Yggdrasil documentation

Temporal storage of simple objects

Overview of Yggdrasil

Terje Kvernes & David Ranvig

August 14, 2008

Contents

1 Introductions **2**
- 1.1 What is Yggdrasil? . . . 2
- 1.2 Objects . . . 2
  - 1.2.1 Properties . . . 2
- 1.3 Relations . . . 3
- 1.4 Temporal dimension . . . 3
- 1.5 Dynamic storage . . . 3

2 Using Yggdrasil **3**
- 2.1 The basics . . . 3
- 2.2 Initializing Yggdrasil . . . 3
- 2.3 Creating entities . . . 4
- 2.4 Creating properties . . . 4
- 2.5 Creating relations . . . 4
- 2.6 Object operations . . . 6

3 Internals **6**

1 Introductions

1.1 What is Yggdrasil?

Yggdrasil aims to a "dynamic relational temporal object database". In essence, Yggdrasil aims to add two abstractions to the traditional view of a relational database: implicit temporal storage and a simple object model to represent the data stored. In addition to this Yggdrasil allows the relations of the entities stored within to be altered, and new entities and their relations to be added while the system is running. The relations are described by the administrator of the system and as soon as any relation is described to Yggdrasil, it is added to the overall structure of the installation.

In Yggdrasil lingo you can think of an entity as a "class" from the OO world. The "object" is an instance of this entity, each object existing in several temporal versions within the system.

Initializing Yggdrasil has one special and specific feature. You need to give Yggdrasil a namespace to work within. This "namespace" parameter defines which namespace will house the entities you access This is, in essence, your class hierarchy. You'll want to ensure it's uncluttered. The rest of the parameters are all sent to the back end storage layer, and both their meaning and their necessity varies depending on said layer. Look at the documentation for Yggdrasil::Storage and its engines if in doubt.

1.2 Objects

An object is an instance of an entity within the system. This instance is the primary working set that Yggdrasil operates on. Objects contain properties which are key / value pairs. Objects are identified by a unique name within each entity.

An object within Yggdrasil isn't a singular instance of grouped data. As any change to this object is kept, every version of the object throughout its existence is stored. The default object is the "current" object, defined as the set of properties that are currently active and not expired.

The limitations are currently bound to the objects being fairly simple, they are not allowed to store anything more complex than anything that can be mapped into a Yggdrasil type[1], and their only relations to other objects are the ones defined by linking objects together. References will be flattened, so you don't want to store them in Yggdrasil. Using Storable is usually a sign something is wrong, either due to something missing from Yggdrasil or just faulty usage of the library.

1.2.1 Properties

Depending on the way Yggdrasil is set up you may or may note have defined types for your properties, and you may or may not have constraints to the data stored for each property.

Yggdrasil can either be flexible and treat all property values as a default type, or you can select between a set of types Yggdrasil guarantees no matter the back end it's running on. The supported types are listed in the type table.

[1] See table 1 on page 3 for more information on Yggdrasil types

Table 1: Types in Yggdrasil

Name	Definition
TEXT	UTF-8 field of some size
BOOLEAN	A field which only accepts 0 or 1 as its values
VARCHAR(size)	A shorter text field, 1 to 255 units long
SERIAL	An automatically incrementing field
INTEGER	Signed, integers up to 2^16 can be stored
FLOAT	Signed, floats of some size
DATE	A full date and time field, resolution varies

1.3 Relations

To prepare Yggdrasil for relations between objects there needs to be a relation between the entities the objects represent. This is to say one defines between what entities relations exist, then objects are linked together directly.

This link allows Yggdrasil to trace connections within the system, and Yggdrasil will follow any links to any length it needs to find all possible relations between objects. The idea here is that the user is freed from providing anything but direct relations, while Yggdrasil will do the work of building a network from this information.

1.4 Temporal dimension

A principal idea of Yggdrasil is that data is only inserted, never deleted. This also means that there is a clear distinction between "available" and "current" data contained within the system. As long as no time frame, or slice, is requested, all requests work on the "current" dataset.

Changes in the structure will retain the information if possible, the system will inform you at any time if any action you take will permanently delete any data.

1.5 Dynamic storage

New entities and new relations between entities can be issued on the fly while the system is running live.

2 Using Yggdrasil

2.1 The basics

Yggdrasil has a few design rules it tries very hard to stay true to. One of the big ones here is the only objects you'll ever see are those from your namespace. Neither Yggdrasil nor its own classes ever return objects to user space. This has been done to avoid any thoughts about Yggdrasil meta structures having an impact on the data that is being stored. Your objects are your data, always.

2.2 Initializing Yggdrasil

Initializing Yggdrasil is fairly straight forward.

```
new Yggdrasil(
                user      => user,
                password  => password,
                host      => host,
                db        => databasename,
                engine    => engine,
                namespace => 'Ygg',
                mapper    => 'SHA1',
              );
```

Listing 1: Initializing Yggrasil

2.3 Creating entities

When an entity, say "Host", is defined within Yggdrasil, access will be created to the class "Ygg::Host". as we earlier defined the namespace for Yggdrasil to work in previously to be named "Ygg". We'll then create a "Host" object called "nommo" and one called "ninhursaga".

```
$hostclass = define Yggdrasil::Entity 'Host';
my $nommo = $hostclass->new( 'nommo' );
my $ninhursaga = Ygg::Host->new( 'ninhursaga' );
```

Listing 2: Defining entities

The return value from a "define" of "Yggdrasil::Entity" is the class the structure represents. It will always be "Namespace::Entityname", and using the return value lets you rely on Perls warnings and strict pragmas (assuming you use them) in case of a typo.

2.4 Creating properties

Objects are created to store values, and to do this Yggdrasil requires its user to define properties to its entities. After we've defined the property, it is instantly accessible for objects of the class in question.

```
define Ygg::Host 'ip', 'Type' => 'IP';
define $hostclass 'comment';

# Set
$nommo->property( 'ip', '129.240.222.1' );
# Get
$nommo->property( 'ip' );
```

Listing 3: Defining properties

2.5 Creating relations

Relations are a fundamental piece of how Yggdrasil works. Relations create the edges in a network with entities as the nodes. Yggdrasil treats all edges as bidirectional and with identical weight.

It is worth contemplating what kind of a network one is building. In a traditional case of three entities, “Room”, “Host” and “Person”, you can easily form a triangle. Both a “Person” and a “Host” resides in a “Room”, but a “Host” might also belong directly to a “Person”, not to a “Room”.

This means we have a loop in our system, and if we ask Yggdrasil to find what “Host” belongs to a specific person we can get two distinct paths as the answer: Person-¿Room and Person-¿Host-¿Room. Let us look at some code.

```
# Create another entity, "Room".
my $roomclass = define Yggdrasil::Entity 'Room';
my $personclass = define Yggdrasil::Entity 'Person';
# Create a room object, 'b701' and a person object, 'terjekv'.
my $b701 = $roomclass->new( 'B701' );
my $terjekv = Ygg::Person->new( 'terjekv' );

# Now create the relation between rooms and hosts.
# $hostclass = 'Ygg::Host', $roomclass = 'Ygg::Room'
define Yggdrasil::Relation $hostclass, $roomclass;
```

Listing 4: Defining relations

This however doesn't do us much good, we need to link objects together, and we do that as follows:

```
# Now, links the host 'nommo' to the room 'b701'
$nommo->link( $b701 );

# What rooms are nommo related to?
my @hits = $nommo->fetch_related( 'Room' );
# $hits[0] will be $b701.
```

Listing 5: Linking “nommo” to “b701”

```
# Let us create the loop
define Yggdrasil::Relation $hostclass, $personclass;
define Yggdrasil::Relation $room, $personclass;

$nommo->link( $terjekv );
$b701->link( $terjekv );

# Getting nommos room will again return B701.
@hits = $nommo->fetch_related( 'Room' );
```

Listing 6: Creating a loop

Now, a small naughty... Yggdrasil does *not* require that all answers of the same entities to be identical. From the earlier example, we now have two paths between Persons and Rooms: Person-¿Room and Person-¿Host-¿Room. Right now, those paths return the same answer, but we're more than allowed to do something about that.

```
my $b810 = $roomclass->new( 'B810' );

$b701->unlink( $terjekv );
$b810->link( $terjekv );
```

Listing 7: Two paths giving different answers

Oddly enough, we're now in the situation where two paths leading to the same entity / class gives different objects. Person-¿Room gives us "B810" now, but Person-¿Host-¿Room goes terjekv-¿nommo-¿b701 and thus gives us "B701". Which one of these are correct? Well, they both are. Which one you want can be controlled by specifying the maximum distance allowed between the objects.

A future addition to Yggdrasil might allow for constraints requiring all paths leading between entities to give the same answer.

2.6 Object operations

3 Internals