8 Inline Action to External Contract

This section of the guide provides instructions for sending actions to an external contract. The examples of the execution of instructions are shown in the contract, where the performed actions are kept.

8.1 Addressbook counter address contract

Log into `CONTRACTS_DIR` and create a directory there with the name `abcounter`, as well as the file `abcounter.cpp`.

```
cd CONTRACTS_DIR
mkdir abcounter
cd abcounter
touch abcounter.cpp
```

Open the `abcounter.cpp` file and write the following program text into it:

```
#include <eosiolib/eosio.hpp>

using namespace eosio;

class [[eosio::contract]] abcounter : public eosio::contract {
  private:
    struct [[eosio::table]] counter {
      name key;
      uint64_t emplaced;
      uint64_t modified;
      uint64_t erased;
      uint64_t primary_key() const {
      return key.value;
    }
  };

    using count_index = eosio::multi_index<"counter"_n, counter>;
  public:
```

```
    using contract::contract;

    abcounter(name receiver, name code,  datastream<const char*> ds):
contract(receiver, code, ds) {}

    [[eosio::action]]
    void count(name user, std::string type) {
      require_auth( name("addressbook"));
      count_index counts(name(_code), _code.value);
      auto iterator = counts.find(user.value);

      if (iterator == counts.end()) {
        counts.emplace("addressbook"_n, [&]( auto& row ) {
          row.key = user;
          row.emplaced = (type == "emplace") ? 1 : 0;
          row.modified = (type == "modify") ? 1 : 0;
          row.erased = (type == "erase") ? 1 : 0;
        });
      }
      else {
        counts.modify(iterator, "addressbook"_n, [&]( auto& row ) {
          if(type == "emplace") { row.emplaced += 1; }
          if(type == "modify") { row.modified += 1; }
          if(type == "erase") { row.erased += 1; }
        });
      }
    }
};

EOSIO_DISPATCH( abcounter, (count));
```

The peculiarity of this text is that it has a restriction on calls to an action for the contract account. For restriction, use `require_auth` for the `addressbook` contract. Only the `addressbook` contract account is authorized to perform `require_auth`. The `count` action is not sent by the user, but by the `addressbook` contract.

```
require_auth( name("addressbook"));
```

8.2 Create an account for abcounter contract

```
cleos create account cyber abcounter YOUR_PUBLIC_KEY
```

8.3 Compile and install the abcounter contract

```
eosio-cpp -o abcounter.wasm abcounter.cpp --abigen
cleos set contract abcounter CONTRACTS_DIR/abcounter
```

8.4 Modify the contract addressbook to send inline actions to a new abcounter contract

```
cd CONTRACTS_DIR/addressbook
```

Open the file `addressbook.cpp` and create in it another helper named `increment_counter` in the private part of the contract.

```
void increment_counter(name user, std::string type) {

  action counter = action(
    permission_level{get_self(),"active"_n},
    "abcounter"_n,
    "count"_n,
    std::make_tuple(user, type)
  );

  counter.send();
}
```

The action body contains:

- выдача разрешения уровня `active` .Для разрешения функция `get_self()` возвращает текущий контракт `addressbook` ;
- the `abcounter` account name of the contract;
- the `count` action to call;
- data, username and line type.

Add calls to «set», «modify» and «erase» helpers.

```
//Emplace
increment_counter(user, "emplace");
//Modify
increment_counter(user, "modify");
//Erase
increment_counter(user, "erase");
```

As a result, the `addressbook.cpp` file should look like this:

```
#include <eosiolib/eosio.hpp>
#include <eosiolib/print.hpp>

using namespace eosio;

class [[eosio::contract]] addressbook : public eosio::contract {

  private:
    struct [[eosio::table]] person {
      name key;
      std::string first_name;
      std::string last_name;
      uint64_t age;
      std::string street;
      std::string city;
      std::string state;

      uint64_t primary_key() const { return key.value; }

      uint64_t get_secondary_1() const { return age;}

    };

    void send_summary(name user, std::string message) {
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        "notify"_n,
        std::make_tuple(user, name{user}.to_string() + message)
      ).send();
    };

    void increment_counter(name user, std::string type) {

      action counter = action(
        permission_level{get_self(),"active"_n},
        "abcounter"_n,
        "count"_n,
        std::make_tuple(user, type)
      );
```

```
41
42      counter.send();
43    }
44
45    typedef eosio::multi_index<"person"_n, person,
46      indexed_by<"byage"_n, member<person, uint64_t, &person::get_secondary_1>
47    > address_index;
48    public:
49      using contract::contract;
50
51      addressbook(name receiver, name code,  datastream<const char*> ds):  con
52
53      [[eosio::action]]
54      void upsert(name user, std::string first_name, std::string last_name, u
55        require_auth(user);
56        address_index addresses(_code, _code.value);
57        auto iterator = addresses.find(user.value);
58        if( iterator == addresses.end() )
59        {
60          addresses.emplace(user, [&]( auto& row ) {
61           row.key = user;
62           row.first_name = first_name;
63           row.last_name = last_name;
64           row.age = age;
65           row.street = street;
66           row.city = city;
67           row.state = state;
68
69           send_summary(user, " successfully emplaced record to addressbook");
70           increment_counter(user, "emplace");
71          });
72        }
73        else {
74          std::string changes;
75          addresses.modify(iterator, user, [&]( auto& row ) {
76            row.key = user;
77            row.first_name = first_name;
78            row.last_name = last_name;
79            row.age = age;
80            row.street = street;
81            row.city = city;
82            row.state = state;
83
84            send_summary(user, " successfully modified record to addressbook");
85            increment_counter(user, "modify");
86        });
87
88      }
89    }
90
91    [[eosio::action]]
92      void erase(name user) {
93        require_auth(user);
94
95        address_index addresses(_code, _code.value);
```

```
96        auto iterator = addresses.find(user.value);
97        eosio_assert(iterator != addresses.end(), "Record does not exist");
98        addresses.erase(iterator);
99        send_summary(user, " successfully erased record from addressbook");
100       increment_counter(user, "erase");
101     }
102
103   [[eosio::action]]
104     void notify(name user, std::string msg) {
105       require_auth(get_self());
106       require_recipient(user);
107     }
108
109   };
110
111 EOSIO_DISPATCH( addressbook, (upsert)(notify)(erase));
```

8.5 Recompile and set the addressbook contract

Since the changes should not affect the ABI, you do not need to re-edit the ABI file (make sure that this file has not been updated).

```
1 eosio-cpp -o addressbook.wasm addressbook.cpp
2 cleos set contract addressbook CONTRACTS_DIR/addressbook
```

8.6 Perform testing of sending an action from an addressbook contract to an abcounter contract

Before testing, make sure that the `abcounter` and `addressbook` contracts are set.

8.6.1 Verify that a notification is sent to the `abcounter` contract, as well as a change in the entry in the `addressbook` contract table by executing:

```
  cleos push action addressbook upsert '["alice", "alice", "liddell", 19, "123
```

The action is considered successful if the output contains the following information:

```
executed transaction: ...
#   addressbook <= addressbook::upsert          {"user":"alice","first_name"
#   addressbook <= addressbook::notify          {"user":"alice","msg":"alic
#         alice <= addressbook::notify          {"user":"alice","msg":"alic
#     abcounter <= abcounter::count             {"user":"alice","type":"mod
```

8.6.2 Check the appearance of a row in the table for the `alice` user.

```
cleos get table abcounter abcounter counts --lower alice --limit 1
```

The action is considered successful if the result is the following:

```
{
  "rows": [{
      "key": "alice",
      "emplaced": 1,
      "modified": 0,
      "erased": 0
    }
  ],
  "more": false
}
```

8.6.3 Check that the `upsert` method modifies the record.

```
cleos push action addressbook upsert '["alice", "alice", "liddell", 21,"1 the
```

The actions are considered successfully completed if the resulting output contains the following information:

```
executed transaction: …
```

```
#   addressbook <= addressbook::upsert          {"user":"alice","first_name"
#   addressbook <= addressbook::notify          {"user":"alice","msg":"alice
>> Notified
#         alice <= addressbook::notify          {"user":"alice","msg":"alice
#     abcounter <= abcounter::count             {"user":"alice","type":"empt
warning: transaction executed locally, but may not be confirmed by the netwo
```

8.6.4 Verify the liquidation of the `alice` user entry from the table by running:

```
cleos push action addressbook erase '["alice"]' -p alice@active
```

The action is considered to be successfully completed if the resulting output contains the following information:

```
executed transaction: ...
#   addressbook <= addressbook::erase           {"user":"alice"}
>> Erased
#   addressbook <= addressbook::notify          {"user":"alice","msg":"alice
>> Notified
#         alice <= addressbook::notify          {"user":"alice","msg":"alice
#     abcounter <= abcounter::count             {"user":"alice","type":"eras
warning: transaction executed locally, but may not be confirmed by the netwo
Toaster:addressbook sandwich$
```

8.6.5 Test the ability to manipulate `abcounter` contract data by executing:

```
cleos push action abcounter count '["alice", "erase"]' -p alice@active
cleos get table abcounter abcounter counts --lower alice
```

The `abcounter` contract table should contain the following information:

```
{
  "rows": [{
      "key": "alice",
      "emplaced": 1,
```

```
      "modified": 1,
      "erased": 1
    }
  ],
  "more": false
}
```