Build Instructions:

Swift_CPP_SDK depends on pthread, JsonCpp and Poco C++ libraries. Pthread should be available on any POSIX compatible operating system. JsonCpp provides an amalgamated source and header file which can be easily compiled with any project. It has been included under /utils/jsoncpp folder. Therefore, you don't need to do anything and it will be compiled and linked automatically. Alternatively you can download, make, install and link to it in the Swift_CPP_SDK makefile.

Finally, the last missing piece is Poco C++ library. There is a copy of this library included under /utils/ folder. Moreover, it has been built and included under utils/poco folder for X86_64 linux. So if you use an X86_64 linux machine, you're good to go and don't need to do anything. Alternatively, you can download, make, install, and link to this library in the Swift_CPP_SDK makefile.

Swift_CPP_SDK uses some new features of C++0x; therefore, it needs to be compiled with a modern compiler. Swift_CPP_SDK has been compiled with both Clang 3.5 and GCC 4.8.
Having all of the prerequisites, you can start building Swift_CPP_SDK by issuing make command in the root directory of Swift_CPP_SDK. After build process completes, you can find a folder named build in the root directory. It contains json, poco, and libSwift. You should then copy these files to your project and point your compiler to include header files and link to the libraries.

Usage:

Swift is a distributed object storages service which is part of openstack project. This SDK still lacks many functionalities and is under active development. There are three important entities in this SDK corresponding to Swift architecture elements. These three elements represent Swift hierarchy. The first element which is the root level in Swift hierarchy is Account which contains authentication information and is a namespace for containers. The second entity is Container, which is a namespace for objects. Finally, there is Object entity which correspond to a single object or file in our storage.

In order to use Swift, you need to have an account first, then you can have different containers within that account; finally, you can have multiple objects within a container. In other words, an account corresponds to a partition or drive, a container corresponds to a folder and an object represents a file in terms of traditional file storage systems.

Swift provides a REST API to create, access, and modify object storage elements. This SDK has implemented all the necessary communications with the Swift server and you won't deal with them. In order to use Swift SDK you need to first create an account entity correspond to your account on the swift server. Swift provides different authentication mechanism including: BASIC, TEMPAUTH, and KEYSTONE. However, in this SDK only KEYSTONE authentication method has been implemented so far. To create an account we need to fill required information in an instance of AuthenticationInfo struct:

```
AuthenticationInfo info;
info.username = "username";
info.password = "****";
```

```
info.authUrl = "url to swift authentication server";//e.g: http://10.42.0.83:5000/v2.0/tokens
info.tenantName = "tenant name";
info.method = AuthenticationMethod::KEYSTONE;
```

Then you can use this information to create an Account instance:

```
SwiftResult<Account*>* authenticateResult = Account::authenticate(info);
```

In this SDK the results of all functions are returned as a pointer to SwiftResult class. SwiftResult class is a templated class which holds the information and payload of transactions to Swift server. SwiftResult class has the following structure:

```
template <class T>
class SwiftResult {
Poco::Net::HTTPResponse *response;
Poco::Net::HTTPClientSession *session;
SwiftError error;
/** Data **/
T payload;
}
```

response and session, are used to make transaction to the Swift server. They have useful information about your connection to the server, and in case something goes wrong you can use these to find out more about the error. For instance, you can dump your HTTPResponse to a stream (cout) as follows:

```
response->write(std::cout);
```

SwiftError is a class which contains information about the error which has happened during performing a transaction. You can see different values of error in the ErrorNo.h file. If a transaction is successful it should have code equal to zero and msg equal to “SWIFT_OK”. Therefore, the first thing to do after each transaction is to check status of error.

Payload contains the actual result of a transaction. For instance in our transaction we should expect a pointer to the Account class as payload, and we can access it using getPayload() method of SwiftResult class:

```
SwiftResult<Account*>* authenticateResult = Account::authenticate(info);
if(authenticateResult->getError() == SWIFT_OK) {
        //Do something with payload
        Account* account = authenticateResult->getPayload();
        ...
} else {
        //Ooops! Something went wrong.
        cerr <<authenticateResult->getError().msg<<endl;
```

```
}
```

After creating an account, the next step is to create or access a container. Please note that Swift does not accept whitespace in the containers or objects name.

```
Container container(authenticateResult->getPayload(), "MyContainer");
```

Notice the use of authentication result (Account) in our container! Now we have a container named “MyContainer”; remember, so far this is only a local instance and nothing has happened on the server. If you want to actually create this container on the server, you should use create function:

```
//Create container
SwiftResult<int*>* containerResult = container.swiftCreateContainer();
```

if this operation completes successfully, “MyContainer” will be created on server. All the SwifResults which have **int*** as their payload type do not contain any data and their payload is just a nullptr. If “MyContainer” already exist and you want to do something with it (or you just created it). You can use its different functions; for example, let's get the list of objects in this container:

```
SwiftResult<std::vector<Object>*>* objects = container.swiftGetObjects();
```

Having an account and container ready, we can proceed to create an object:

```
//Create Object
Object object(&container, "MyObject");
string data = "Hello Swift :)";
SwiftResult<int*> *objResult = object.swiftCreateReplaceObject(data.c_str(), data.length(), true);
```

Similar to creating a container, if this operations completes successfully Swift server will create/replace “MyObject”.

One important point to remember is that because all the results of transactions are allocated dynamically and returned as pointers it's responsibility of user to call delete on them once they are not needed anymore. Otherwise, there will be memory leaks in your program. Putting all this together, the following code snippet would be a very simple and basic usage of this SDK with the swift object storage service:

```
AuthenticationInfo info;
info.username = "username";
info.password = "****";
info.authUrl = "url to authentication server";//e.g:http://10.42.0.83:5000/v2.0/tokens
info.tenantName = "tenant name";
info.method = AuthenticationMethod::KEYSTONE;
```

```
//Do authentication
SwiftResult<Account*>* authenticateResult = Account::authenticate(info);
if(authenticateResult->getError() != SWIFT_OK){
        //Ooops! Something went wrong.
        cerr <<AuthenticateResult->getError().msg<<endl;
        delete authenticateResult;
        return -1;
}

//Create Container
Container container(authenticateResult->getPayload(), "MyContainer");
SwiftResult<int*>* containerResult = container.swiftCreateContainer();
if(containerResult->getError() != SWIFT_OK){
        //Ooops! Something went wrong.
        cerr <<containerResult->getError().msg<<endl;
        delete containerResult;
        return -1;
}
delete containerResult;


//Create Object
Object object(&container, "MyObject");
string data = "Hello Swift :)";
SwiftResult<int*> *objResult = object.swiftCreateReplaceObject(data.c_str(), data.length(), true);
if(containerResult->getError() != SWIFT_OK){
        //Ooops! Something went wrong.
        cerr <<containerResult->getError().msg<<endl;
        delete containerResult;
        return -1;
}
delete containerResult;

//Done with authenticate result as well
delete authenticateResult;
return 0;
```

API Documentation

You can find example usage of all the following functions in test.cpp file.

Account:

```
/**
* Returns all the containers under this account
* @return
* a vector of Containers
* _newest
* If set to True, Object Storage queries all replicas
* to return the most recent one. If you omit this header,
* Object Storage responds faster after it finds one valid
* replica. Because setting this header to True is more
* expensive for the back end, use it only when it is
* absolutely needed.
*/
SwiftResult<std::vector<Container>*>* swiftGetContainers(bool _newest = false);

/**
* Shows details for this account
* @return
* An stream containing details of this account
* _formatHeader
* Specifies format of returned information
* _reqMap
* You can add additional query parameters to this request. Please refer
* the Swift API documentations to see available query parameters.
* http://docs.openstack.org/api/openstack-object-storage/1.0/content/index.html
*/
SwiftResult<std::istream*>* swiftAccountDetails(HTTPHeader &_formatHeader =
HEADER_FORMAT_APPLICATION_JSON,
std::vector<HTTPHeader> *_reqMap = nullptr, bool _newest = false);

/**
* Adds metadata to this account
* @return
* Nothing
* _metaData
* A vector of string pairs (key,value)
*/
SwiftResult<int*>* swiftCreateMetadata(
std::vector<std::pair<std::string, std::string>> &_metaData,
std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* Updates an existing metadata for this account
* @return
* Nothing
* _metaData
* A vector of string pairs (key,value)
```

```
*/
SwiftResult<int*>* swiftUpdateMetadata(
std::vector<std::pair<std::string, std::string>> &_metaData,
std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* Removes specified metadata (with key) from this account
* @return
* Nothing
* _metaDataKeys
* A vector containing keys of metadata which should be removed.
*/
SwiftResult<int*>* swiftDeleteMetadata(
std::vector<std::string> &_metaDataKeys,
std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* Gets the existing metadata for this account
* @return
* Nothing. The payload is nullptr; however, the returned metadata are
* part of httpresponse. For example, getResponse()->write(cout);
*/
SwiftResult<int*>* swiftShowMetadata(bool _newest = false);
```

Container:

```
/**
* Lists the objects under this container
* @return
* A vector of Objects under this container
*/
SwiftResult<std::vector<Object>*>* swiftGetObjects(bool _newest = false);

/**
* Similar to swiftGetObjects; however, only returns the name
* of existing objects in this account.
* @return
* A stream containing names of objects under this account
* in the specified format(json/xml)
*/
SwiftResult<std::istream*>* swiftListObjects(HTTPHeader &_formatHeader = HEADER_FORMAT_APPLICATION_JSON,
std::vector<HTTPHeader> *_uriParam = nullptr, bool _newest = false);

/**
* Creates this container on the Swift server
* @return
* Noting.
*/
SwiftResult<int*>* swiftCreateContainer(std::vector<HTTPHeader> *_reqMap=nullptr);
```

```
/**
* Deletes this container from Swift server
* @return
* Noting.
*/
SwiftResult<int*>* swiftDeleteContainer();

/**
* Adds metadata to this Container
* @return
* Nothing
* _metaData
* A vector of string pairs (key,value)
*/
SwiftResult<int*>* swiftCreateMetadata(
std::vector<std::pair<std::string, std::string>> &_metaData,
std::vector<HTTPHeader> *_reqMap=nullptr);

/**
* Updates existing metadata for this Container
* @return
* Nothing
* _metaData
* A vector of string pairs (key,value)
*/
SwiftResult<int*>* swiftUpdateMetadata(
std::vector<std::pair<std::string, std::string>> &_metaData,
std::vector<HTTPHeader> *_reqMap=nullptr);

/**
* Removes specified metadata (with key) from this container
* @return
* Nothing
* _metaDataKeys
* A vector containing keys of metadata which should be removed.
*/
SwiftResult<int*>* swiftDeleteMetadata(
std::vector<std::string> &_metaDataKeys,
std::vector<HTTPHeader> *_reqMap=nullptr);

/**
* Gets the existing metadata for this container
* @return
* Nothing. The payload is nullptr; however, the returned metadata are
* part of httpresponse. For example, getResponse()->write(cout);
*/
SwiftResult<int*>* swiftShowMetadata(bool _newest = false);
```

Object:

```
/**
* Returns content of this Object
* @return
* An stream containing content of this object.
*/
SwiftResult<std::istream*>* swiftGetObjectContent(
std::vector<HTTPHeader> *_uriParams = nullptr,
std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* Creates or replace this object (if already exist)
* @return
* Nothing.
* _data
* A pointer to the data which should be written to this object.
* _size
* size of input data
* _calculateETag
* Whether to calculate ETag for this object or not; it's highly
* recommended to do so because it'll check the integrity of object
* on the server.
*/
SwiftResult<int*>* swiftCreateReplaceObject(const char* _data, ulong _size,
bool _calculateETag = true, std::vector<HTTPHeader> *_uriParams = nullptr,
std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* In order to create a variable length object you need to use this fucntion.
* You need to pass a pointer to an outputstream then you can use that to write
* as much as data you want in this object; however, there is a max limit of
* 5GB for each object.
* How to use this function:
* you should pass a outputStream pointer so you will have access to an output stream to write your data
* this pointer is valid as long as you call httpclientsession->recveiveRespone() to receive actual response
* the valid http response code is HTTP_CREATED(201).
* Here is an example to use this function:
* ostream *outStream = nullptr;
* SwiftResult<HTTPClientSession*> *chunkedResult = chucnkedObject.swiftCreateReplaceObject(outStream);
*
* (*outStream)<<mydata;
*
* HTTPResponse response;
* chunkedResult->getPayload()->receiveResponse(response);
* if(response.getStatus() == HTTP_CREATED)
* success.
*
* @return
* A pointer to the httpsession to the Swift server so you can send your request after you are
```

```
* done with writing your content to this object.
*/
SwiftResult<Poco::Net::HTTPClientSession*>* swiftCreateReplaceObject(
std::ostream* &ouputStream, std::vector<HTTPHeader> *_uriParams = nullptr,
std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* Makes a copy of this object to another object on the server.
* @return
* Nothing.
* _dstObjectname
* Name of destination object.
* _dstContainer
* Name of destination Container.
*/
SwiftResult<int*>* swiftCopyObject(const std::string &_dstObjectName,
Container &_dstContainer, std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* Deletes this object form server.
* _multipartManifest
* Used for multipart objects(not implemented yet)
*/
SwiftResult<std::istream*>* swiftDeleteObject(
bool _multipartManifest = false);
/**
* Adds metadata to this Object
* @return
* Nothing
* _metaData
* A vector of string pairs (key,value)
*/
SwiftResult<std::istream*>* swiftCreateMetadata(
std::map<std::string, std::string> &_metaData,
std::vector<HTTPHeader> *_reqMap = nullptr, bool _keepExistingMetadata =
true);
/**
* Updates existing metadata for this Object
* @return
* Nothing
* _metaData
* A vector of string pairs (key,value)
*/
SwiftResult<std::istream*>* swiftUpdateMetadata(
std::map<std::string, std::string> &_metaData,
std::vector<HTTPHeader> *_reqMap = nullptr);

/**
* Removes specified metadata (with key) from this object
* @return
```

```
* Nothing
* _metaDataKeys
* A vector containing keys of metadata which should be removed.
*/
SwiftResult<std::istream*>* swiftDeleteMetadata(
std::vector<std::string> &_metaDataKeys);
/**
* Gets the existing metadata for this object
* @return
* Nothing. The payload is nullptr; however, the returned metadata are
* part of httpresponse. For example, getResponse()->write(cout);
*/
SwiftResult<int*>* swiftShowMetadata(std::vector<HTTPHeader>* _uriParams =
nullptr, bool _newest = false);
```