FORM Prototype Design Overview

Introduction

FORM (Flexible-grained Object-oriented Reading/Writing Model) is a configuration-driven persistence infrastructure, developed for Phlex, providing a structured approach to data storage and retrieval for scientific computing workflows. FORM serves as a mediation layer between physics frameworks and storage systems, translating high-level data product concepts into persistence operations through declarative configuration. The design maintains clean separation between framework-specific and persistence concerns, enabling future portability while meeting current Phlex requirements.

Framework Philosophy

FORM is designed around three core principles:

Clean Abstraction Boundaries: FORM maintains distinct responsibilities across layers—interface translation, persistence navigation, addressing, and I/O operations. Each layer operates through well-defined contracts, allowing independent evolution and testing of components.

Technology Neutrality: FORM's interface and orchestration logic remain independent of specific storage technologies. Configuration specifies technology choices, and FORM routes operations accordingly without exposing technology-specific details to framework code.

Configuration-Driven Approach: Users specify which data products are persisted, where they are stored, and which technologies are used through configuration objects rather than hardcoded logic. This declarative model separates I/O specification from algorithm implementation, enabling persistence strategies to evolve without modifying physics code.

Key Design Decisions

Addressing Abstraction: Placement and Token objects encapsulate physical addressing details, decoupling logical product names from storage locations and allowing flexible organization strategies.

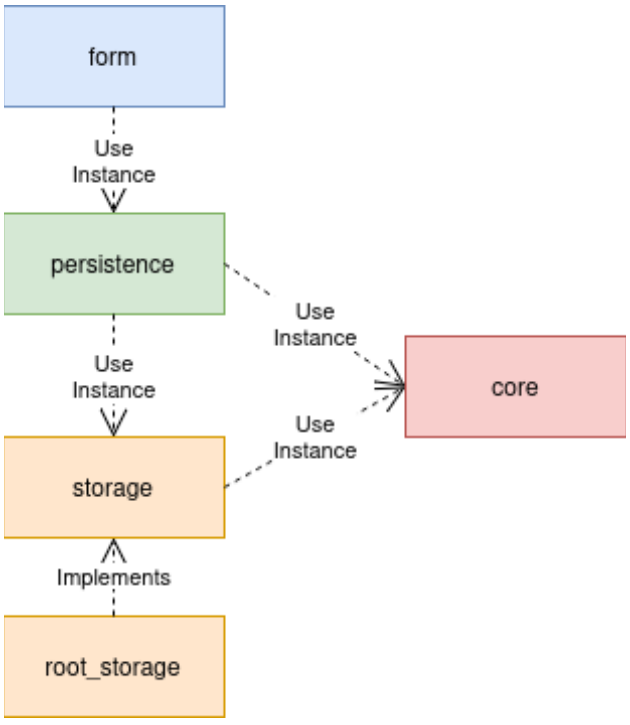
Multi-product Operation Support: The interface supports writing multiple related products in a single transaction, reducing overhead for workflows where products are naturally grouped (e.g., by event or processing segment).

Multi-file Operation Support: The interface supports seamlessly writing each data product to multiple different files. The same product can be written with either the same technology or different technologies. The user only has to request multiple files in the configuration. A single write command from the interface will then write each data product to as many files as are configured.

Technology-agnostic Interface: The framework can leverage different I/O technologies through FORM with a single interface. FORM manages the lifetimes of I/O and file handles and dispatches read and write operations in a uniform way. Technology-specific I/O implementations only have to implement FORM's storage interface.

Architecture Overview

FORM employs a layered design with distinct responsibilities:



mock_phlex Package (Framework Domain):

- Defines data product structure (`product_base`) containing metadata and payload pointers

- Provides type registry (`product_type_names`) for runtime type resolution
- Specifies persistence rules through configuration objects (`parse_config`, `PersistenceItem`)
- Will be replaced by Phlex

form Package (Interface and Translation):

- Presents primary API (`form_interface`) to framework code for read/write operations
- Translates framework configuration into FORM-internal representations
- Resolves product types and performs configuration lookups
- Supports both single-product and batch-product operations

persistence Package (Orchestration):

- Coordinates multi-product, multi-file persistence workflows
- Transforms logical product identifiers into physical addressing information
- Manages index containers for efficient product retrieval
- Delegates storage operations while maintaining technology independence

core Package (Addressing Primitives):

- Provides immutable addressing descriptors (`Placement`, `Token`)
- Encapsulates file names, container paths, and technology identifiers
- Separates write-time addressing (Placement) from read-time addressing (Token with row IDs)

storage Package (Dispatching I/O)

- Maps I/O requests in uniform format to technology-dependent implementations
- Creates I/O links to files on demand at runtime
- Links major technology and minor technology implementations
- Dispatches configuration to implementation layers

root_storage Package (Example I/O Implementation)

- Implements storage interfaces using TTree and TFile from the ROOT toolkit
- Encapsulates all ROOT dependencies
- Writes/reads data products to/from a TTree in a TFile per-TBranch

Data Flow

Write Operations: Framework creates `product_base` objects containing product metadata and data pointers. The `form_interface` consults configuration to determine target files and technologies, then delegates to the persistence layer which generates addressing information and coordinates the write workflow. `storage` translates addresses into technology-specific

write function calls. `root_storage` is an example implementation of writing to a columnar file format.

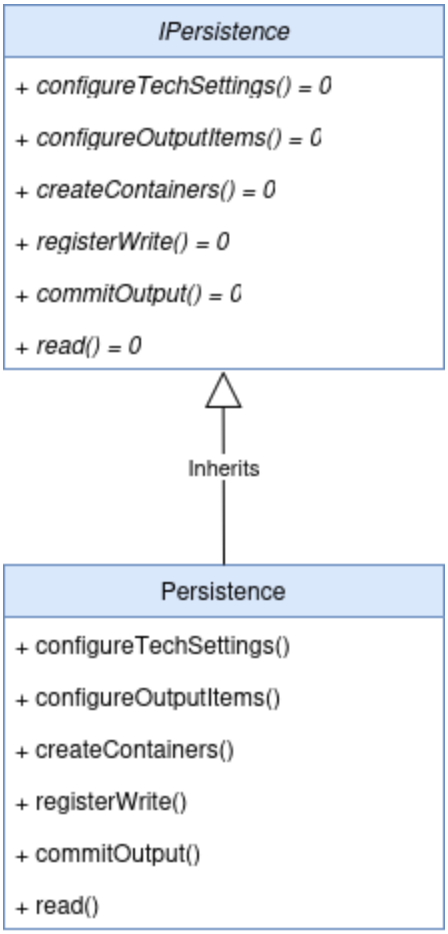
Read Operations: Framework provides product identification through `product_base` (with product ID). The `form_interface` routes the request to the persistence layer, which performs index lookups to resolve product IDs to physical row locations, generates appropriate addressing tokens, and retrieves the data. `storage` dispatches reads from a row to a technology-specific read implementation. `root_storage` is an example of reading from a columnar file format.

Configuration Flow: Framework-side configuration (`parse_config`) is translated into FORM-internal representations at interface construction time, enabling efficient runtime lookups and maintaining separation between framework and persistence concerns. Persistence forwards configuration information to the storage layer which constructs technology-specific read, write, and file operation implementations on-demand. Storage sorts configuration information by major and minor technology so that each I/O component only receives relevant configuration parameters.

FORM Interface Design

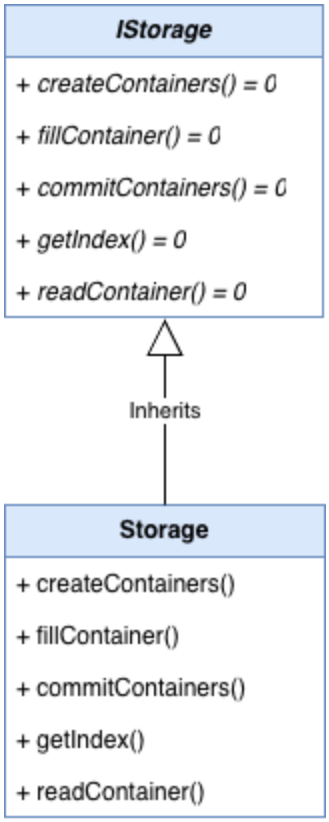
Interface Hierarchy

The FORM framework employs a minimal inheritance hierarchy focused on abstraction of the persistence layer:

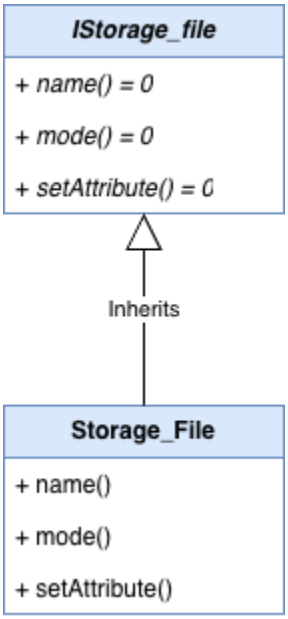


Design Rationale: The single inheritance relationship isolates the persistence contract from its implementation, enabling alternative persistence strategies while maintaining a stable interface for the FORM layer.

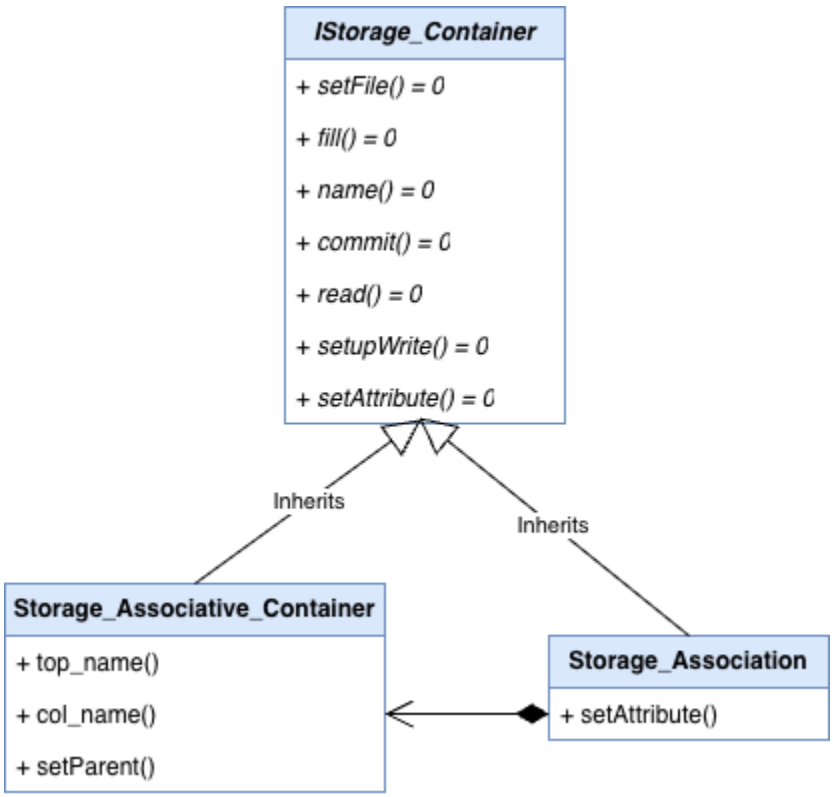
The Storage layer has a more complex inheritance hierarchy with three layers of interfaces:



Design Rationale: The Storage interface provides a single entry point for Placement to dispatch read- and write-related operations. FORM comes with a default Storage implementation that looks up files and containers based on information in a placement and creates a file and/or container on demand when one is not found. In the default Storage implementation, all state is handled by file and container interfaces after they are created.



Design Rationale: IStorage_File is a handle to a filesystem resource whose lifetime is managed by an IStorage implementation.

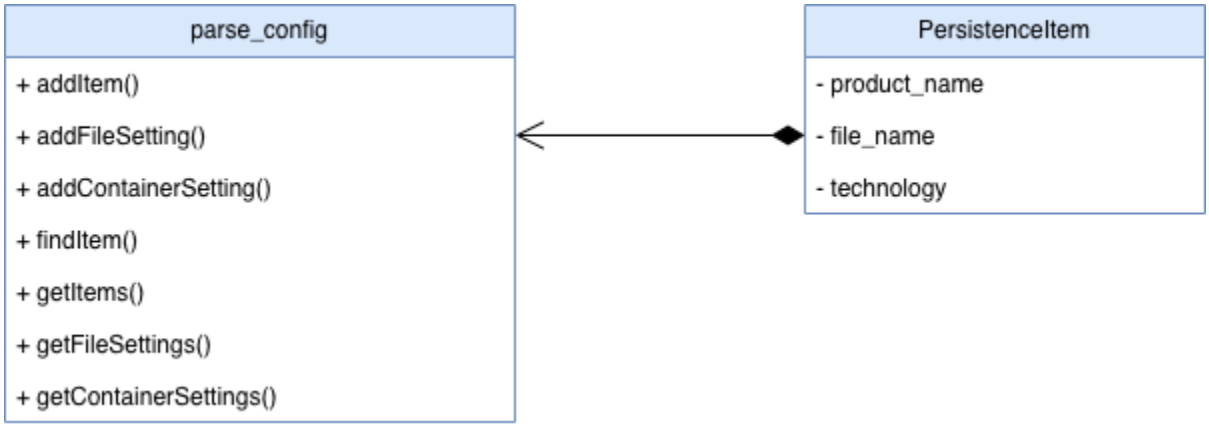


Design Rationale: An IStorage_Container is a location where one data product can be written to an associated file. An IStorage_Container is associated with a file by an IStorage implementation. A Storage_Associative_Container is an IStorage_Container that communicates with a parent IStorage_Container. The parent is expected to be a Storage_Association.

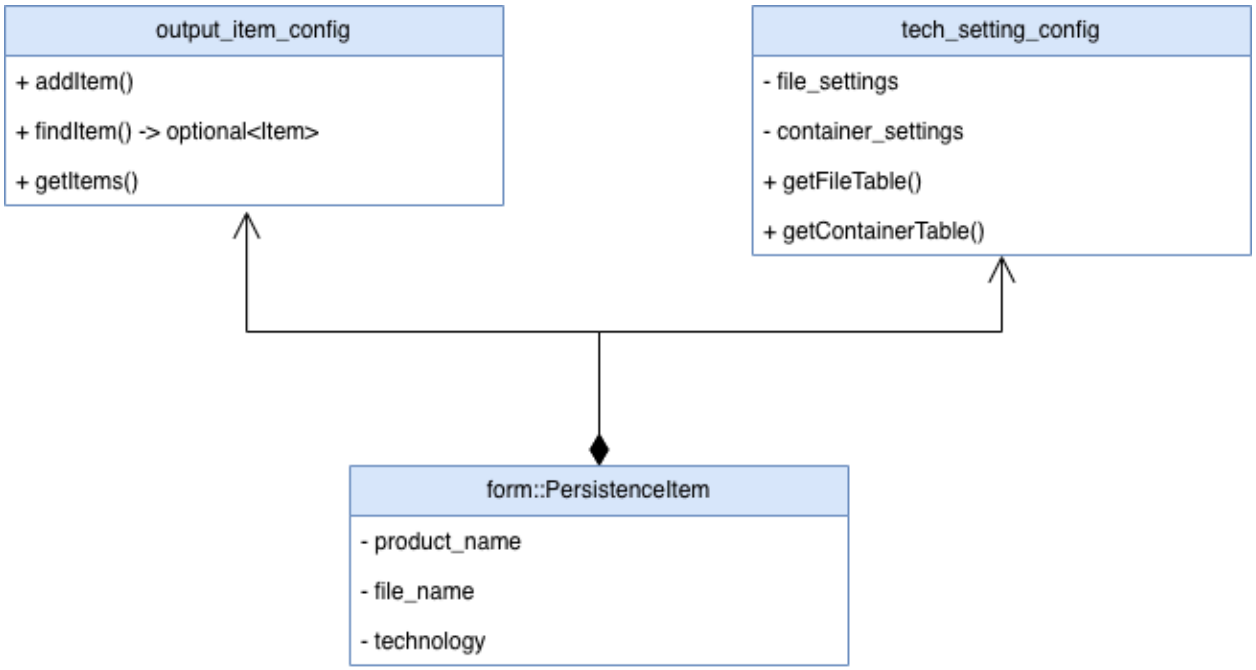
FORM Configuration Classes (No Inheritance)

Configuration classes exist in two parallel hierarchies maintaining framework independence:

mock_phlex Configuration Domain:

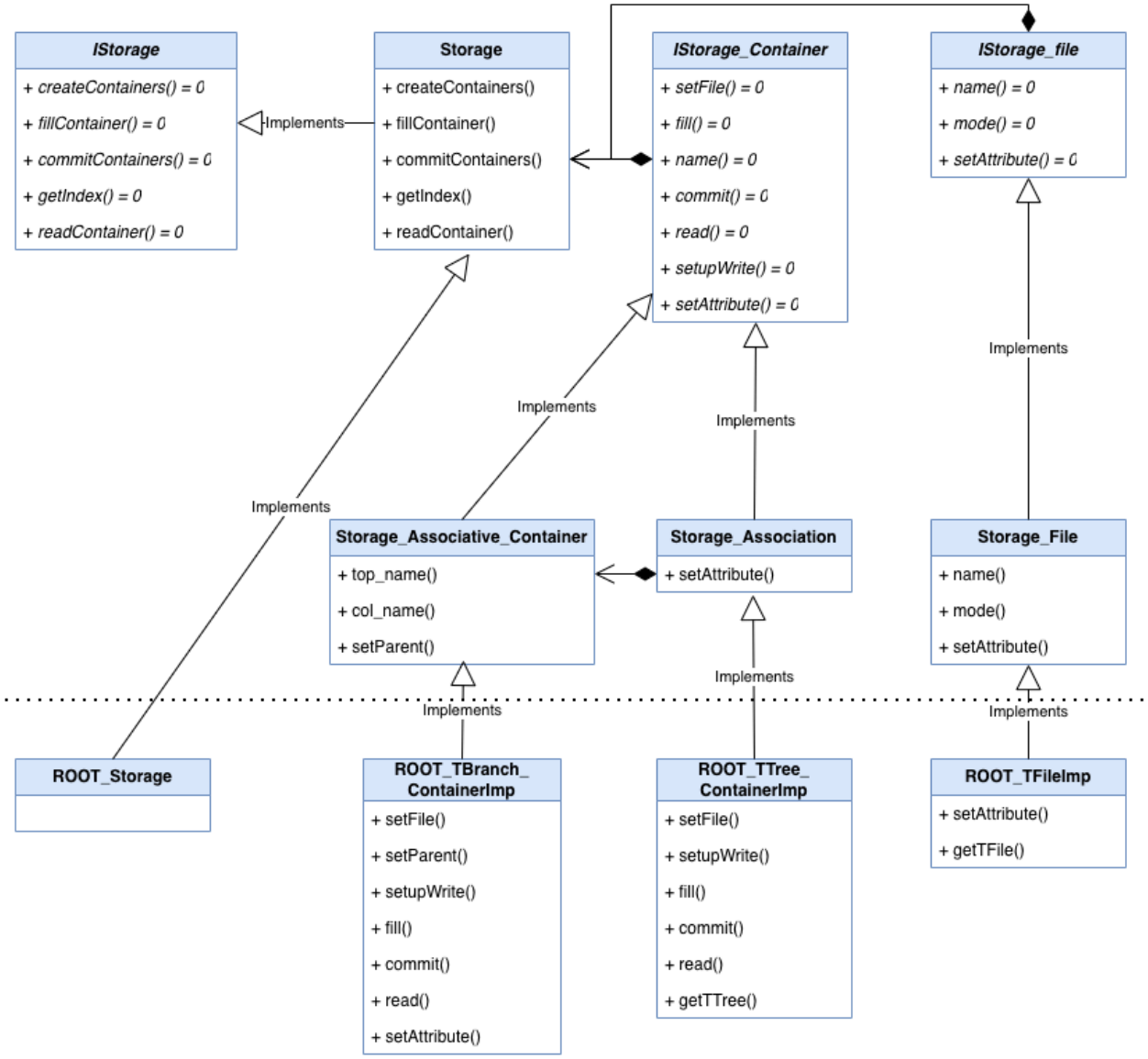


FORM Configuration Domain:



Design Rationale: Two parallel PersistenceItem types (structurally identical but in different namespaces) maintain clear framework boundaries. mock_phlex configuration can evolve independently of FORM internals. Translation occurs once at `form_interface` construction.

FORM Storage Class Relationships



Design Rationale: Storage separates `IStorage_File` implementations from `IStorage_Container`: major technology and minor technology respectively. An `IStorage_Container` encapsulates how one data product type is read and written to and from a file. `Storage_Association` was introduced to better model the lifetime of `TTree` and other associative container technologies that share a handle to a file when creating new containers during the event loop. A `Storage_Associative_Container` implements the other half of this pattern. It is an `IStorage_Container` that uses a parent `Storage_Association` to connect to a file. `ROOT_TTreeContainerImp`,

`ROOT_TBranch_ContainerImp`, and `ROOT_TFileImp` form a concrete example of how to use CERN's ROOT library to implement FORM's Storage interfaces.

Composition

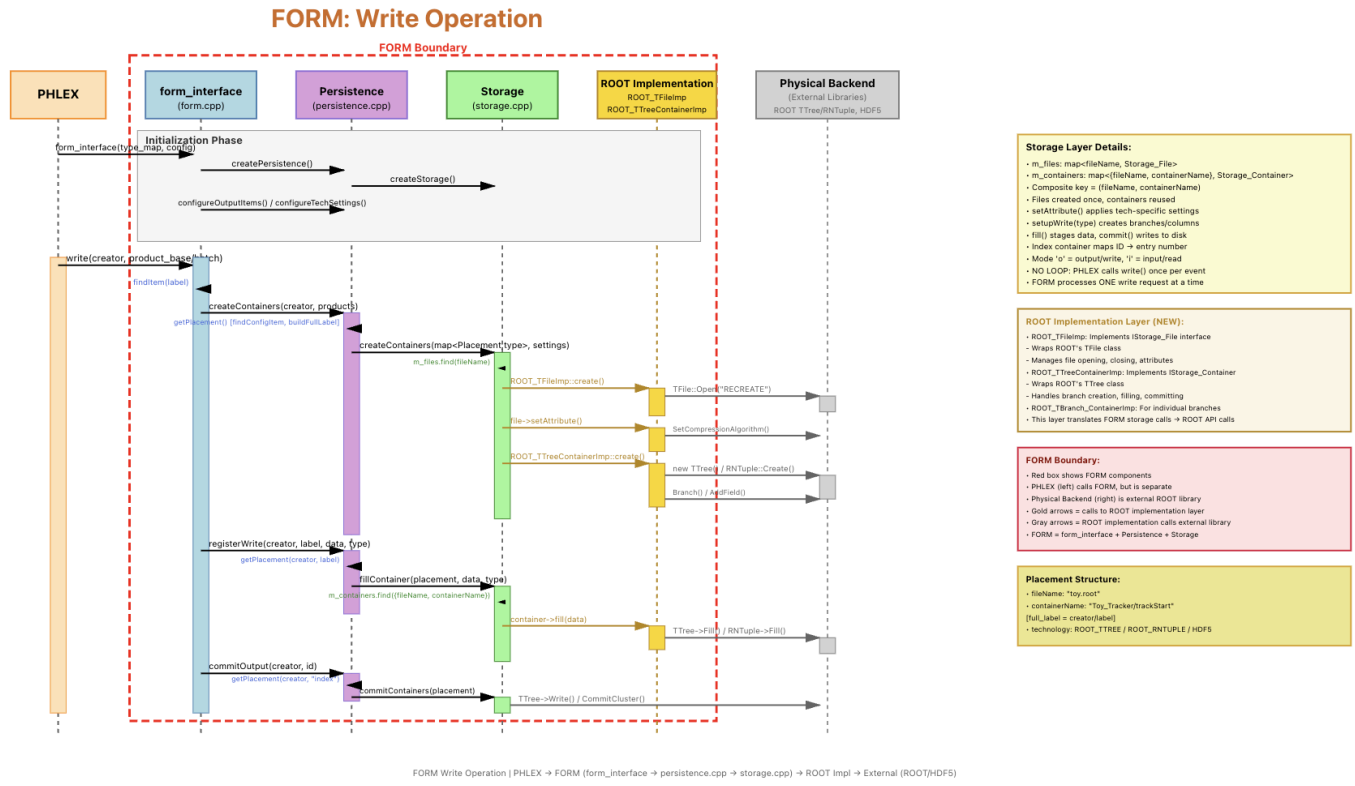
The composition of FORM centers on a clear top-down flow from the framework interface (`form_interface`) to technology-specific storage implementations. Each layer composes instances of the layer below it, translating high-level requests into progressively more concrete operations.

1. **Interface to Persistence (form → persistence)**
   The `form_interface` class serves as the framework's entry point. It receives data products from Phlex (or mock_phlex) and interprets configuration objects to determine where and how each product should be persisted.
   Internally, `form_interface` **owns an instance of** **`Persistence`** through the abstract interface `IPersistence`. This composition isolates framework logic from persistence orchestration, allowing alternative persistence strategies without affecting the interface layer.
2. **Persistence to Storage (persistence → storage)**
   The `Persistence` class orchestrates all write and read operations. It **uses an instance of** **`IStorage`** to delegate low-level I/O handling while remaining technology-agnostic.
   `Persistence` is responsible for constructing addressing descriptors (`Placement`, `Token`) and managing index containers that connect logical product names and identifiers to their physical storage locations. Once the addressing is resolved, `Persistence` calls into `Storage` to perform actual I/O operations.
3. **Storage to Technology Implementations (storage → root_storage and others)**
   The `Storage` class acts as a dispatcher and cache manager. It **composes collections of** **`IStorage_File`** **and** **`IStorage_Container`** **instances**, creating them on demand based on configuration provided by `Persistence`. `Storage` separates file management (`IStorage_File`) from data container management (`IStorage_Container`).
   - A `Storage_Association` manages the relationship between files and containers, ensuring that containers sharing a file handle remain synchronized.

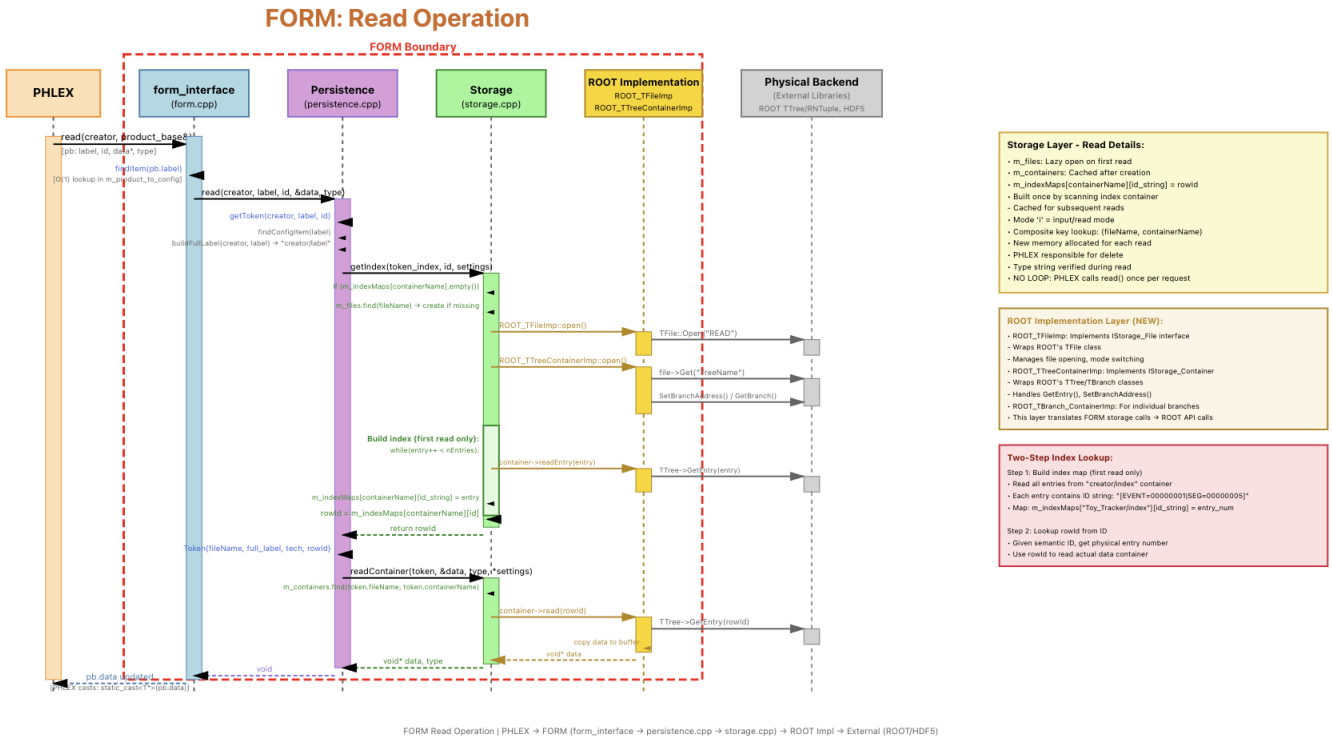
    - A `Storage_Associative_Container` references its parent association to access shared file handles during write and read operations.
4. **ROOT Storage Backend (root_storage)**
   The `ROOT_Storage` package provides a concrete implementation of the storage interfaces for ROOT I/O.
    - `ROOT_TFileImp` implements `IStorage_File`, encapsulating ROOT's `TFile`.
    - `ROOT_TTree_ContainerImp` and `ROOT_TBranch_ContainerImp` implement `IStorage_Container`, mapping FORM's logical containers and columns to ROOT's `TTree` and `TBranch` structures.
      These components are instantiated and managed by `Storage`, but their lifetimes are tied to the file associations that own them.
5. **Core Addressing (core)**
   The `core` package provides lightweight, immutable classes (`Placement`, `Token`) used throughout `Persistence` and `Storage`. These classes are **value-type members**, not polymorphic objects, ensuring safe and predictable ownership semantics while clearly separating write-time and read-time addressing.

Sequence Diagrams

Write Operation: This diagram illustrates how a write request flows through FORM, from the framework layer down to the persistence and storage layers. PHLEX calls FORM's write() method once per event; FORM processes one write operation at a time without internal loops. The diagram shows the ROOT implementation layer (ROOT_TFileImp, ROOT_TTreeContainerImp) that translates FORM's storage interface calls into ROOT-specific API calls.

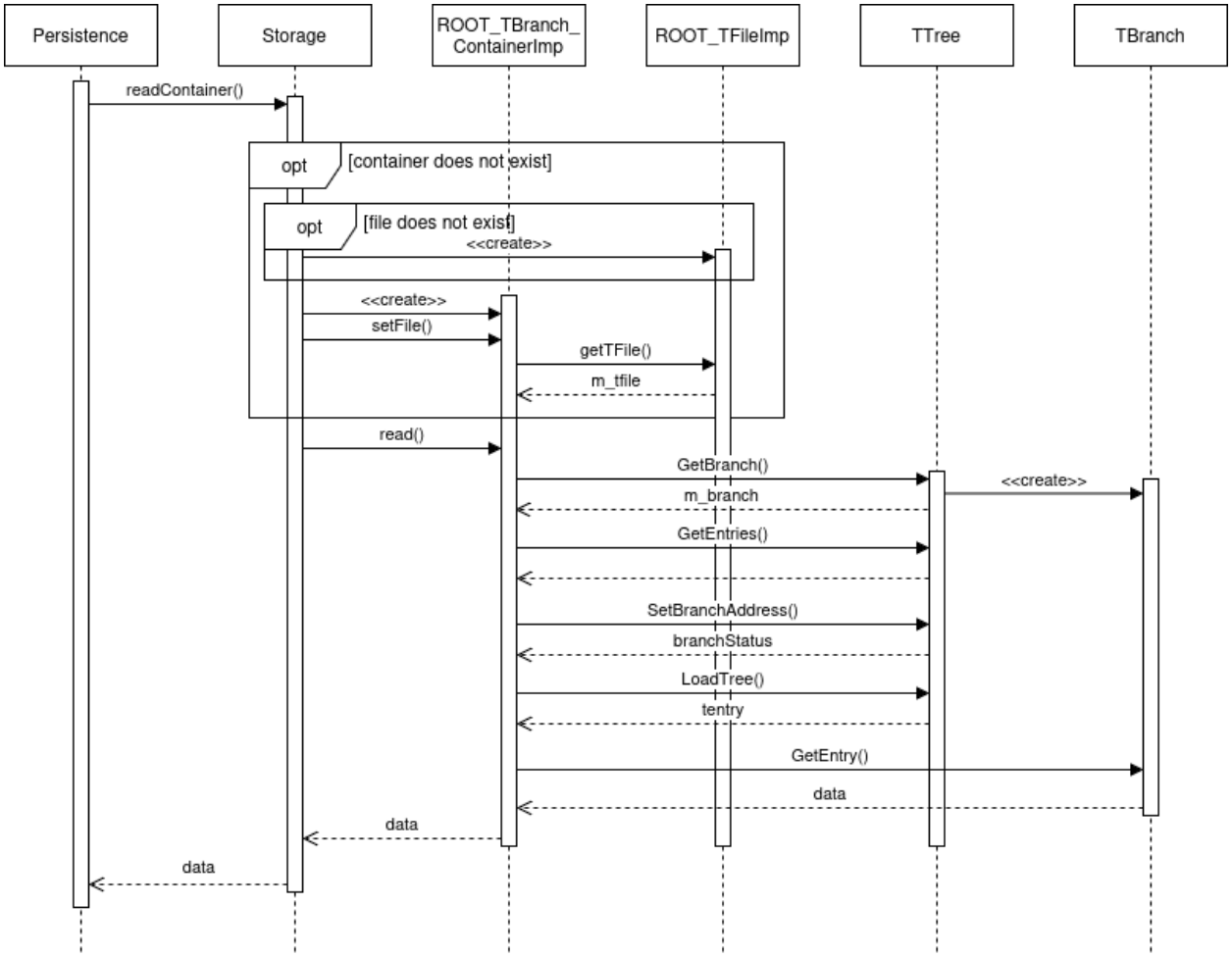


Read Operation: This diagram shows the full read path inside FORM, beginning with a product lookup request and continuing through index resolution, token construction, and backend-specific data retrieval. PHLEX calls FORM's read() method once per request; FORM processes one read operation at a time without internal loops. The diagram shows the ROOT implementation layer that translates FORM's storage interface calls into ROOT-specific API calls.

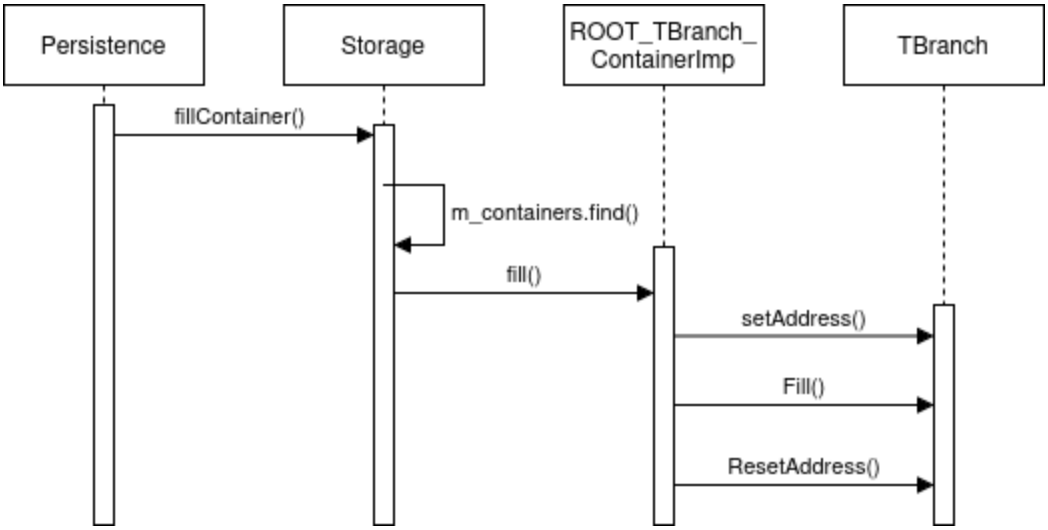


Storage Sequence: This sequence focuses on the internal mechanics of the Storage layer, illustrating in detail how a ROOT TTree backend processes a readContainer call. It exposes the low-level interactions between Storage, the ROOT_TTreeContainerImp, TFile, TTree, and TBranch components that allow FORM to retrieve a single data product from a ROOT columnar file.

The storage package has a deep hierarchy of dependencies between objects at runtime. The default implementation of IStorage::readContainer() using a ROOT_TTreeContainerImp illustrates how the components of this package work together.



Most operations in the storage package are much simpler than IStorage::readContainer(). IStorage::fillContainer() is a good example of the complexity of other functions in the storage package.


Persistence
Storage
ROOT_TBranch_
ContainerImp
TBranch
fillContainer()
m_containers.find()
fill()
setAddress()
Fill()
ResetAddress()