

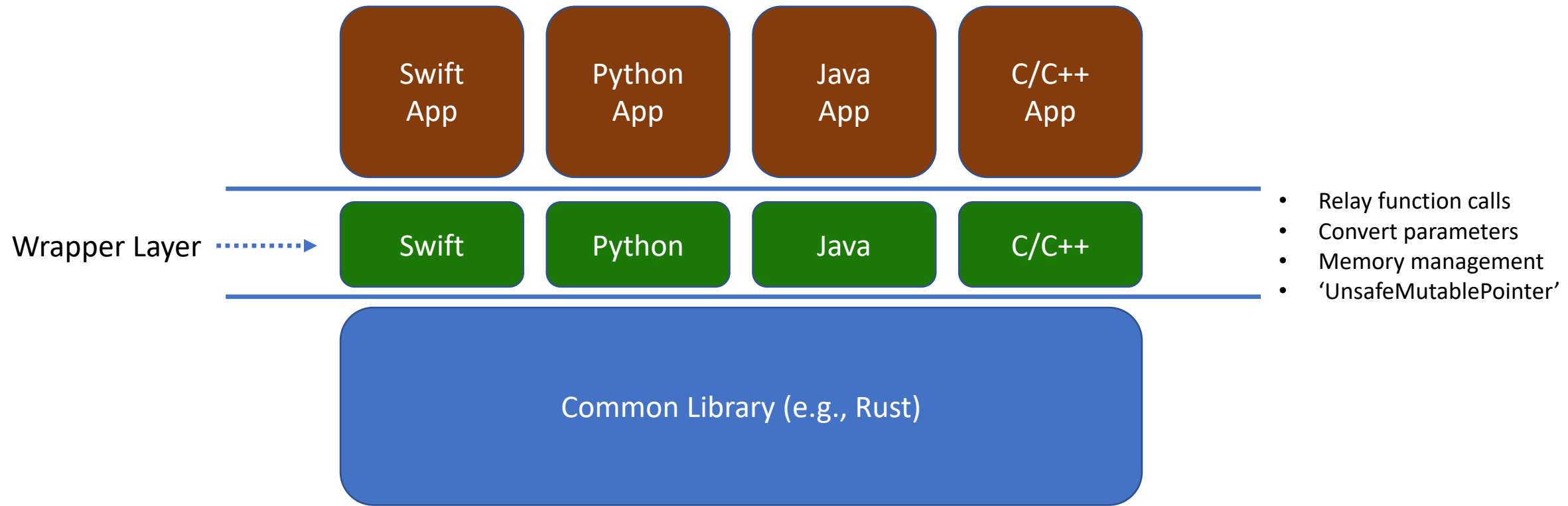
# Auto-Generating Language-Specific Wrappers for Rust Libraries

Steve McCown

[smccown@anonymome.com](mailto:smccown@anonymome.com)

4 Nov 2021

# Common Libraries with Language Wrappers



# Wrappers Require Unsafe C-Style Coding

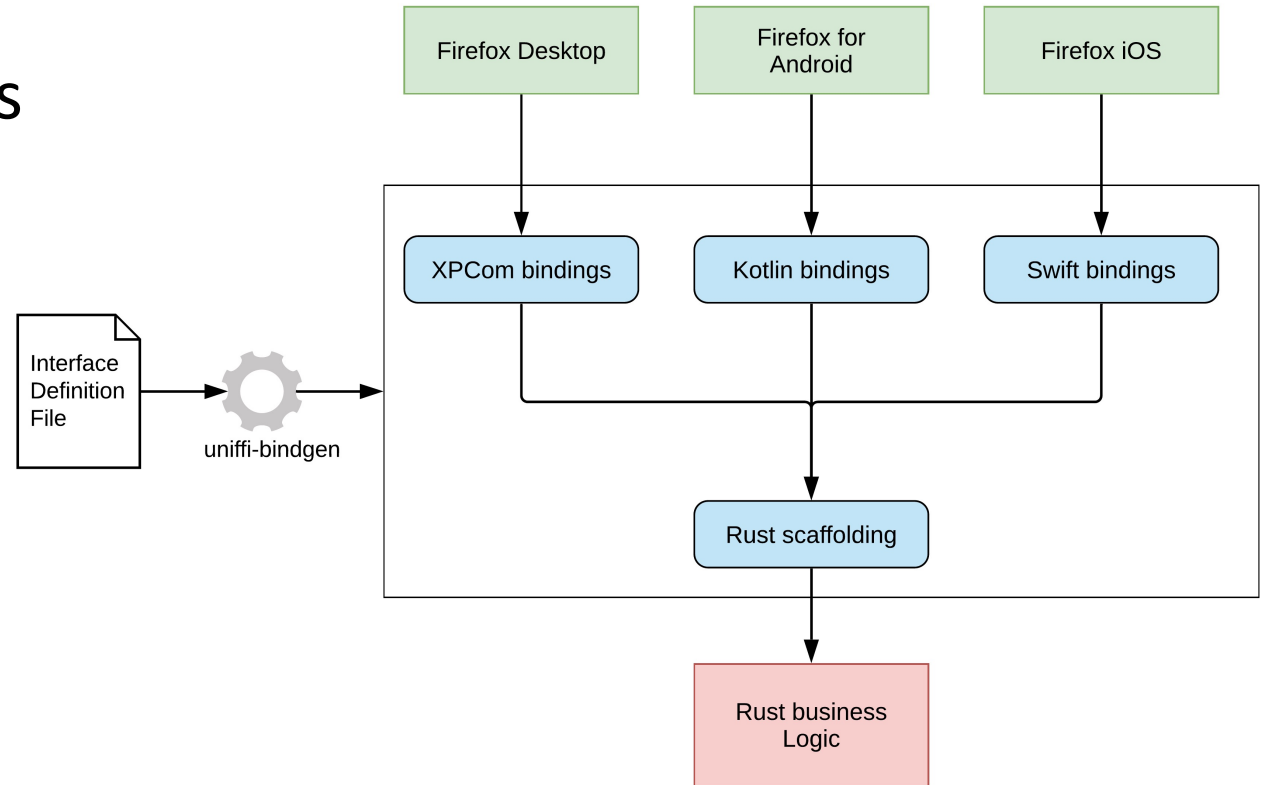
Requires all of this

Just to do this

```
5  #[no_mangle]
6  pub extern "C" fn count_characters(ptr: *const c_char) -> u32 {
7
8      // Dereference and wrap the incoming raw pointer.
9      let c_string = unsafe {
10         assert!(!ptr.is_null());
11
12         CString::from_ptr(ptr)
13     };
14
15     // Convert into a rust string.
16     let rust_string = c_string.to_str().unwrap();
17
18     // Return the number of characters.
19     rust_string.chars().count() as u32
20 }
```

# UniFFI by Mozilla

- Automatically generates foreign-language bindings for Rust libraries
- Consolidates business logic into a portable library
- Builds wrappers for
  - Kotlin
  - Swift
  - Python
  - C++
- <https://github.com/mozilla/UniFFI-rs>



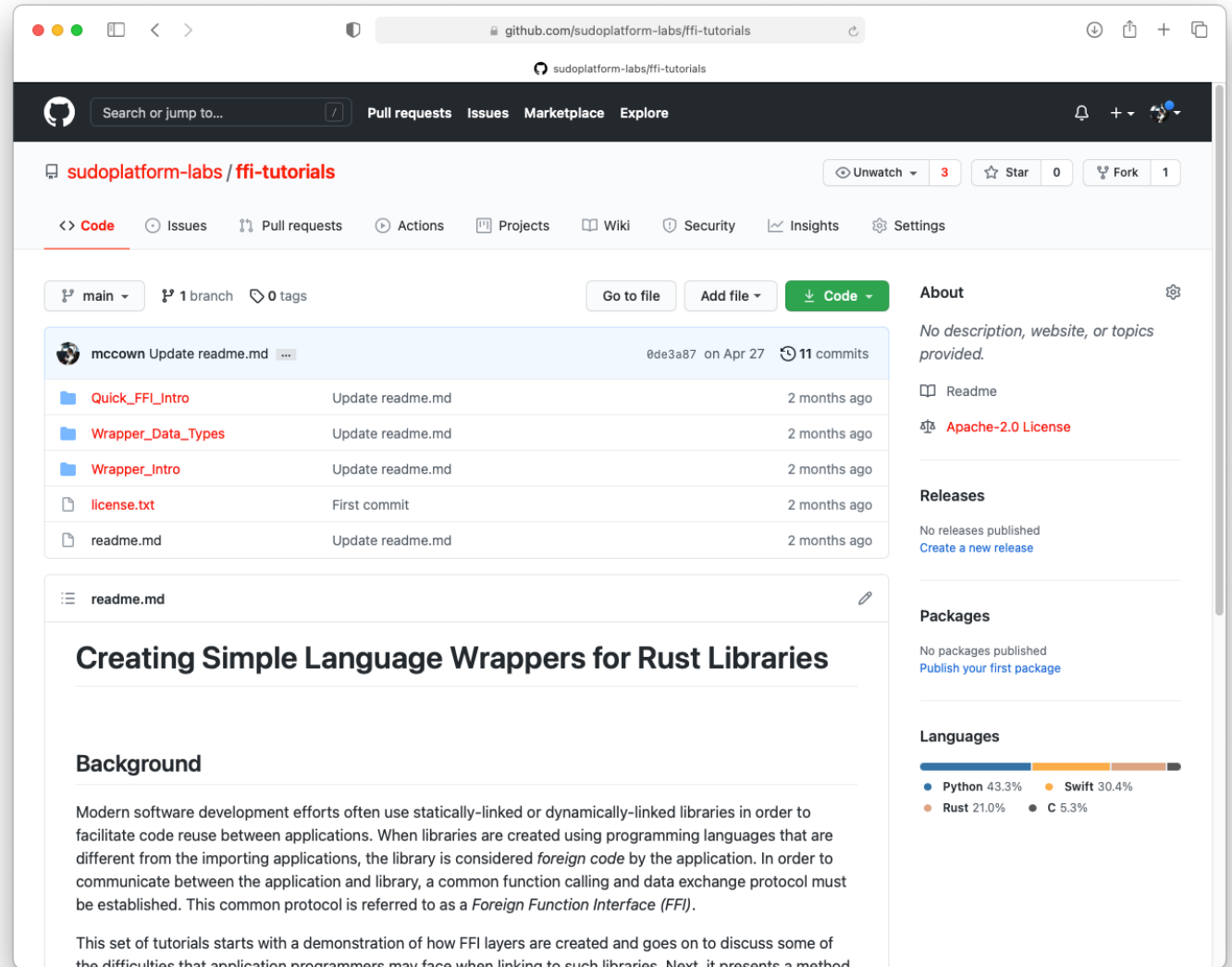
# Basic Wrapper Tutorials

## Tutorials

- Quick FFI Intro
- Wrapper Intro (simple UniFFI)
- Wrapper Data Types  
(the main types supported by UniFFI)

Source:

<https://github.com/sudoplatform-labs/ffi-tutorials>



# UniFFI: How it works

1. **Write** custom Rust library
  - Make API functions public
  - Build crate as linkable library
2. **Write** a *UDL* representation of API functions
  - Similar to Interface Definition Language (IDL)
3. **Generate** a “Scaffolding” layer
  - FFI code that creates c-style calls, memory conversions, etc.
4. **Generate** language-specific implementation layer
  - Native code layer to cover up complicated FFI calls (e.g., *Swift FFI feels like native Swift*)
5. **Import** generated code & library into native application

# Create Rust Library

Snake  
Case

```
1 include!("library.uniffi.uniffi.rs");
2
3 fn bool_inc_test(value: bool) -> bool {
4
5     return !value
6 }
7
```

lib.rs

Cargo.toml

```
1 [package]
2 name = "library"
3 version = "0.1.0"
4 authors = ["Steve McCown <smccown@anonymome.com>"]
5 license = "Apache version 2.0"
6 edition = "2018"
7 build = "build.rs"
8
9 [dependencies]
10 uniffi = "0.14"
11 # includes the 'thiserror' crate.
12 thiserror = "1.0"
13
14 [lib]
15 name = "library"
16 crate-type = ["cdylib"]
17
18 [build-dependencies]
19 uniffi_build = "0.14"
```

UniFFI additions

(NOTE: the didcomm\_rs library currently uses UniFFI version 0.14.0)

# Create UDL for API Functions

```
1 namespace library {  
2  
3     boolean bool_inc_test(boolean value);  
4  
5 };  
6
```

library.UniFFI.udl

Functions, structures, errors, enums, etc. are defined in an independent format



# Generate Scaffolding Layer

Run this

```
% uniffi-bindgen scaffolding ./src/library.uniffi.udl
```

To generate this

```
118 #[allow(clippy::all)]
119 #[doc(hidden)]
120 #[no_mangle]
121 pub extern "C" fn library_a699_bool_inc_test(
122     value: i8,
123     err: &mut uniffi::deps::ffi_support::ExternError,
124 ) -> i8 {
125     // If the provided function does not match the signature specified in the UDL
126     // then this attempt to call it will not compile, and will give guidance as to why.
127     uniffi::deps::log::debug!("library_a699_bool_inc_test");
128
129     uniffi::deps::ffi_support::call_with_output(err, || {
130         let _retval = bool_inc_test(<bool as uniffi::ViaFfi>::try_lift(value).unwrap());
131         <bool as uniffi::ViaFfi>::lower(_retval)
132     })
133 }
```

(for more details, see: `library.UniFFI.UniFFI.rs`)

# Generate a Swift Interface

Run this

```
uniffi-bindgen generate ./src/library.uniffi.udl --language swift
```

Camel Case

Generate  
library.swift

```
500 public func boolIncTest(value: Bool ) -> Bool {
501     let _retval = try! rustCall(
502
503         UniffiInternalError.unknown("rustCall")
504     ) { err in
505         library_a699_bool_inc_test(value.lower() ,err)
506     }
507     return try! Bool.lift(_retval)
508 }
509
510 }
```

# MacOS Swift App

The image shows a screenshot of the Xcode IDE. On the left, the Project Navigator displays the file structure for a project named 'swift\_test\_app'. Three blue callout boxes labeled 'Add' point to the files 'liblibrary.dylib', 'uniffi\_library-Bridging-Header.h', and 'main.swift'. The main editor window shows the source code for 'main.swift'. A blue callout box labeled 'Call the library function' points to line 15 of the code, which contains an assertion statement.

```
1 //
2 //  main.swift
3 //  test_app
4 //
5 //  Created by Steven H. McCown on 2/9/21.
6 //
7
8 import Foundation
9
10 print(" --- Running Tests --- ")
11
12 print("Running boolean test...")
13 let valueBool: Bool = false
14 var resultBool: Bool = boolIncTest(value: valueBool)
15 assert(!valueBool == resultBool, "Bool test failed")
16
17 print(" --- All Tests Succeeded! --- ")
18
```

# Generate Python Interface

Run this

```
uniffi-bindgen generate ./src/library.uniffi.udl --language python
```

Generate  
library.py

```
287 def bool_inc_test(value):  
288     value = bool(value)  
289     _retval = rust_call_with_error(InternalError, _UniFFILib.library_a699_bool_inc_test, (1 if value else 0))  
290     return (True if _retval else False)  
291
```

# Python Test App

Add

```
1 import library
2
3 value = True
4 print('\n The opposite of ' + str(value) + ' is ' + str(library.bool_inc_test(value)) + '\n')
5
```

Call the library  
function

UniFFI Applied to DIDComm\_rs

# UDL for didcomm\_rs

UniFFI specifies how Rust objects (that may contain multiple public members & methods) are presented to calling applications.

Note: from a UniFFI perspective a referenced object can only be presented as either:

1. Object (interface):
  - Contains methods
  - Passed by reference

or

2. Dictionary:
  - Contains data elements
  - Passed by value

In UniFFI, an object cannot be presented as both an object and a dictionary.

```
1 interface Message {
2
3     [Name=new]
4     constructor();
5
6     [Name=new_receive]
7     constructor([ByRef] string incomming, [ByRef] sequence<u8> sk);
8
9     sequence<string> get_to();
10    [Self=ByArc]
11    void set_to(sequence<string> str);
12
13    string get_from();
14    [Self=ByArc]
15    void set_from(string from);
16
17    sequence<u8> get_body2();
18    [Self=ByArc]
19    void set_body2([ByRef] sequence<u8> body);
20
21    string as_raw_json2();
22
23    [Self=ByArc]
24    void set_crypto_algorithm_xc20_p();
25
26    sequence<u8> unwrap_base58_key(string key);
27
28    string seal2([ByRef] sequence<u8> sk);
29
30    string set_routed_by([ByRef] sequence<u8> ekey, [ByRef] string mediator_did_string);
31 };
32
33 namespace didcomm_rs {
34
35 };
```

Note: [Self=ByArc] is used since didcomm\_rs required an Arc<T> for the self parameter (object is on the heap).

# didcomm\_rs: message.rs

Added: Using getters & setters allows access to data elements by objects without needing direct access. This allows the didcomm Message object to be specified by UniFFI as an object interface while still allowing calling applications to access and modify the object member variables.

```
130 // The capabilities provided by uniffi need getters and setters rather
131 // than accessing data directly. This is a helper function.
132 pub fn get_to(&self) -> Vec<String> {
133
134     return self.didcomm_header.read().unwrap().to.clone();
135 }
136
137 // The capabilities provided by uniffi need getters and setters rather
138 // than accessing data directly. This is a helper function.
139 // NOTE: the parameter "self: Arc<Self>" is used in place of "&mut self".
140 pub fn set_to(self: Arc<Self>, to: Vec<String>) -> () {
141
142     let mut header = self.didcomm_header.write().unwrap();
143
144     for s in to {
145         header.to.push(s.to_string());
146     }
147     while let Some(a) = header.to.iter().position(|e| e == &String::default()) {
148         header.to.remove(a);
149     }
150 }
151
152 // The capabilities provided by uniffi need getters and setters rather
153 // than manipulating data directly. This is a helper function.
154 pub fn get_from(&self) -> String {
155
156     let header = (*self).didcomm_header.read().unwrap();
157
158     match &header.from {
159         None => "".to_string(),
160         Some(x) => x.to_string(),
161     }
162 }
163
164 // The capabilities provided by uniffi need getters and setters rather
165 // than accessing data directly. This is a helper function.
166 // NOTE: the parameter "self: Arc<Self>" is used in place of "&mut self".
167 pub fn set_from(self: Arc<Self>, from: String) -> () {
168
169     let mut header = self.didcomm_header.write().unwrap();
170
171     // self.didcomm_header.from = Some(String::from(from));
172     header.from = Some(String::from(from));
173 }
174
```



Scaffolding layer is auto-generated as:  
didcomm-rs.UniFFI.UniFFI.rs

```
211  #[allow(clippy::all)]
212  #[doc(hidden)]
213  #[no_mangle]
214  pub extern "C" fn didcomm_rs_15b9_Message_set_to(
215      handle: u64,
216      str: uniffi::RustBuffer,
217      err: &mut uniffi::deps::ffi_support::ExternError,
218  ) -> () {
219      uniffi::deps::log::debug!("didcomm_rs_15b9_Message_set_to");
220      // If the method does not have the same signature as declared in the UDL, then
221      // this attempt to call it will fail with a (somewhat) helpful compiler error.
222      use uniffi::UniffiMethodCall;
223      UNIFFI_HANDLE_MAP_MESSAGE.method_call_with_output(err, handle, |obj| {
224          let _retval = Message::set_to(obj, <Vec<String> as uniffi::ViaFfi>::try_lift(str).unwrap());
225          _retval
226      })
227  }
```

Language wrapper is auto-generated as:  
didcomm-rs.swift

```
618     public func setTo(str: [String]) {  
619         try! rustCall(  
620             UniffiInternalError.unknown("rustCall")  
621         ) { err in  
622             didcomm_rs_15b9_Message_set_to(self.handle, str.lower(), err)  
623         }  
624     }
```

Once the UniFFI generation is performed, the didcomm\_rs test routines are easily called from Swift.

Source: <https://github.com/anonymo/didcomm-rs>

The screenshot shows an IDE window with a Swift file named `main.swift`. The code defines a function `send_receive_didkey_test()` that performs a DIDComm test. The function includes comments explaining the steps: creating a message, setting sender and receiver DIDs, and selecting the ChaCha20 encryption algorithm. The output window below the code shows the execution results, including the function name and a JSON-serialized plaintext message.

```
26 // send_receive_didkey_test()
27 // calls into the linked didcomm_rs library to create, encrypt, and decrypt didcomm
28 // messages. For simplicity, the actual transmission of the message is assumed to
29 // have been performed.
30 func send_receive_didkey_test() {
31
32     // Visual separator
33     print("\n-----\n")
34     print("send_receive_didkey_test()")
35     print("\n-----\n")
36
37     // For this first part, Alice creates a didcomm message to send to Bob.
38
39     // Create the DIDComm Message object.
40     let m = Message.init()
41
42     // Set the sender's DID (i.e., Alice's DID). Using "did:key" lets the public key
43     // be specified inline within the DID, so that no external DID Method or lookup is
44     // necessary. This keeps this particular test routine very simple.
45     m.setFrom(from: "did:key:z6MkiTBz1ymuepAQ4HEHYSF1H8quG5GLVVQR3djdX3mDooWp")
46
47     // Set the receiver's DID (i.e., Bob's DID). Using "did:key" lets us specify the
48     // public key inline, which keeps this test routine very simple. Since this type is an
49     // array, multiple recipients can be specified.
50     m.setTo(str: ["did:key:z6MkjchhfUsD6mmvni8mCdXHW216Xzm9bQe2mBH1P5RDjVJG"])
51
52     // Select the ChaCha20 encryption algorithm that DIDComm uses. ChaCha20 is an alternative to
53     // AES-256 that has a faster software implementation on CPUs without dedicated cryptographic
```

```
*** Starting didcomm-rs tests ***
-----

send_receive_didkey_test()
-----

Plaintext Message:
{
  "from" : "did:key:z6MkiTBz1ymuepAQ4HEHYSF1H8quG5GLVVQR3djdX3mDooWp",
  "iv" : "vhD1iZ6cgbPrzXAr14PNA2rYHb1gN8-R",
  "id" : 13548408123231832466,
  "enc" : "XC20P",
  "alg" : "ECDH-ES+A256KW",
  "type" : "application/didcomm-plain+json",
  "to" : [
    "did:key:z6MkjchhfUsD6mmvni8mCdXHW216Xzm9bQe2mBH1P5RDjVJG"
  ],
  "body" : "SGVsbG8gV29ybGQh",
  "+vsn" : "1wM"
}
```

Limitations (temporary?)

# libindy: blob\_storage.rs

Return  
Result object

Added: Method  
Without Future

```
19 // UNIFFI: Added this convenience method, so the uniffi code can connect with a function without a Future.
20 pub fn u_open_reader(xtype: &str, config_json: &str) -> Result<i32, IndyError> {
21
22     return open_reader(xtype, config_json).wait();
23 }
```

With Future

```
24
25 pub fn open_reader(xtype: &str, config_json: &str) -> Box<dyn Future<Item=IndyHandle, Error=IndyError>> {
26     let (receiver, command_handle, cb) = ClosureHandler::cb_ec_handle();
27
28     let err = _open_reader(command_handle, xtype, config_json, cb);
29
30     ResultHandler::handle(command_handle, err, receiver)
31 }
```

Core method

```
32
33 fn _open_reader(command_handle: CommandHandle, xtype: &str, config_json: &str, cb: Option<ResponseI32CB>) -> ErrorCode {
34     let xtype = c_str!(xtype);
35     let config_json = c_str!(config_json);
36
37     ErrorCode::from(unsafe { blob_storage::indy_open_blob_storage_reader(command_handle, xtype.as_ptr(), config_json.as_ptr(), cb) })
38 }
39
```

Note: UniFFI does not (currently?) support the Rust Future designator, so a companion function, `u_open_reader()`, was created without the Future designator and this method was specified in the `.udl` file.

# libindy: anoncreds.rs

Returns a tuple

```
50 pub fn issuer_create_schema(issuer_id: &str, name: &str, version: &str, attrs: &str)
51     -> Box<dyn Future<Item=(String, String), Error=IndyError>> {
52
53     let (receiver, command_handle, cb) = ClosureHandler::cb_ec_string_string();
54
55     let err = _issuer_create_schema(command_handle, issuer_id, name, version, attrs, cb);
56
57     ResultHandler::str_str(command_handle, err, receiver)
58 }
```

Note: UniFFI does not currently support the return of custom tuples. To compensate, a custom dictionary type (containing the tuple members) was created and added to `pub fn issuer_create_schema()`. This allows the data to be returned and accessed by the calling application.

Return a dictionary

```
28 [External="lib"]
29 typedef extern IndyError;
30
31 dictionary StrStr {
32     string a;
33     string b;
34 };
35
36
37
38
39 namespace anoncreds {
40
41     [Throws=IndyError]
42     StrStr issuer_create_schema([ByRef] string issuer_id, [ByRef] string name, [ByRef] string version, [ByRef] string attrs);
43 }
```

Questions?