# Chapter 3: Design Overview

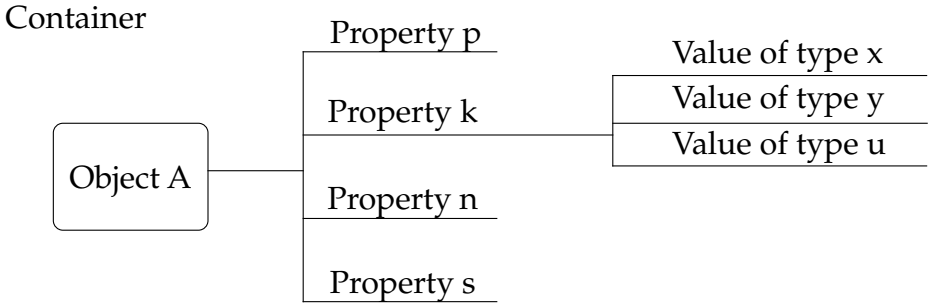
This chapter provides an overview of the Bento design. It describes it more from the API perspective than the format perspective. However, most of the concepts also apply to the format level. In some respects the format is simpler than the API, but it difficult to understand without first understanding the API functionality it is intended to support.

## Bento Data Model

The easiest way to begin understanding the Bento design is probably to review the entities that the API manipulates.

### Primary Entities

The most important entities in the Bento design are containers, objects, properties, values, and types. Their relationship is displayed in the following diagram:




Every object is in some container. An object consists of a set of properties. The properties are not in any particular order. Each property consists of a set of values with distinct types. The values are not in any particular order. Every object must have at least one property, and that property must have at least one value. Each value consists of a variable length sequence of bytes.

Now let us look at these primary entities in more detail.

Containers

All Bento objects are stored in containers. Bento knows very little about a container beyond the objects in it. However, the container always contains a distingushed object, and applications can add arbitrary properties to that object, so applications can specify further information about the container if they wish.

Containers are often files, but they can also be many other forms of storage. For example, in various applications developers already support the following types of containers: blocks of memory, the clipboard, network messages, and Bento values. Undoubtedly other types of containers will be useful as well.

Objects

Each Bento object has a persistent ID which is unique within its container. Other than that, objects don't really exist independent of their properties. An object contains no information beyond what is stored in its properties.

Properties

A property defines a role for a value. Properties are like field names in a record or struct, with two differences. First, properties can be added freely to an object, so an application should never assume an object only has the properties it knows about. Second, property names are globally unique, so that they can never collide when various different applications add properties to the same object. This also means that the same property name always means the same thing, no matter what object it is in. Properties are distinct from types, just as field names are distinct from the data type of the field.

For example, different properties of an object might indicate the name of an object, the author of the object, a comment, a copyright notice, etc These different properties could all have values of the same type: string.

Conversely, a property indicating the date created might have a string, Julian day, or OSI standard date representation. These different formats would not be indicated by the property, but by the type (see below).

Values

Values are where the data is actually stored. The data for a value can be stored anywhere in a container. In fact, it can be broken up into any number of separate pieces, and the pieces can be stored anywhere. (See the discussion of value segments below.)

Each value may range in size from 0 bytes to $2^{64}$ bytes (if you have that much storage). The overhead per value varies depending on the circumstances. For an object with a single value, the typical overhead will be 21 bytes. For a small value which is one of several values associated with a property, the overhead can be as low as five bytes.

Types

The type of a value describes the format of that value. Types record the structure of a value, whether it is compressed, what its byte ordering is, etc. Bento provides an **open-ended** mechanism, so that types can be extended to include whatever metadata is required.

To continue the example above, the type of a string value would indicate the alphabet, whether it was null terminated, and possibly other information (such as the intended language). It might also indicate that the string was stored in a compressed form, and would indicate the compression technique, and the dictionary if one was required. If the string used multi-byte characters, and the byte-ordering was not defined by the alphabet, the type would indicate the byte-ordering within the characters.

Bento defines an inheritance mechanism to make building complex types like this efficient. The structure of types is tied into the mechanism for accessing values, so that the type associated with a value causes the appropriate code to be invoked to access the value, decompress it, byte-swap it, etc. The specific mechanism for doing this is discussed in “Dynamic Values” below.

**Secondary Entities**

There are several additional entities that play supporting roles in the Bento design. These entities are important to fully understand how Bento works, but they do not signficantly change the picture given above.

Type and property descriptions

Each property associated with a value is a reference to a property description. Similarly, the type is a reference to a type description. These type and property descriptions are objects, and their IDs are drawn from the same name-space as other object IDs.

Many type and property descriptions will simply consist of the globally unique name of the type or property. To continue the example above further, the type of a string of 7-bit ASCII, not compressed or otherwise transformed, would simply be described by a globally unique name. This would allow applications to recognize the type.

References to type and property descriptions are distinct from references to ordinary objects in the API to allow language type checking to catch errors in the manipulation of type and property references. However, type and property references can still be passed to the object and value operations, so that value manipulation can be done on types and properties as well as normal objects.

Globally unique names

Globally unique names public or private indentifiers in a format defined by the ISO 9070 standard. They are simply strings written in a subset of 7 bit ASCII. They begin with a name that is assigned by a naming authority designated by ISO (companies can easily register as naming authorities). After this come additional segments, as determined by the naming authority, each of which is unique in the context of the previous segments.

The most common globally unique names will be generated by system vendors or commercial application developers, and may be registered. However, in many cases names will be generated by vertical application developers to record their local types and properties. To meet this need, the naming rules allow for local creation of unregistered unique names, for example by using a product serial number as one of the name segments.

Note that while these names are in 7-bit ASCII, and can easily be printed, they are not designed to be meaningful to end-users. No doubt developers will find reading the specific names useful, but messages to users should be provided in terms of names explicitly designed for user consumption. Such names can easily be provided in type and property descriptions.

IDs

Each object is assigned a persistent ID that is unique within the container in which the object is created. These IDs are never reused once they have been assigned, so even if an object is deleted, its ID will never be reassigned.

These IDs are obviously essential to the functioning of the Bento format, but they do not appear directly in the API. The only points at which an application actually deals with anything corresponding to an ID is when it needs to store an object reference into a value, or find the object corresponding to a reference retrieved from a value. Even in this case, however, the API does not give the application direct access to an object ID, but only to a token that corresponds to the ID in the context of that particular value. This hiding of actual IDs is necessary to allow for reference tracking.

Refnums

In the API types, properties, and objects are referred to using opaque refnums ("magic cookies") provided by the API. The refnums are much more convenient to use than IDs because they are unique within the session, while an ID would need to be used together with a container reference. Since they are opaque, they allow implementations of the API that support caching schemes in which only portions of the container metadata are in memory at any given time.

Refnums have no persistent meaning, so they cannot be stored in values as references to other values. The tokens provided by the reference calls must always be used for persistent references.

Dynamic values

As mentioned above in the discussion on "Types", a Bento value can be compressed, encrypted, byte-swapped, etc. during I/O. Furthermore, these transformations can be composed together.

In addition to data transformation, the same mechanism also supports I/O redirection. In this case a value actually stored in a container is a description of how to find the data, rather than the data itself. Such descriptions can be as simple as references to files, or to objects in another container, or as complex as queries that cause data to be retrieved from a database.

Both I/O transformations and I/O redirection are carried out implicitly by the Bento library, using handlers determined by the type of the value (see Handlers, below). These handlers are attached to temporary entities called dynamic values created by the library. Dynamic values are never visible to the application, and have no persistent meaning.

Value segments

To support interleaving and other uses that require breaking a value up into pieces, Bento allows a value to consist of multiple segments stored at different locations in the container. These segments are not visible at the API, which glues them together to create a single stream of bytes.

The API also takes advantage of value segments to represent insertions, deletions, and overwrites of contiguous bytes in a value. This allows Bento to represent these operations directly in recording updates, rather than having to create a new copy of the value.

**Handlers**

Bento makes use of dynamically linked handlers supplied by the execution environment for two reasons:

- Portability. Use of handlers means that the Bento library is almost trivially portable, since all the system dependencies are in the handlers.
- Extensibility. The Bento library is designed to be easily extended by writing new handlers. The hander interfaces are carefully designed to provide cleanly encapsulated abstractions.

There are three types of handlers:

Session handlers

Certain operations are global to the session as a whole. These include allocating and deallocating memory, and reporting errors. These handlers provide portability, but don't do much for extensibility.

Container handlers

Actual I/O to containers is always done using container handlers, to provide platform independence. The many different types of containers mentioned in the first section are not actually implemented in the Bento library. Instead, the library simply calls different types of handlers, all of which provide the same interface. These handlers map I/O to the underlying storage in a way that depends on the container type. Container handlers basically provide a stream I/O interface to the container storage.

Clearly, container handlers provide portability, by insulating the library from the I/O mechanisms. They also provide extensibility, allowing new types of containers to be defined. Less obviously, they allow extension of the I/O mechanism, such as the addition of transparent buffering without modification of the library. Finally, they support portability between machines with different byte order and even different sizes of bytes.

Value handlers

Both I/O transformations and indirect values are implemented by value handlers, and these handlers are determined by the type of the value. New handlers to carry out new types of data transformations or support new types of indirect values can be written at any time. These handlers are invoked entirely by the library. The accessing application does not need to know that it is using handlers to access the value.

Value handlers are provided mainly for extensibility. However in some cases, such as platform-specific interpretation of references, they also support portability.