenet:** Első paraméter az útvonal binárosan, a második párban rendre sor és oszlop információ, az adott változóról.
  - **Kimenet:** `error`, amennyiben valami hiba történt, `ok` amennyiben minden rendben.
  - **Leírás:** A `RefactorErl` node felé intézett RPC híváson keresztül kiszámolja egy változó `Origin` értékét.
- `-spec variable_reach(uri(), {number(), number()}) -> error | ok.`
  - **Bemenet:** Első paraméter az útvonal binárosan, a második párban rendre sor és oszlop információ, az adott változóról.
  - **Kimenet:** `error`, amennyiben valami hiba történt, `ok` amennyiben minden rendben.
  - **Leírás:** A `RefactorErl` node felé intézett RPC híváson keresztül kiszámolja egy változó `Reach` értékét, amit Web Socketen keresztül megküldd, a Visualiser bővítmény felé.
- `-spec dependency_graph(string(), func | mod)-> error | ok.`
  - **Bemenet:** Az első paraméter a függvény vagy modul nevét tartalmazza rendre `<modul_neve>:< fggvny_neve >/< arits >`, illetve `<module_neve>` formában.