# Design proposal: Namespaces

## Use cases

- **Name conflicts** – dataset names can easily conflict when using general terms, like “events” or “users”. Namespaces provide a way to use simple names and avoid or resolve collisions.
- **Logical grouping** – storing data for multiple instances of an application or experiment is simpler using namespaces to logically group datasets. The dataset names in the application do not change, just the namespace as a configuration option.
- **Parity with existing systems** – both traditional and hadoop-based (Hive) stores commonly provide namespaces (databases). Adding namespaces to Kite allows a 1-to-1 mapping with these existing systems.

### Why not use DatasetRepository?

The original reference guide suggests using the DatasetRepository as a similar solution:

> “Developers may organize datasets into different dataset repositories for reasons related to logical grouping, security . . . ”

This is an option, but has drawbacks:

- Potential DatasetRepository instances can’t be listed. Using repositories for logical grouping means not being able to discover what groups are available. Some back-ends, like Hive, can list namespaces, but there is no generic way to do this for HDFS.
- DatasetRepository doesn’t have a 1-to-1 mapping with existing back-ends. Work-arounds are possible by either interpreting the dataset name[1] or adding a namespace to the repository URI. But these rely on documentation to be clear about how that mapping is done for each repository and adds implementation-specific conventions.
- Using repositories for namespacing means that some applications will use multiple DatasetRepository instances. Application developers are focused on datasets and interaction with repositories should be kept to a minimum.
- Multiple repositories for a some backends are confusing. For example, if a DatasetRepository is mapped to a Hive database, it is unclear why there are multiple Hive “repositories”.

[1] In this case, the user is required to concatenate namespace and dataset names, which is error-prone compared to a API supported approach.

# Proposed changes

Namespaces are *optional*. When the namespace argument is not present, the dataset name will be split on `'.'` to recover a dataset name and, if present, a namespace that precedes it. Application developers must add a namespace if the dataset name already contains a `'.'` character. If the namespace is not given, then it will be `default`.

The main changes to `DatasetRepository` are to add a namespace string argument to any methods that currently require only a dataset name[2]. This creates a 1-to-1 mapping with Hive and other ackends that support namespaces. The `list` method is replaced by two methods to list namespaces and datasets within a namespace.

### API

```
DatasetRepository:
    <E> Dataset<E> create(String namespace, String name, DatasetDescriptor
desc)
    <E> Dataset<E> create(String name, DatasetDescriptor desc)
    <E> Dataset<E> load(String namespace, String name)
    <E> Dataset<E> load(String name)
    <E> Dataset<E> update(String namespace, String name, DatasetDescriptor
desc)
    <E> Dataset<E> update(String name, DatasetDescriptor desc)
    boolean delete(String namespace, String name)
    boolean delete(String name)
    boolean exists(String namespace, String dataset)
    boolean exists(String namespace)
    List<String> datasets(String namespace)
    List<String> namespaces()
    ~~List<String> list()~~
```

Namespaces are supported as arguments to encourage applications to handle the namespace component separately so that it can be changed through configuration.

# Implementation details

### FileSystem

Namespaces will be represented in the FS implementation as an extra folder layer within a repository. New datasets will be created in the correct location, inside a namespace folder. Any existing datasets will not be changed to avoid breaking expectations in other systems like flume.

---

[2] Backward-compatibility can be handled for most cases by implementing the methods with only dataset name arguments in `AbstractDataset`, which calls the methods with namespace arguments.

When the `default` namespace is passed to any method, the FS DatasetRepository will first check for the dataset at the new location, within the `default/` folder, and also check in the old location, directly in the repository folder.

## Hive

Kite namespaces will be represented as Hive databases. External tables will still rely on the FS implementation, which will use a namespace folder under the repository storage location.

Both managed and external repositories will load any dataset created by Kite. The only difference between external and managed repositories will be where data is stored when the repository creates new datasets[3]. Hive tables that were not created by Kite (or cannot be opened) will not be listed by `namespaces` or `datasets`[4].

Hive managed tables are currently stored in the `default` database. The proposed updates will access existing tables without modification. Existing location URIs will be used to access old datasets.

## HBase

Kite namespaces will be represented using HBase namespaces, available in HBase 0.96.0 forward. For versions prior to 0.96, namespaces will be embedded in the table name using HBase's separator, `':'`.

When the `default` namespace is passed to a repository method, the HBase support will first check for the expected name, with namespace set correctly or embedded. For versions before 0.96.0, Kite should check for the dataset name without the default namespace embedded. In 0.96.0, *any* failed lookup should be followed by a secondary check to see if the default namespace has a table with the Kite namespace embedded to handle migrations from previous versions.

---

[3] This isn't quite true: to delete an existing dataset, the repository must delete external data and let Hive delete managed data.

[4] In the future, we *could* use HCatalog to open any Hive table, but this would require significant work to translate from Hive objects to Avro.