5 Data Persistence

This section provides instructions for creating and setting up a table for user data, creating an instance of the table, adding new rows as well as instructions for adjusting and deleting them. In addition, there are instructions for setting up an ABI file and testing the installed contract.

To store user data, a contract can be created to be used as an address book. Despite the fact that this use of data is not very practical this operation illustrates a method of storing data in CyberWay without any distortion of business logic.

5.1 Create a new directory for the address book contract

```
cd CONTRACTS_DIR
mkdir addressbook
cd addressbook
```

5.2 Create file

```
touch addressbook.cpp
```

Fill the file with the following content:

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

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contrac
    public:

```

```
    private:

};
```

5.3 Create address book data structure

Before a table can be configured and created, it is necessary to create an address book data structure. Since this is an address book, the table must contain information about people, and the contract itself must store some important data for each record or person. A possible `person` structure is the following:

```
struct person {
    name key;
    std::string first_name;
    std::string last_name;
    std::string street;
    std::string city;
    std::string state
};
```

> The structure of the table cannot be changed as long as it is filled out with data. If necessary, change data structure of the table. For that you must first delete all its rows.

5.4 Configure table indexes

The table has to be configured after determining its data structure. The first step is to define `eosio::multi_index` constructor to use the table data structure. Add the following line to the created class:

```
typedef eosio::multi_index<"people"_n, person> address_index;
```

This definition of `multi_index` creates a table called `people`, which allows the following:

- to use the `_n` operator to determine the type of `eosio::name` and also to use this name for the table. This table will contain several descriptions of individual «person»s, so you can assign the name «people» to the table;
- to transfer the structure for an individual defined in the previous step;
- to declare the type of this particular table. This type will be used to create instances of this table;
- to configure and accommodate the table indexes, which are to be mentioned later.

At this stage the file should have the following form:

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

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contra

  public:

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

    typedef eosio::multi_index<"people"_n, person> address_index;
  public:

};
```

5.5 Create class constructor

When working with C++ classes, the first open method to be created is the constructor. The constructor determines the initial setting of the contract. The contracts allow you to expand the class. To do this, you need to initialize the parent class of the contract with the code name of the contract and the recipient.

```
  addressbook(name receiver, name code, datastream<const char*> ds):contract(re
```

Параметр:
`name code` — a contract account name.

5.6 Create table entries

5.6.1 The following conditions must be met:

- Any changes are added to the address book by the user himself only.
- For the ease of use, the contract should provide an ability to create or modify a row of table in one action.

5.6.2 Define the user action to add a record or modify a table. This action should take any values necessary to create or edit an entry in the table.

Format the method. For further convenience and simplicity of the interface, the same method will perform both the creation and the modification of lines. It is given the name «upsert» (as a combination of «update» and «insert»).

Only the user has the right to control his own entry in the table. To perform this, use the `require_auth` method provided by `eosio.cdt` . This method takes one argument — the name of the account performing the transaction. The `upsert` method is:

```
void upsert(
    name user,
    std::string first_name,
    std::string last_name,
    std::string street,
    std::string city,
    std::string state
) {
  require_auth( user );
}
```

5.6.3 Create a table. The `multi_index` table is configured and declared as `address_index`. Creating an instance of a table requires two arguments:

- contract account code;
- scope of the contract.
  You also need to set an iterator for a frequent use.
  The table will be the following:

```
void upsert(
    name user,
    std::string first_name,
    std::string last_name,
    std::string street,
    std::string city,
    std::string state) {
        require_auth( user );
        address_index addresses(_code, _code.value);
        auto iterator = addresses.find(user.value);
}
```

5.6.4 Write the logic for creating or modifying a table, the logic for determining the existence of a user. To do this, you can use the table's method called `find`, passing the user parameter. The `find` method returns a table iterator.

```
void upsert(
…
) {
    require_auth( user );
    address_index addresses(_code, _code.value);
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
      {
            //The user isn't in the table
     }
      else {
         //The user is in the table
      }
}
```

5.6.5 Create an entry in the table using the `emplace` method. This method takes two arguments: the «payer» of this record, which pays for the use of storage and the callback function.

The callback function for the `emplace` method should use lambda to create a link. Inside a body, assign values to the rows for the `upsert`.

```
void upsert(
…
) {
    require_auth( user );
    address_index addresses(_code, _code.value);
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
    {
        addresses.emplace(user, [&]( auto& row ) {
        row.key = user;
        row.first_name = first_name;
        row.last_name = last_name;
        row.street = street;
        row.city = city;
        row.state = state;});
    }
    else {
        //The user is in the table
    }
}
```

5.6.6 To modify an entry in a table, you can use the `modify` method with the following arguments:

- iterator — an iterator (previously defined);
- user — a user who is the payer;
- a function to handle the table modification.

```
...
if( iterator == addresses.end() )
{
...
}
else {
std::string changes;
addresses.modify(iterator, user, [&]( auto& row ) {
    row.key = user;
    row.first_name = first_name;
    row.last_name = last_name;
    row.street = street;
    row.city = city;
    row.state = state;
});
}
...
```

Upon completion of the instructions in this clause, the address book contract will have a functional action that will allow the user to create or modify a row in the table.

5.7 Deleting records from the table

Create a public method in your address book and make sure that it includes ABI-ads. Also create `require_auth` to check the user's right to perform the action. Only the owner of the record can change his account. In the address book, each account has only one entry.

To delete a record, you need to set an iterator using `find` and call the `erase` method

```
...
void erase(name user) {
    require_auth(user);
    address_index addresses(_code, _code.value);
    auto iterator = addresses.find(user.value);
    eosio_assert(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
}
...
```

Upon completion of the instructions in this clause, the address book contract will have a functional action that will allow the user to create, modify or delete any entry in the table. However, the contract is not yet ready for compilation.

5.8 Edit the ABI file

Before you compile the contract you need to edit the `addressbook.cpp` file.

5.8.1 At the end of the file, place the macro EOSIO_DISPATCH, specifying the name of the contract, as well as the «upsert» and «erase» actions.

This macro sends calls to specific methods in this contract. The macro ensures the compatibility of the cpp-file with the wasm EOSIO interpreter. Failure to include this declaration may lead to an error in the deployment of the contract.

```
EOSIO_DISPATCH( addressbook, (upsert)(erase) )
```

5.8.2 Declare actions in the ABI file.
Although eosio.cdt includes an ABI generator, you also need to manually modify the generated ABI file. At the top of the description of the functions `upsert` and `erase` add a declaration in the following form:

```
[[eosio::action]]
```

This declaration allows you to extract action arguments and to create the necessary ABI structure descriptions in the generated ABI file.

5.8.3 Declare tables in an ABI file.
Matching the previous paragraph instructions change the declaration for the table:

```
struct [[eosio::table]] person {
```

The final state of the contract with the address book before compilation is the following:

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

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contrac

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

```cpp
    std::string state;

    uint64_t primary_key() const {return key.value;}
  };
  typedef eosio::multi_index<"person"_n, person> address_index;

public:
  using contract::contract;

addressbook(name receiver, name code,  datastream<const char*> ds): contrac

  [[eosio::action]]
  void upsert(name user, std::string first_name, std::string last_name, std
        require_auth( user );
        address_index addresses(_self, _self.value);
        auto iterator = addresses.find(user.value);
        if( iterator == addresses.end() )
        {
            addresses.emplace(user, [&]( auto& row ) {
                row.key = user;
                row.first_name = first_name;
                row.last_name = last_name;
                row.street = street;
                row.city = city;
                row.state = state;
            });
        }
        else {
                std::string changes;
                addresses.modify(iterator, user, [&]( auto& row ) {
                row.key = user;
                row.first_name = first_name;
                row.last_name = last_name;
                row.street = street;
                row.city = city;
                row.state = state;
        });
      }
  }

  [[eosio::action]]
  void erase(name user) {
    require_auth(user);

    address_index addresses(_self, _self.value);

    auto iterator = addresses.find(user.value);
    eosio_assert(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
  }

};

EOSIO_DISPATCH( addressbook, (upsert)(erase))
```

5.9 Compile a contract

```
eosio-cpp -o addressbook.wasm addressbook.cpp --abigen
```

5.10 Describe the index of the primary key of the table in the ABI file

The generator creates a table without indices during the compilation of the ABI file:

```
...
"tables": [
{
    "name": "person",
    "type": "person",
    "index_type": "i64",
    "key_names": [],
    "key_types": []
}],
…
```

You must edit the table in the ABI file, keeping only the `name` and `type` unchanged and replacing the remaining fields with the following description of the primary key index:

```
…
  "indexes": [
  {
      "name": "primary",
      "unique": "true",
      "orders": [
      {"field": "key", "order": "asc"}
 ]}
…
```

5.11 Install a contract

5.11.1 Create an account for the contract by executing:

```
cleos create account olga addressbook <public key>
```

5.11.2 Set an addressbook contract by executing:

```
cleos set contract addressbook CONTRACTS_DIR/addressbook -p addressbook@activ
```

5.12 Test an established contract

5.12.1 Check an addition of a row to a table for a user named `alice`.

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

As an output the following information should appear:

```
executed transaction: ...
#   addressbook <= addressbook::upsert         {"user":"alice","first_name'
```

5.12.2 Verify that the `alice` user cannot create an entry for another user by running:

```
cleos push action addressbook upsert '["bob", "bob", "is a loser", "doesnt ex
```

The `require_auth` method in a contract should not allow another user to create (or modify) strings. As a result, you should receive an error message:

```
Error 3090004: Missing required authority
Ensure that you have the related authority inside your transaction!;
If you are currently using 'cleos push action' command, try to add the relev
```

5.12.3 Get a user record with the `alice` name.

```
cleos get table addressbook addressbook people --lower alice --limit 1
```

As an output the following information should appear:

```
{
  "rows": [{
      "key": "3773036822876127232",
      "first_name": "alice",
      "last_name": "liddell",
      "street": "123 drink me way",
      "city": "wonderland",
      "state": "amsterdam"
    }
  ],
  "more": false
}
```

5.12.4 Verify that the user with the name `alice` can delete the record from the table by executing::

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

As an output the following information should appear:

```
executed transaction: …
#   addressbook <= addressbook::erase           {"user":"alice"}
warning: transaction executed locally, but may not be confirmed by the netw
```

Check for no record by running:

```
cleos get table addressbook addressbook people --lower alice --limit 1
```

The following Information should appear:

```
{
  "rows": [],
  "more": false
}
```