2.1.2.2 Proposed DIDO Solution Stack

Return to DIDO CLI Background

The proposed DIDO Solution Stack is modeled after the Database Solution Stack. The core features of both stacks is persistent storage of data and the modification of data using Transaction. The major difference is that the DIDO data objects Transactions are journaled with each Transaction being distributed to all the nodes in the node network. The details about how the Transactions are bundled, validated and verified vary amongst DIDO platforms. For example, some platforms bundle the Transactions into a block and the block is distributed once it has been verified by a mining operation. Others use "neighboring" nodes to valid and verify the Transactions.

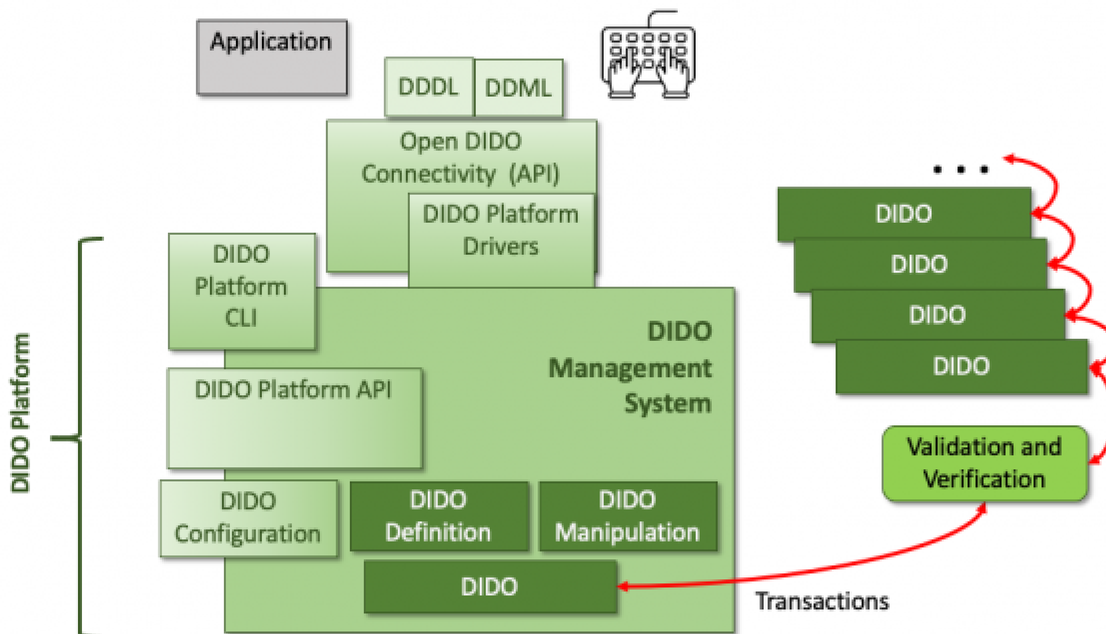

Figure 1: Proposed DIDO Management "stack"

The following is a brief description of each component in the DIDO Stack. Note, these components are normative in nature and each DIDO may be slightly different. On the left side of the diagram there the components that are good candidates for inclusion into the DIDO platform are identified.

- **Application** - The Application is similar to the Application in the Database Solution Stack, is the application, program, utility, service or microservice that needs to interact with the DIDO. This is done using standardized DIDO Data Definition Language (DDL) or Data Manipulation Language (DML) using textual commands that adhere to the DIDO-CLI specifications See: ISO/IEC 9075-01:2016 Database languages — SQL — Part 1: Framework (SQL/Framework) and related specifications.
- **Users** - And end user can enter DIDO compliant DDDL or DDML text commands into a terminal or even produce scripts that contain a series of commands that sent to the Open DIDO Connectivity API (ODAPI)for processing. Just as with the Application, these commands need to be compliant with the proposed Standard See: 4.0 DIDO Data Lifecycle Language (DDLL), 5.0 DIDO Data Definition Language (DDDL), and 6.0 DIDO Manipulation Language (DDML).

- **Open DIDO Connectivity API (Open DIDO Data Binary Connectivity Layer (ODDBCL)** - This a utility (i.e., usually a library) that the application calls to process the DIDO-CLI commands and check for validity and verification that the commands are correct.
- **DIDO Data Definition Language (DDDL)** - The DIDO Data Definition Language (DDDL) is used to create and modify the structure of DIDO objects in a DIDO. These DIDO objects include Types, objects, oracles, exchanges, aggregates, and smart contracts, etc. Often, the DDDL commands require a different set of privileges than the DIDO Data Manipulation Language (DDML).
- **DIDO Data Manipulation Language (DDML)** - The DIDO Data Manipulation Language (DDML) includes commands permitting users to manipulate (i.e., read) data in a DIDO. Note, the Data in the DIDO is immutable. This manipulation involves inserting data into DIDO Objects, making for deletion objects, creating a Transaction to update the objects and associated data from different objects together.
- **DIDO Drivers** - The DIDO Drivers are computer programs that implements a protocol (i.e., ODDBCL) for a DIDO connection.
- **DIDO Data Management System (DDMS)** - The DIDO Data Management System is a software package designed to define, manipulate, retrieve and manage DIDO Objects in a DIDO. A DIDO generally manipulates the objects itself, the object format, field names, object structure. It also defines rules to validate and manipulate this data.
    - **DIDO Data Command Line Interpreter (DDCLI)** - The Command Line Interpreter translates the tokenized DIDO commands into DIDO Platform Specific instructions.
    - **DIDO Platform CLI** - Sme DIDO platforms offer their own proprietary CLI. There have been efforts to “re-use” a particular DIDO Platform CLI on different Platforms.
    - **DIDO API** - Many DIDO Platforms provide APIs allowing a programmer to directly access a DIDO. For example, Ethereum provides a Javascript API Libraures, https://ethereum.org/en/developers/docs/apis/javascript/ or Ethereum JSON RPC, https://ethereumbuilders.gitbooks.io/guide/content/en/ethereum_json_rpc.html.
    - **DIDO Configuration** - There are always two aspects to configuring a DIDO. The first is setting up the environment externally to the DIDO Platform (i.e., downloading the software, running a wizard to install and configure the DIDO, etc), the second is the startup, upgrade, shutdown, etc of the DIDO Platform.
    - **DIDO Data Definition** - A DIDO provides a way to define Types, objects, oracles, exchanges, aggregates, and smart contracts, etc. to the DIDO.
    - **DIDO Data Manipulation** - A DIDO provides a way to manipulate the inserting data into DIDO Objects, making for deletion objects, creating a Transaction to update the objects and associated data from different objects together.
    - **DIDO Datastore** - A data store is the actually distributed data held within the DIDO (i.e., the data itself). To store the data in the Datastore, a transaction is created that can make the desired change. Depending on the DIDO Platform, the methodology for verifying and validating the transaction varies. Once it has been valdated and verified, the transaction is propagated to all the Node in the Node Network.

Proposed DIDO Solution Stack Scenarios

Return to Top

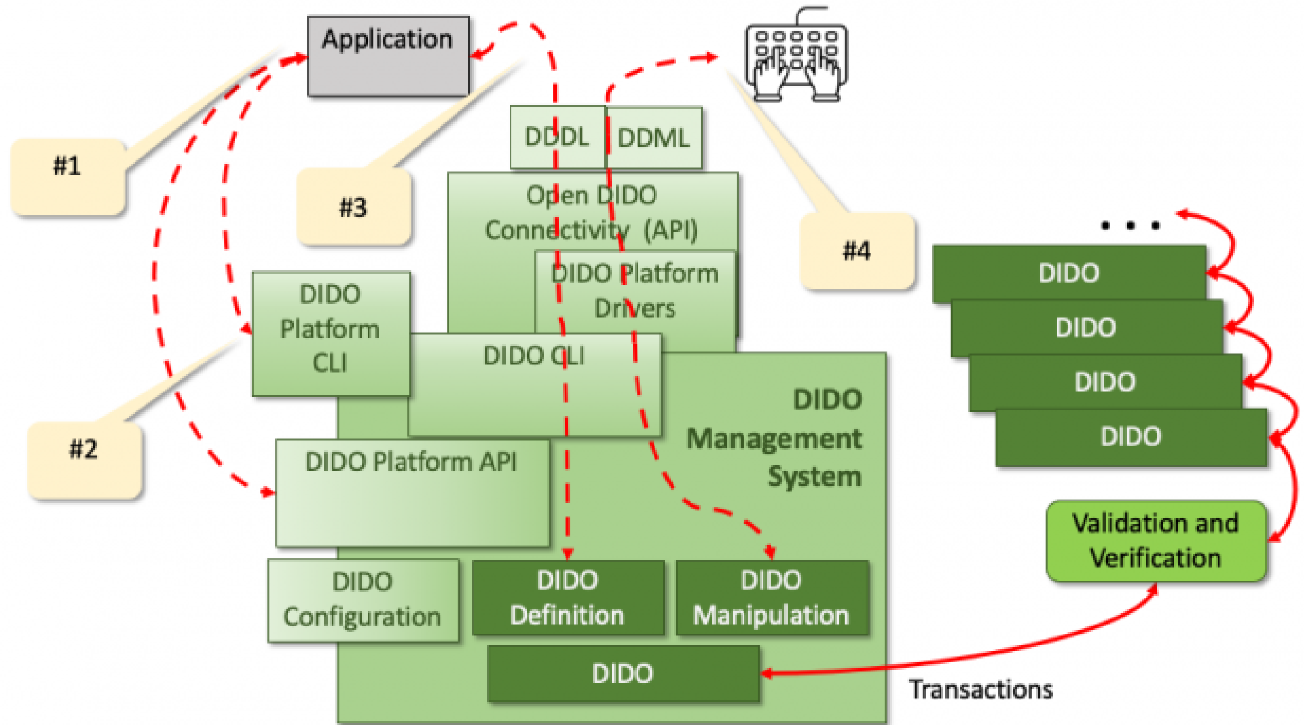

Figure 2: Three different User Scenario Paths through the DIDO Solution Stack.

Scenario #1

Return to Top

This is currently the path used by most DIDOs.The DIDO Platform provides an Application Programming Interface (API) that an Applications can be use to access the DIDO Data Store. These APIs are proprietary in nature and although there are attempts to "re-use" one proprietary API specification on another proprietary platform (interchangeable implementations) it has met with limited success. Also, there are disagreements as to what languages the API should be written in (or support). For example, if the API is written in C, many languages have a way of accessing these function. For examplem Java can ue wither the Java Native Interface (JNI) or the Java Native Access (JNA)[1].

Another issue is whether to use the Static Library and Shared Library approach. There advantages and disadvantages provided in Table 1[2].

Table 1: The Advanatages and Disadvantages to STatic or Shared Libraries.

Properties	Static Library	Shared Library
Linking time	It happens as the last step of the compilation process. After the program is placed in the memory	Shared libraries are added during linking process when executable file and libraries are added to the memory.
Means	Performed by linkers	Performed by operating system
Size	Static libraries are much bigger in size, because external programs are built in the executable file.	Dynamic libraries are much smaller, because there is only one copy of dynamic library that is kept in memory.

Properties	Static Library	Shared Library
External file changes	Executable file will have to be recompiled if any changes were applied to external files.	In shared libraries, no need to recompile the executable.
Time	Takes longer to execute, because loading into the memory happens every time while executing.	It is faster because shared library code is already in the memory.
Compatibility	Never has compatibility issue, since all code is in one executable module.	Programs are dependent on having a compatible library. Dependent program will not work if library gets removed from the system.

Note: One of the goals for DIDOs is to minimize the side effects that can arise in distributed objects. Remember, each transactions when applied to any node should result in the same output. There is no guarantee that distributed object will be deterministic if there is **"no need to recompile the executable"** when new libraries are downloaded on different Nodes. This is particularly a problem on Nodes that host other software which may download different versions of the shared library. Furthermore, each component must be deterministic in its behavior, meaning that given the same set of inputs, the outputs will always be the same. In other words, not only will all the same implementations (i.e., configurations) of a node produce the same output, but multiple implementations of a component within a node will provide the same results given the same inputs.

Scenario #2

Return to Top

Naturally, many of these APIs are focused on Representational State Transfer (REST) over Hypertext Transfer Protocol (HTTP) types of interfaces that define the standardized set of actions or verbs (i.e., GET, POST, PUT, DELETE, PATCH) as part of the Hypertext transfer protocol (HTTP) Request of what needs to be done, not the actual "what" needs to be acted upon. The what is usually sent along as a standardized document (payload) as Extensible Markup Language (XML) or JSON. also, sometimes the HTTP Request Header is used to send information to the service. The Header is a set of key-value pairs with no or little standardization as the what keys are to be returned nor what the values are. ER-20

There is a similar problem with the Hypertext transfer protocol (HTTP) Response. Although the document (payload) that is returned is in a standardized XML or JSON format, the contents of the document are not standardized. Some HTTP Responses send the results back as part of the HTTP Header which is comprised of non-standardized key-value pairs. Some examples of standardization efforts are the de facto Standard ERC-20 and the Government Standard National Information Exchange Model (NIEM) Information Exchange Package Documentation (IEPD) Specification.

Scenario #3

Return to Top

- **Note:** Senario #3 and #4 use a traditional Client/Server model as depicted in Figure