

The SPOCP Plug-In Developers Manual

by Roland Hedberg (Umeå University), Torbjörn Wiberg (Umeå University)

Table of contents

1 Introduction.....	2
2 Boundary Condition Definitions in Rules.....	2
2.1 Unitsel.....	2
3 Function Prototype.....	3
4 Module format.....	4
5 APIs.....	5

1. Introduction

Much of the power in SPOCP sits in the backends. It is here that the boundary conditions are evaluated and all the connections to external information resources are handled.

If the doxygen documentation is built, then you can find more detailed information on functions and structs there.

2. Boundary Condition Definitions in Rules

The boundary condition definition follows a specific format:

```
bcond      = pname ":" *unitsel ":" pspec
```

Where *pname* is the name of the plug-in, *unitsel* is the specification of which pieces of information should be gotten from the query S-expression. And *pspec* is the information given using a format that is specific to this plug-in.

2.1. Unitsel

This part of the boundary condition is for picking information from the query. Since the query in a parsed format is presented to the function, it can implement whatever extraction function it needs. But since we, rightly or wrongly, assumed that most implementers wants to concentrate on the function which the backend is going to provide, we thought it would be a good thing to provide this functionality.

S-expressions can be regarded as representing a tree structure, so the use of [XPATH](#) seems reasonable. But we have limited our usage to a small subset of what XPATH provides; you can use absolute location path (/lpw/subj/id) or descendants (//id) to get the sublist/-s in question, and then a selection to get at the elements of that sublist.

Thus, if we start with the query S-expression:

```
(lpw (service Tentamen)(method skapa)(departmentNumber 64545)(uid leifj))
```

Then "{//uid[1]}" will pick the first element, after the tag, in the sublist that has "uid" as a tag, the tag being the first element (index 0). In this case that is the string "leifj".

If the S-expression is:

```
(spocp (resource jaccount (page 316))(action read)(subject (uid ola) (role \
```

```
manager)))
```

"{/spocp/subject/role/*}" will pick the sequence of elements from the specific sublist with the tag "role" that starts with the second element. In this case that is a single element sequence that only contains the string "manager". The element "/*" is equivalent to "[1-]".

As selections, you can use "/*", or one of "[n]", "[n-]", "[n-m]", "[-m]" where n and m are numbers and n < m. You can also, instead of m, use the string "last" to denote the last element in a sublist.

So, if the s-expression is:

```
(http (resource gif sysinfo_en.png -)(action 0 0)(subject(ip
127.0.0.1)(host
localhost)))
```

Then:

- "//resource/*" => ("gif" "sysinfo_en.png" "-")
- "//resource[1]" => "gif"
- "//resource[last]" => "-"

3. Function Prototype

All the plug-in functions MUST follow the same format:

```
spocp_result_t befunc( cmd_param_t *, octet_t *);
```

Where the structure `cmd_param_t` has the elements:

element_t *q

Parsed version of the query S-expression

element_t *r

Parsed version of the S-expression of the matching rule

element_t *x

The result of applying the variable extraction expressions (*unitsel* mentioned above) on the query S-expression.

octet_t *arg

The plug-in specific part (*pspec* above) of a boundary definition

pdyn_t *pd

A structure that holds information needed for caching and backend connection pool handling

void *con

A pointer to something the plug-in has created to store plug-in specific information

And the `octet_t` struct is where the plug-in should place dynamically produced blobs.

4. Module format

We are using a similar format to what Apache is using, in that the plug-in source file must contain information about the plug-in functions. This includes both of the two basic (the test and the initialization) functions, as well as the ones that should be used to handle plug-in configuration directives for this plug-in, and a function that can remove whatever the plug-in has stored on its own accord.

Snippet from the `foobar` plug-ins source file:

```
typedef struct foobar_t {
    int foo_defined;
    ...
} foobar_t ;

spocp_result_t foobar_test( cmd_param_t *cpp, octet_t *blob)
{
    ....
}

spocp_result_t foofunc( void **conf, void *cmd_data, int argc, char **argv)
{
    foobar_t *ft = (foobar_t *) *conf;

    if (ft == NULL)
        *conf = ft = (foobar_t *) calloc (1, sizeof( foobar_t ));

    ft->foo_defined = 1;
    ....
    return SPOCP_SUCCESS;
}

spocp_result_t free_args( void **conf, void *cmd_data, int argc, char
**arg\
v)
{
    foobar_t *ft = (foobar_t *) *conf;

    if (ft) {
        free( ft );
        *conf = 0;
    }
}
```

```
    }  
    return SPOCP_SUCCESS;  
}  
  
conf_com_t conf = {  
    { "foo", foofunc, NULL, "This is the function handling the foo directive"  
    \,  
    },  
    NULL,  
};  
  
plugin_t      foobar_module = {  
    SPOCP20_plugin_STUFF,  
    foobar_test,  
    foobar_init,  
    conffunc,  
    free_args  
};
```

5. APIs

Almost all of the backends need the same support from the server, and also has similar needs when dealing with connections to external resources and caching previous results from evaluations. This explains why we have created a set of APIs for use by the backends.

Check the doxygen documentation for specifics on the APIs.