

ZNIX Data Store

May 2005



Document Identifier: { ZNIX:ZNIXDS}{1.0} (PDF)
Document Location: <http://znix.sourceforge.net/specs/znixds/>
Errata Location: <http://znix.sourceforge.net/specs/errata/>

Editors:

| | | |
|---------|--------|------------------------|
| Sameera | Perera | University of Moratuwa |
|---------|--------|------------------------|

Contributors:

| | | |
|--|--|--|
| | | |
|--|--|--|

Abstract

This document describes architectural and implementation guidelines for the ZNIX Data Store, in order to provide an open, extendable and consistent storage mechanism for the ZNIX Framework.

The specifications goal is to steer Implementers in a path which is inline with the overall objectives of the Framework, rather than to dictate specific technologies.

Status

This is an initial draft submitted to the interested public for review. Please direct all comments to the message forums which can be found at <http://www.sourceforge.net/znix/>.

Table of Contents

| | | |
|----------|---------------------------------------|----------|
| 1 | INTRODUCTION..... | 2 |
| 2 | NOTATIONS AND TERMINOLOGY..... | 3 |
| 2.1 | Notational Conventions..... | 3 |
| 2.2 | Conventions..... | 3 |
| 2.3 | Acronyms and Abbreviations..... | 4 |

| | | |
|------------|-----------------------------------------------------|-----------|
| 2.4 | Terminology | 4 |
| 2.4.1 | Open | 4 |
| 2.4.2 | Extendable | 4 |
| 2.4.3 | Consistent..... | 4 |
| 2.4.4 | Framework Developer, Developer | 5 |
| 2.4.5 | Implementer | 5 |
| 2.4.6 | Object..... | 5 |
| | | |
| 3 | INTRODUCTION TO ZNIX DATA STORE | 5 |
| | | |
| 3.1 | Project ZNIX | 5 |
| | | |
| 3.2 | ZNIX Data Store | 5 |
| | | |
| 3.3 | Goals of ZNIX DS | 6 |
| 3.3.1 | Open and Extendible Storage Format | 6 |
| 3.3.2 | Simple, efficient Query Mechanism | 6 |
| 3.3.3 | Protection from Unintended Corruption by User | 6 |
| | | |
| 3.4 | Design Analogy | 6 |
| | | |
| 3.5 | Definitions | 7 |
| 3.5.1 | Actor | 7 |
| 3.5.2 | Role..... | 7 |
| 3.5.3 | Agent..... | 7 |
| 3.5.4 | Attribute | 8 |
| 3.5.5 | Resolver | 8 |
| 3.5.6 | Primary and Secondary Fetch Nodes | 8 |
| | | |
| 4 | ACTORS | 8 |
| | | |
| 4.1 | The Actor Configuration File | 9 |
| 4.1.1 | The composition of an ACF..... | 9 |
| | | |
| 5 | ROLES | 10 |
| | | |
| 5.1 | The Role Configuration File | 10 |
| 5.1.1 | The Composition of a RCF | 10 |
| | | |
| 6 | METADATA AND TRIGGER ATTRIBUTES | 11 |
| | | |
| 6.1 | Guarding Attributes | 11 |
| 6.1.1 | Metadata Tags | 12 |
| | | |
| 6.2 | Trigger Attributes | 13 |

1 Introduction

This specification is the result of research conducted by the editors and contributors listed above. The specifications goal is to steer Implementers in a

path which is inline with the overall objectives of the Framework, rather than to dictate specific technologies. An exception to this statement is the use of XML. The specification requires that the information contained in the data store be accessible in XML form (while other interfaces to the same data may exist). However, only the serialization to XML is REQUIRED. The specification is not concerned with how the information is ultimately stored or retrieved within the data store.

2 Notations and Terminology

This section specifies the notations, namespaces, and terminology used in this specification.

2.1 Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

When describing abstract data models, this specification uses the notational convention used by the XML Infoset. Readers are presumed to be familiar with the terms in XML, XPath specifications.

2.2 Conventions

All natural language terms which have specific meaning in the context of this document are expressed with their initial letter capitalized. For example, the term an Actor has a specific meaning in this document, where as an actor need not.

All code segments are displayed in monospaced characters as in the following example:

```
Code Segment Example.
```

Namespace URIs (of the general form "some-URI") represents some application-dependent or context-dependent URI as defined in RFC 2396 [URI].

The XML namespace URIs that MUST be used by implementations of this specification are as follows:

```
http://znix.sourceforge.net/
```

The namespaces used in this document are shown in the following table (note that for brevity, the examples use the prefixes listed below but do not include the URIs – those listed below are assumed).

| Prefix | Namespace |
|--------|-----------------------------|
| znix | http://znix.sourceforge.net |

2.3 Acronyms and Abbreviations

The following (non-normative) table defines acronyms and abbreviations for this document.

| Term | Definition |
|------|------------|
|------|------------|

2.4 Terminology

2.4.1 *Open*

In the context of Project ZNIX, Open or Openness refers to being free of any proprietary technological or licensing requirements, for the modification, reconstruction, or the distribution of the relevant technology.

2.4.2 *Extendable*

Being Extendable, in the context of this specification, refers to a technology's or an implementation's ability to be easily modified, (i.e. without the requirement for a considerable architectural change) to accommodate new sets of information that were unforeseen during the initial design and implementation phases.

2.4.3 *Consistent*

Consistency, refers to the absence of errors or discrepancies that breaks the integrity of the information store in the data store.

2.4.4 Framework Developer, Developer

A Framework Developer (or simply Developer) is a programmer who either writes software to make use of Framework functionality or writes components to extend it.

2.4.5 Implementer

An Implementer is a software programmer whose interest is to provide an implementation for the ZNIX DS, rather than to write software to utilize it.

2.4.6 Object

Main components of the Framework consist of what are known as Roles and Actors. Collectively these are referred to as Objects.

3 Introduction to ZNIX Data Store

3.1 Project ZNIX

Project ZNIX is an attempt to refine traditional information management systems to provide an object-based, application-independent retrieval and manipulation framework targeting *casual users*¹.

3.2 ZNIX Data Store

ZNIX Data Store or ZNIX DS is the core component of the ZNIX Framework. This Open, Extendable and Consistent storage mechanism, is responsible for maintaining the relationships between elements of information that are contained within it much in the same manner as relational database systems, however, without exposing the Framework Developer to the underlying complexity (e.g. foreign keys, tables, views, joins, etc. as in the case of RDBMSs).

¹ A casual user is identified as a person with little training as to the use of computer systems and software, who interacts with a computer purely for entertainment or information management purposes. In this documentation when referred to as users, casual users are implied.

3.3 Goals of ZNIX DS

The Data Store represents a decisive factor in the success or failure of the ZNIX Framework. It is where most of the Framework's objectives materialize. The following considerations are considered to be of paramount importance;

3.3.1 Open and Extendible Storage Format

Project ZNIX argues that traditional software locks down information in proprietary or vendor specific formats, barricading them from collaborative use by other products. In order to break this barrier, the Data Store MUST to be built on top of an extendable and non-proprietary data storage mechanism.

3.3.2 Simple, efficient Query Mechanism

All Implementers should comply with the zPath specification (ZPATH) when providing a query mechanism for the ZNIX DS.

3.3.3 Protection from Unintended Corruption by User

Often the most significant threats to database integrity are the direct modifications made by unsuspecting human users. When left around in plain sight and in human-readable formats, users are tempted to tinker with data often to apply a quick fix or for the sheer pleasure of "out-smarting the system". Such modifications could potentially compromise the integrity of the data store.

3.4 Design Analogy

In an attempt to provide an elegant and a self-explanatory naming scheme, the Core API for the ZNIX Framework was built around an Agent-Role-Actor Analogy.

Often in film industry, an Agent serves as middle-man between directors and Actors. They generally maintain a pool of Actors under contract and are solely responsible for the number of jobs each Actor receives. During a film casting, a director would contact such an Agent and give him the details of the Roles present in the script. The Agent, fully aware of the capabilities and the

experiences of each of his Actors, would pick the one most appropriate for the Role. The director then interviews the candidate and decides whether he is suitable or not. If the latter is the case, the director may continue to ask for another candidate until he is satisfied. In some instances, rather than asking the Agent for a candidate, the director may ask for a specific Actor, whom he himself knows to be the best.

The Core components of the ZNIX Framework too, function in a similar manner. A client process is analogous to a director. When a client needs to query or manipulate some set of information, it will ask the Framework's "Agent" to provide him with an interface to that information. This interface is called an "Actor". Like a real life director, the client may know exactly which Actor is appropriate for the task at hand. For example, if the client wants to store the e-mail address of a person named John, it will ask the Agent for the Actor named "John." Alternatively, the client may ask to the Agent to fetch him any known Actors with the email address similar to john@somecompany.com so that he could figure out which belongs to John.

3.5 Definitions

3.5.1 Actor

Actors are the basic blocks that build up the ZNIX Data Store. They are first class citizens of the ZNIX framework. See Chapter 4 for more details.

3.5.2 Role

Roles define all the characteristics (or Attributes, as they are formally known), that Actors may possess. It is said that an Actor *adopts* a Role.

3.5.3 Agent

Agents are responsible for dispatching Roles and Actors to client processes. Therefore, they are only a part of the Data Store API and are not reflected in the backend.

3.5.4 *Attribute*

An Attribute is an atomic piece of information that an Actor can hold. Atomic means that the Framework itself differs from trying to sub divide the information contained in an Attribute. However, one is always free to come up with his own set of rules to store structured data in an Attribute. This is achieved by providing what are known as Resolvers.

3.5.5 *Resolver*

A Resolver is an object specifying how an Attribute should be interpreted. Two of the most commonly used types of Resolvers are Text Resolvers and AID Resolvers. Text Resolvers are used to store or interpret a textual data item while AID Resolvers are used when an Attribute of one Actor, refers to another Actor.

3.5.6 *Primary and Secondary Fetch Nodes*

Often a client process would require quick access to a Role or an Actor. This requires that the framework maintain an index of such objects. In another scenario, a process would look for an Object or a set of Objects that match a particular criterion. It is quite obvious that each Actor, Role and each of their Attribute values cannot be indexed as it would cost an enormous amount of disk space.

Therefore, this specification describes two indexing strategies called Primary and Secondary Fetch Nodes. Primary Fetch Nodes are indexed, and are used for providing quick access to Objects. Secondary Fetch Nodes on the other hand are never indexed but are retrieved in an ad hoc manner.

4 Actors

Actors are the basic blocks that build up the ZNIX Data Store. They are first class citizens of the ZNIX framework; which simply means that the Identity of an Actor is unique and non-dependant on any other Actor. Each Actor is uniquely identified by an Actor ID or an AID.

4.1 The Actor Configuration File

The Actor MAY be perceived to be self contained within an Actor Configuration File (ACF). Once serialized an ACF, SHOULD be in the following basic format.

4.1.1 The composition of an ACF

```
<znix:Actor>
  <znix:Identity>
    <znix:Id>...</znix:Id>
    <znix:Name>...</znix:Name>
  </znix:Identity>
  <znix:Roles>
    <znix:Role name="...">...</znix:Role>
    ...
  </znix:Roles>
  <znix:Attributes>
    <znix:Attribute name="...">...</znix:Attribute>
    ...
  </znix:Attributes>
</znix:Actor>
```

The following describes the attributes and elements listed above:

4.1.1.1 /znix:Actor

This element encapsulates the information relevant to a single Actor.

4.1.1.2 /*/znix:Identity

This element when applied to any Object encapsulates its Primary Fetch Nodes. The Implementer MAY decide which information should be placed inside this element, rather than the user.

4.1.1.3 /*/znix:Identity/znix:Id

The ID (AID or RID) of the Object.

4.1.1.4 /*/znix:Identity/znix:Name

The primary string that would fetch the current object.

4.1.1.5 /znix:Roles

Encapsulates the list of Roles that the current Object adopts.

4.1.1.6 /znix:Role

Represents an individual Role that the current Object adopts.

4.1.1.7 /znix:Role[@name]

The name of the Role adopted.

4.1.1.8 /znix:Role/

The ID of the Role adopted.

4.1.1.9 /znix:Actor/znix:Attributes

This element encapsulates the list of Attributes the current Actor wishes to expose.

4.1.1.10 /znix:Actor/znixAttributes/znix:Attribute

Holds a single Attribute value for the current Actor.

4.1.1.11 /znix:Actor/znix:Attributes/znix:Attribute[@name]

The name of the Attribute.

4.1.1.12 /znix:Actor/znix:Attributes/znix:Attribute[@roleId]

The ID of the Role that responsible for the Attribute.

5 Roles

Roles define all the characteristics (or Attributes, as they are formally known), that Actors may possess. An Actor adopting a Role, will be able to hold all the types of information understood by that Role. When the Framework encounters an Attribute, it will ask the Role responsible for the Attribute to “explain” how it should be interpreted. Roles are uniquely identified by Role IDs (or RIDs).

5.1 The Role Configuration File

The Role Configuration File or RCF, is similar in structure to an ACF, differing only in that the `znix:Attributes` element now holds a list of Attribute-Resolver pairs rather than Attribute-Value pairs.

5.1.1 The Composition of a RCF

```
<znix:Role>
  <znix:Identity>
    <znix:Id>...</znix:Id>
    <znix:Name>...</znix:Name>
  </znix:Identity>
  <znix:Roles>
    <znix:Role name="...">...</znix:Role>
    ...
  </znix:Roles>
  <znix:AdoptingRoles>
    <znix:Role name="...">...</znix:Role>
    ...
</znix:Role>
```

```
    </znix:AdoptingRoles>
    <znix:Attributes>
      <znix:Attribute resolver="...">...</znix:Attribute>
      ...
    </znix:Attributes>
  </znix:Role>
```

The following describes the attributes and elements listed above:

5.1.1.1 /znix:Role/znix:AdoptingRoles

Contains the list of Roles that adopt the current Role.

5.1.1.2 /znix:Role/znix:Attributes/znix:Attribute

Represents an Attribute that the current Role understands.

5.1.1.3 /znix:Role/znix:Attributes/znix:Attribute/

The name of the Attribute.

5.1.1.4 /znix:Role/znix:Attributes/znix:Attribute[@resolver]

The Resolver for the Attribute.

6 Metadata and Trigger Attributes

6.1 Guarding Attributes

Some client applications may depend on the presence of certain Attributes. Others may require taking special action when certain changes are made to an Attribute.

For example, an Email user-agent might be programmed to fetch an Actor by the sender's email address on an incoming message. It is however, infeasible to allow such queries as, /Person/**/[email='sender@somehost.com'], as such top down queries may potentially require multitude of lookups. Therefore, this specification RECOMMENDS the following work-around.

- When a client depends on the presence of a particular Attribute, it SHOULD determine which Role is best appropriate to hold it. The client MAY interact with the user in making this decision. For instance, in the above example, the email user-agent might decide to place the 'email' Attribute under a top level Role that will be used by all human Actors, such as 'person'.

- The client then SHOULD inform the Framework of this dependency by attaching a metadata component to the relevant Attribute inside the RCF. The value of the metadata component created SHOULD uniquely identify the client process; an obvious choice therefore would be the physical path to the client's executable.
- Similarly, if what a client desires is to be told when a specific action is taken, it SHOULD place relevant metadata components expressing its interest in the Attribute.

This gives rise to the following metadata items.

6.1.1 Metadata Tags

6.1.1.1 Dependant

| | |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Sample Value | C:\Program Files\Outlook Express\msimn.exe |
| Description | Indicates to the Framework that there is a client dependant on the attached Attribute. The Framework SHOULD not remove an Attribute until all Dependant tags are detached; instead it SHOULD insert MarkedForDeletion so that once all Dependants have released the Attribute, removal can proceed. |

6.1.1.2 MarkedForDeletion

| | |
|--------------|----------------------------------------------------------------------------------------------------------------------|
| Sample Value | (Only the presence of the tag is required) |
| Description | Indicates to the Framework that once all Dependant tags are removed, the Attribute too must be removed from the RCF. |

As an example, the above email user-agent would create an Attribute Email under Role, /person, along with a 'Dependant' metadata tag. Say, another client process would make itself a dependant of the Email Attribute, and later decides to release its dependency and tries to remove the Attribute from the Role. The Framework SHOULD, remove the 'Dependant' node with the clients identifier. Then it will add a MarkedForDeletion tag to the Attribute.

However, since the 'Dependant' tag for the user-agent remains, the Attribute is not removed from the Role.

6.2 Trigger Attributes

Several events that occur within the framework MAY be of interest to some client processes. For example, a SMTP user-agent would need to be informed of the creation of a new email message (obviously an Actor), under /email/outbox, so that the message may be put into its outgoing queue.

This requirement is to be fulfilled using what are known as Trigger Attributes. These are similar to normal Attributes differing only in that their values are not handled to user designated Resolvers. Instead, the Framework will look at the Attributes' values in order to find a reference to a client process to be invoked in the face of the relevant event.

The following table lists the set of Trigger Attributes that are recognized by this specification. Note that list renders the Attribute names concerned, as reserved.

| Attribute Name | Description |
|----------------|------------------------------------------------------------------------------------------------------|
| OnCreateRole | The client is notified when a Role adopts the one responsible for this Attribute |
| OnCreateActor | The client is notified when an Actor adopts the Role responsible for this Attribute. |
| OnRoleDetach | The client is notified when a Role removes its adaptation of the one responsible for this Attribute. |
| OnActorDetach | The client is notified when an Actor removes its adaptation of the Role. |