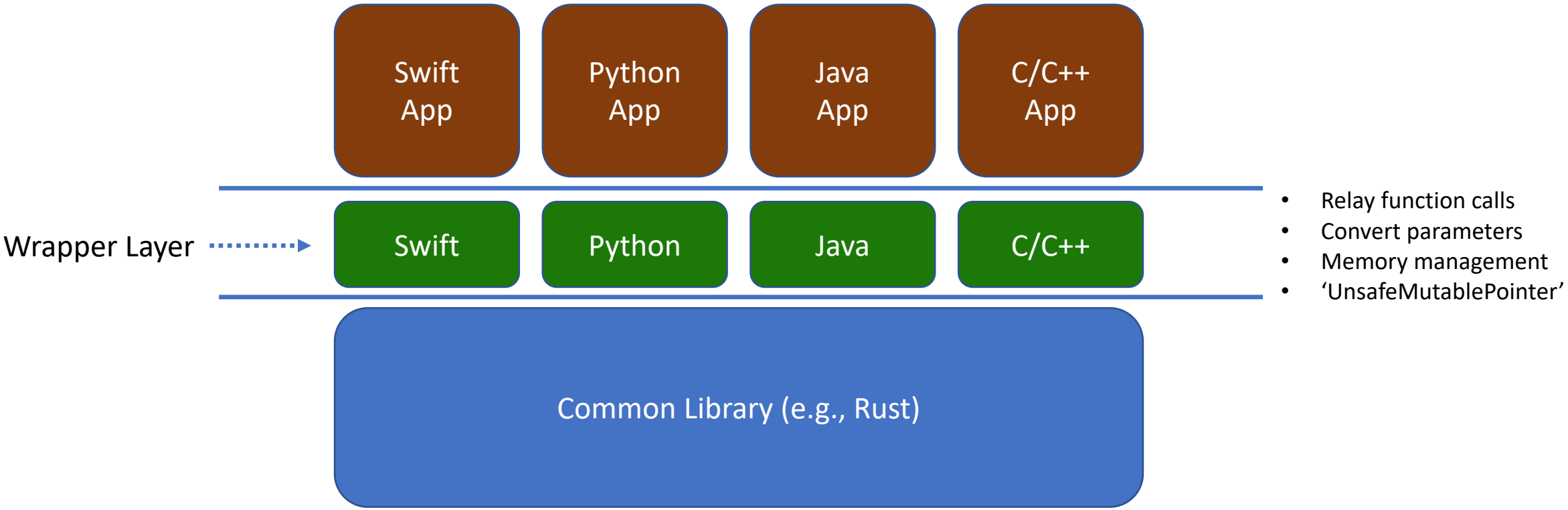
Auto-Generating Language-Specific Wrappers for Rust Libraries

Steve McCown

smccown@anonyome.com

4 Nov 2021

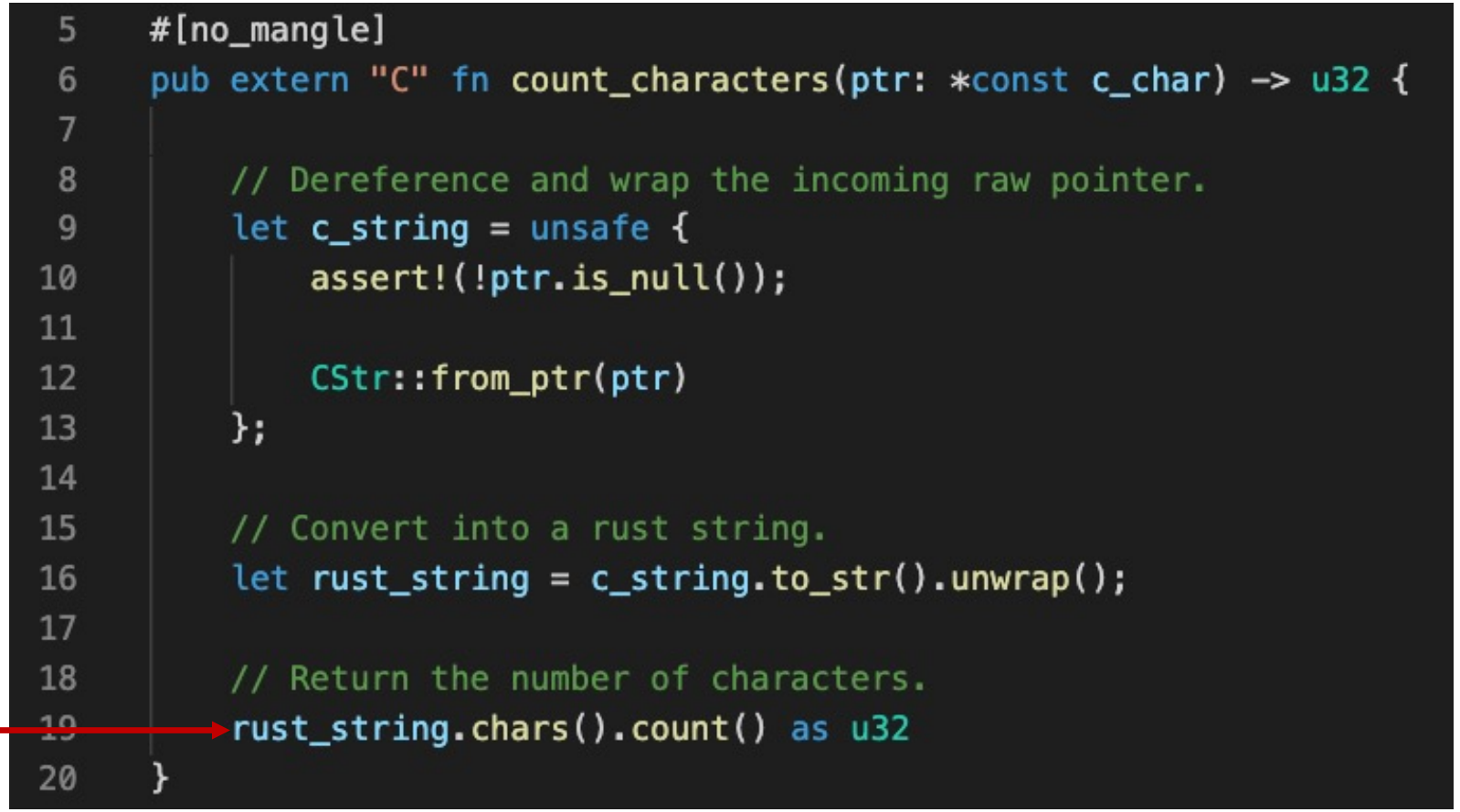
Common Libraries with Language Wrappers



Wrappers Require Unsafe C-Style Coding

Requires all of this

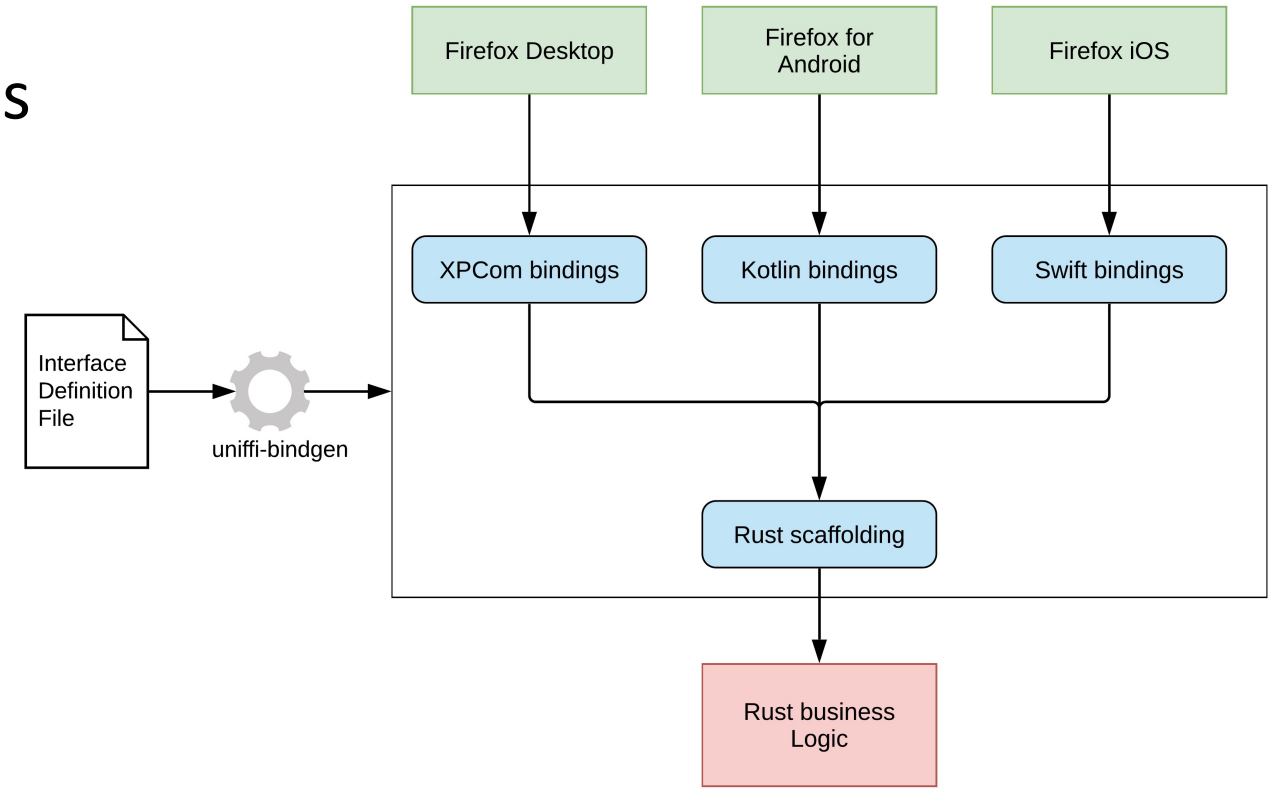
Just to do this



```
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          CStr::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

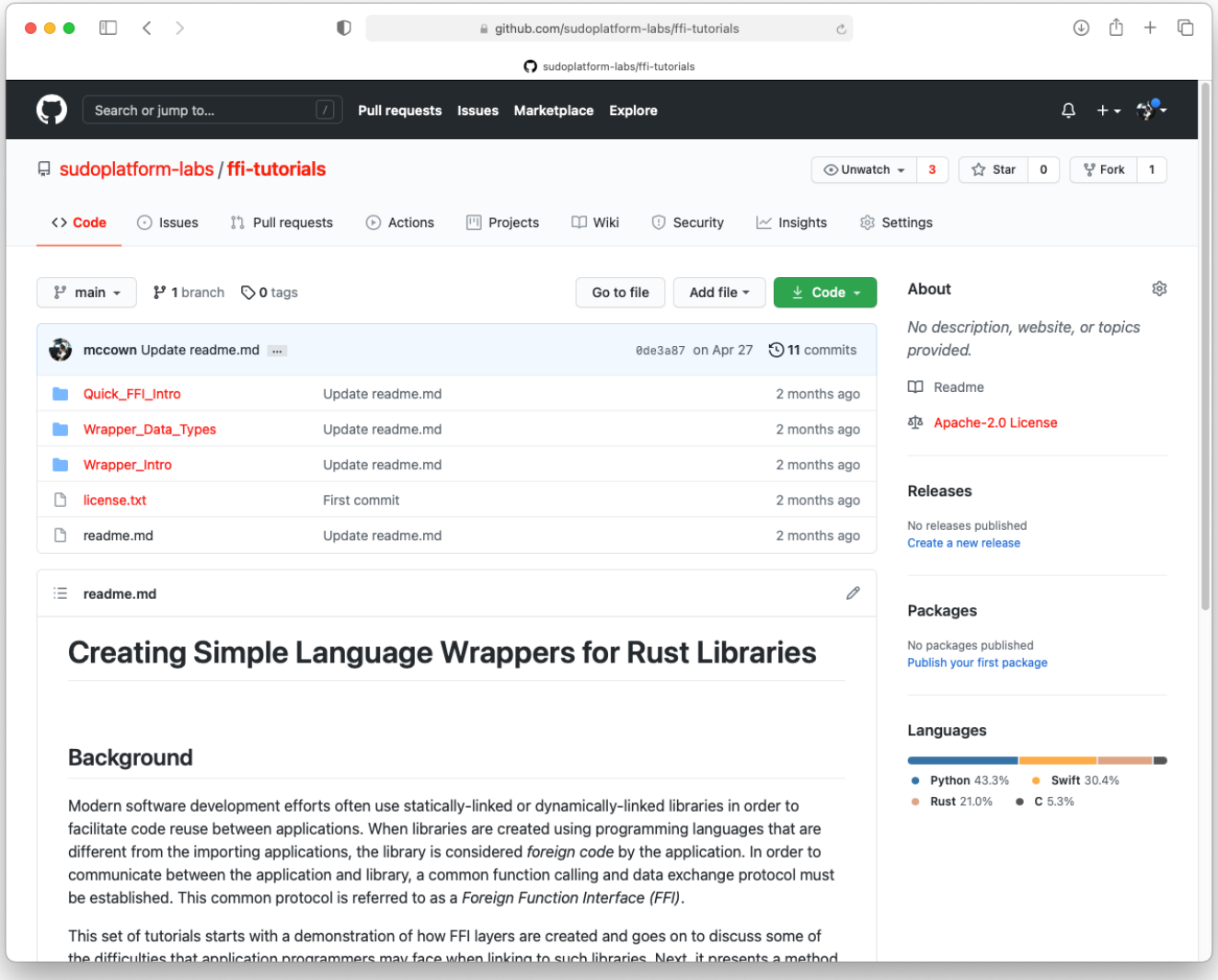


Basic Wrapper Tutorials

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

```rust
include!("library.uniffi.uniffi.rs");

fn bool_inc_test(value: bool) -> bool {

    return !value
}

```

lib.rs

Cargo.toml

```
[package]
name = "library"
version = "0.1.0"
authors = ["Steve McCown <smccown@anonyome.com>"]
license = "Apache version 2.0"
edition = "2018"
build = "build.rs"

[dependencies]
uniffi = "0.14"
# includes the 'thiserror' crate.
thiserror = "1.0"

[lib]
name = "library"
crate-type = ["cdylib"]

[build-dependencies]
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
```

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

library.UniFFI.udl

Generate Scaffolding Layer

Run this

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

To generate this

```rust
#[allow(clippy::all)]
#[doc(hidden)]
#[no_mangle]
pub extern "C" fn library_a699_bool_inc_test(
    value: i8,
    err: &mut uniffi::deps::ffi_support::ExternError,
) -> i8 {
    // If the provided function does not match the signature specified in the UDL
    // then this attempt to call it will not compile, and will give guidance as to why.
    uniffi::deps::log::debug!("library_a699_bool_inc_test");

    uniffi::deps::ffi_support::call_with_output(err, || {
        let _retval = bool_inc_test(<bool as uniffi::ViaFfi>::try_lift(value).unwrap());
        <bool as uniffi::ViaFfi>::lower(_retval)
    })
}
```

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

Generate a Swift Interface

Run this

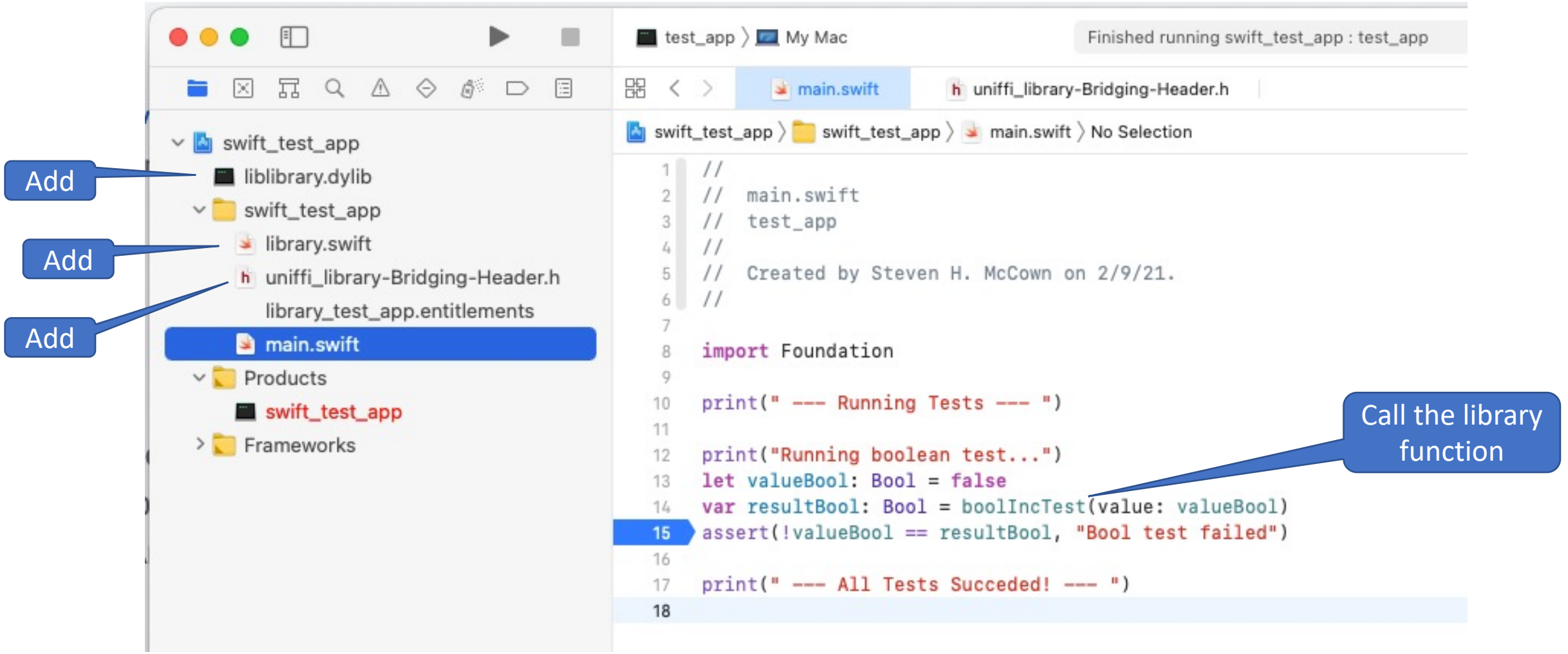
```
uniffi-bindgen generate ./src/library.uniffi.udl --language swift
```

Camel Case

Generate library.swift

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

MacOS Swift App



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

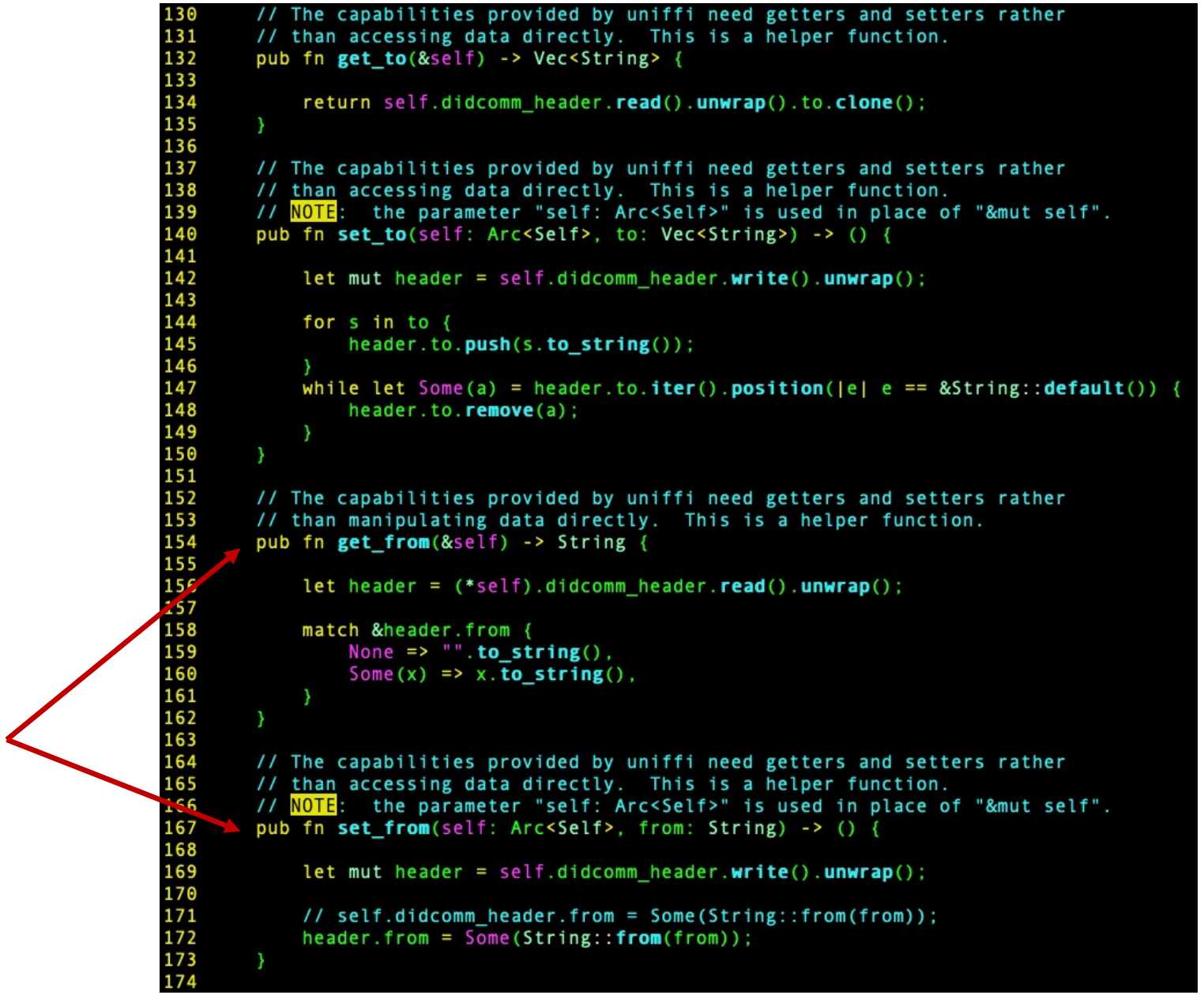
In UniFFI, an object cannot be presented as both an object and a dictionary.

```
interface Message {

    [Name=new]
    constructor();

    [Name=new_receive]
    constructor([ByRef] string incomming, [ByRef] sequence<u8> sk);

    sequence<string> get_to();
    [Self=ByArc]
    void set_to(sequence<string> str);

    string get_from();
    [Self=ByArc]
    void set_from(string from);

    sequence<u8> get_body2();
    [Self=ByArc]
    void set_body2([ByRef] sequence<u8> body);

    string as_raw_json2();

    [Self=ByArc]
    void set_crypto_algorithm_xc20_p();

    sequence<u8> unwrap_base58_key(string key);

    string seal2([ByRef] sequence<u8> sk);

    string set_routed_by([ByRef] sequence<u8> ekey, [ByRef] string mediator_did_string);
};

namespace didcomm_rs {

};
```

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

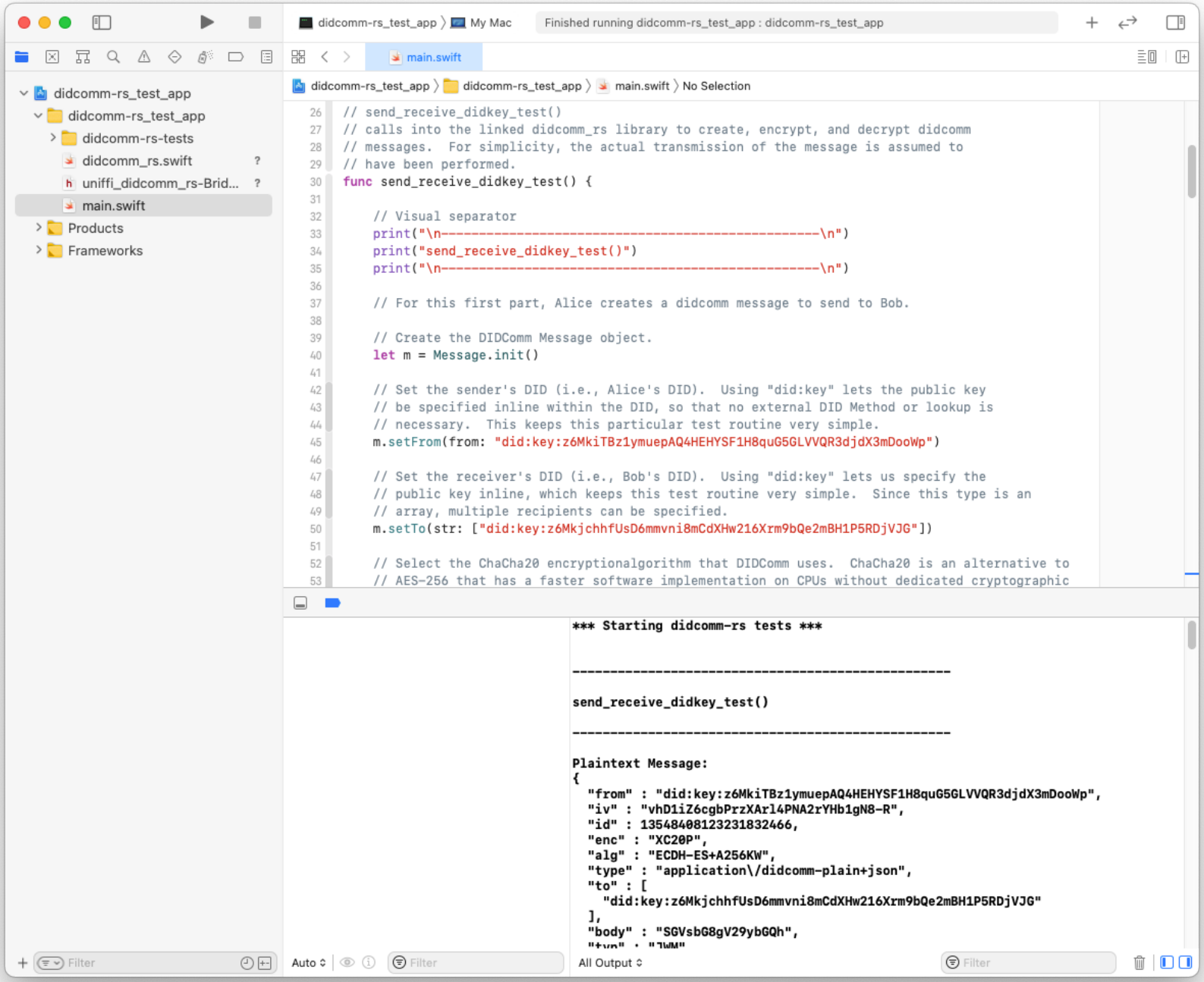
```rust
#[allow(clippy::all)]
#[doc(hidden)]
#[no_mangle]
pub extern "C" fn didcomm_rs_15b9_Message_set_to(
    handle: u64,
    str: uniffi::RustBuffer,
    err: &mut uniffi::deps::ffi_support::ExternError,
) -> () {
    uniffi::deps::log::debug!("didcomm_rs_15b9_Message_set_to");
    // If the method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

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/anonyome/didcomm-rs



Limitations (temporary?)

libindy: blob_storage.rs

Return
Result object

Added: Method
Without Future

With Future

Core method

```
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 }
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 }
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

```rust
pub fn issuer_create_schema(issuer_did: &str, name: &str, version: &str, attrs: &str)
    -> Box<dyn Future<Item=(String, String), Error=IndyError>> {

    let (receiver, command_handle, cb) = ClosureHandler::cb_ec_string_string();

    let err = _issuer_create_schema(command_handle, issuer_did, name, version, attrs, cb);

    ResultHandler::str_str(command_handle, err, receiver)
}
```

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

```
[External="lib"]
typedef extern IndyError;

dictionary StrStr {
  string a;
  string b;
};



namespace anoncreds {

    [Throws=IndyError]
    StrStr issuer_create_schema([ByRef] string issuer_did, [ByRef] string name, [ByRef] string version, [ByRef] string attrs);

```

Questions?