LULO: Manual para el desarrollador

Sobre este manual

En este manual se describe cómo desarrollar software básandose en Lulo, un ORM avanzado con funcionalidad de generación automática de formularios y permisos en IntelliWeb.

La intención de este manual es la de proporcionar una introducción al desarrollo de aplicaciones con Lulo.

Este manual asume que conoces los detalles de desarrollo son IntelliWeb, tales como el concepto de stack, configuración por stack, etc. En algunas ocasiones, se nombrarán a los modelos antiguos y se realizarán comparaciones con éstos. Si no los conoces o no los has usado no importa, basta decir que fueron desarrollados por mí y su código era generado por un generador de código que leía la estructura de la base de datos (justo el enfoque opuesto al que usa Lulo).

Vamos a trabajar todos juntos desarrollando un framework moderno, fácil de usar y útil para todos los ingenieros de intelligenia. Para ello, es importante hacer los cambios de forma ordenada y limpia.

Espero que este manual te ayude a aprender este framework y espero con ilusión tu colaboración en este proyecto.

Introducción

Lulo es un ORM, Object-Relational-Mapper basado en Django, el entorno de trabajo para desarrollar aplicaciones web de forma sencilla.

Lulo no es tan ambicioso como Django, y hace uso de una gran cantidad de código existente, bien en IntelliWeb y en otras librerías externas. Esto es, Lulo proporciona principalmente acceso controlado a base de datos y formularios de creación y edición de objetos.

Además, Lulo proporciona una interfaz de administración análoga a la de Django, que permite el listado de objetos de cada uno de los modelos que tenga definido el desarrollador en el sitio web actual.

El principal objetivo del uso de este entorno de trabajo es el de dotar a los desarrolladores de una herramienta que agilice el desarrollo rápido de aplicaciones sobre la plataforma IntelliWeb de forma similar a como Django las agiliza con respecto al desarrollo de aplicaciones web escritas en Python.

Funcionamiento

El problema

El objeto de Lulo es el de proporcionar la máxima funcionalidad de la manera maś simple y eficiente posible.

Anteriormente, se hacía uso de un generador de código PHP que, a partir de una tabla de la base de datos y un archivo XML con las relaciones entre otros modelos generaba un código PHP extenso con todos los métodos de consulta, actualización, eliminación y obtención de objetos relacionados. Obviamente el 80% del código era prácticamente el mismo entre un modelo y otro y esto hacía que trabajar con estos modelos no fuera eficiente.

Esta aproximación, basada en la generación de código, además, tenía varios problemas.

- En primer lugar, las relaciones eran unidireccionales lo que obligaba a los desarrolladores a generar los XML para todas las clases a priori.
- También suponía que toda clase relacionada con la actual existía, y si no era así, daba un error completo. No había forma fácil de desarrollar primero de forma local a la clase y luego añadirle las relaciones.
- Era fácil romper el generador de código, producir clases sin percatarse del error y después de detectar el error, había que regenerar decenas de archivos de clases.
- El gran tamaño de los archivos (del orden de miles de líneas de código) que hacía difícil su tratamiento por los humanos y por el servidor web.
- Dificultad de modificar la estructura de datos. Como se necesitase modificar la tabla de la base de datos debido a la inclusión de un nuevo atributo, había que volver a regenerar el código.

La solución

La solución pasa por definir las operaciones que pueden utilizar los modelos de forma abstracta y que los modelos las hereden.

¿Cómo se hace esto? Se define de forma estática (con atributos y métodos de clase) todas las propiedades del modelo. Es decir, los atributos, relaciones, la clave primaria y otras propiedades se definen de forma estática y el modelo hereda de forma automática las operaciones de inserción, edición y eliminación.

Otras operaciones como validaciones se implementan como métodos estáticos con determinados nombres que se indican más adelante en este mismo documento.

Las relaciones entre modelos también se agilizan. Antes se generaba un método de consulta, actualización y eliminación para el modelo. Ahora sólo existe una operación de consulta de relaciones (da igual el tipo de ésta) y una consulta de eliminación. Es más sencillo y se mantiene la consistencia, permitiendo a los desarrolladores trabajar de forma más cómoda.

Un aspecto importante es la posibilidad de definir qué clase de conexión de base de datos usar. Esto permite modelizar las tablas de sólo lectura o con base de datos remota.

En definitiva todos los problemas que se detectaron en el generador antiguo se han solucionado:

- Las relaciones ya son bidireccionales.
- Los modelos de una aplicación no son un bloque monolítico. Los modelos son más fácilmente inteligibles por cualquier programador.
- Si se rompe uno de los modelos abstractos, se modifica en un sitio y se arregla en todo. También es más sencillo introducir nueva funcionalidad.
- Los archivos rara vez superan las 300-400 líneas de código (con comentarios amplios).
- Modificar la estructura de datos es muy sencillo. No hace falta regenerar código.

Dependencias software

Intelliweb es un ecosistema software complejo que contiene una gran cantidad de módulos software. Lulo hace uso de varios de ellos

PHP

Lulo requiere de una versión de PHP >= 5.4, supongo que todos los desarrolladores tienen versiones iguales o superiores.

Sistema de Gestión de Base de datos

Lulo está preparado para trabajar con cualquier base de datos para la que haya un conector con la interfaz DBHelper, y de hecho, existen otras interfaces de conexión. Aun así, si no es estrictamente necesario, deberías usar siempre DBHelper.

Módulos concretos

DB

Clase que proporciona una interfaz de comunicación con la base de datos. En algunos sistemas puede tener otros nombres como DB.

Composer

Instalación de los módulos Twig y AdoDB mediante composer.

Configuración necesaria

Configuración de la base de datos

La base de datos no requiere una configuración especial, pero hay varios detalles a tener en cuenta.

Por ahora, **todo modelo Lulo requiere un identificador único autoincrementado llamado id** desde el SGBD, por lo que has de incluir un campo que actúe como clave primaria y que se autoincremente. Tanto el motor de tablas `MyISAM` como `InnoDB` implementan esta característica.

Por supuesto, todo campo con valor predeterminado ha de tener su valor definido en la estructura de la tabla.

Por ejemplo, puedes ver en el modelo `User` que tiene un campo `stack` e `id`, siendo ambos clave primaria y siendo id un campo único autoincrementado. Esa clase además, proporciona un buen ejemplo de uso de Lulo.

Modelos

Antes de empezar a programar, lo primero es tener el diseño de toda la estructura de la base de datos y el diseño de la aplicación (obviamente asumo que el análisis ya está completo).

¿De qué modelo heredar?

Una vez que quede claro qué va a hacer la aplicación y cómo, piensa en la base de datos que vas a usar y las operaciones que necesitas sobre ella, si te sirve solamente con operaciones de lectura o también quieres operaciones de escritura. Aquí tienes el algoritmo a seguir para escoger de qué modelo ha de heredar tu modelo:

- Si vas a usar una BD en sólo lectura, hereda de `ROModel`[1].
- Si vas a usar una BD distinta a IntelliWeb
    - Si vas a leer, hereda de `ROModel`.
    - Si vas a escribir, hereda de `RWModel`
    - Sea cual sea el caso, asegúrate de que la API de ese nuevo `DB` sea la misma que la del `DBHelper` original.
- Si vas a usar la BD de IntelliWeb:
    - Si vas a leer, hereda de `ROModel`.
    - Si vas a escribir, hereda de `LuloModel`.
    - No hace falta sobrescribir la constante `DB` si heredaste de `LuloModel`[2], y sí que hace falta si heredaste de `ROModel`.

La siguiente imagen muestra las relaciones entre los modelos[3]:

Relaciones inversas

Otro aspecto importante es cuestionarse si las relaciones son bidireccionales o no. Normalmente lo son, así tendrás que definir los modelos con los que tiene relaciones en el atributo estático `$RELATED_MODELS` y luego, en el mismo fichero donde está definido tu modelo, pero fuera de la clase deberás incluir el siguiente código:

```
<NOMBRE_MODELO>::init();
```

Donde `<NOMBRE_MODELO>` es el nombre del modelo. Este método estático inicializa las relaciones inversas en función de los modelos indicados en `$RELATED_MODELS` y las relaciones de cada uno de ellos.

Estas relaciones se incluyen de forma automática en el atributo estático `$RELATIONSHIPS`, sólo que quedan marcadas con una propiedad especial (`inverse_of`) que indica de qué relación son inversas.

---

[1] Obviamente independientemente del caso que escojas, tendrás que sobrescribir la constante DB que contiene el nombre de la clase que tiene la conexión con base de datos, poniendo ahí el nombre de la clase que sea.

[2] LuloModel ya tiene `DB = "DB"`

[3] También incluye los modelos de ejemplo User, Tag, Post y Photo.

Es muy importante realizar esta llamada a init, porque si no, no funcionarán las relaciones inversas (de hecho, no existirán).

Ahora, pasamos a describir cada uno de los modelos padre que existen:

ROModel

Heredar de este modelo permite consultar una tabla o una vista. Nótese que no hay funcionalidad de edición o inserción. Sólo hay de lectura (de ahí el nombre *read-only*).

Este modelo es el que define los atributos que serán sobrescritos en los modelos finales. Es decir, **es obligatorio sobrescribir los siguientes atributos de clase y constantes con los valores adecuados**:

Nombre de la tabla

El nombre de la tabla (o vista) se define en la constante `TABLE_NAME` y es una cadena con el nombre de la tabla en base de datos.

Metainformación sobre el modelo

La meta información se almacena en un array estático `$META` en el que se incluyen los siguientes atributos como claves del array:

- `model_description`: una cadena que contiene una descripción textual del modelo.
- `verbose_name`: nombre legible por humanos del modelo (en singular).
- `verbose_name_plural`: nombre legible por humanos del modelo (en plural).
- `gender`: género del modelo. Usado para el cálculo de los artículos en la interfaz de administración.
- `order`: orden a usar en los listados. El formato es el siguiente: `[<ATRIBUTO1>=>"ASC|DESC", <ATRIBUTO2>=>"ASC|DESC", …]`

Atributos de los objetos del modelo

En el array `$ATTRIBUTES` se define la estructura de la tabla que se consultará y sobre la que se insertarán tuplas. Todos los campos de la tabla han de estar presentes aquí, a menos que tengan un valor por defecto y se rellenen de forma automática.

Cada elemento del array representa a un atributo. Cada atributo estará asociado a un campo de la base de datos.

Cada atributo tiene los siguientes metatributos:

- `type`: tipo de dato. Puede tomar cualquiera de los valores siguientes: string, blob, float, int. NOTA: el tipo "blob" identifica a cadenas que representan ficheros.
- `subtype`: tipo semántico. Si no existe, se asume que no hay ninguna restricción. Valores posibles[4]:
  - `phone`: teléfono.
  - `email`: correo electrónico.
  - `ddmmyyyy`: fecha en formato dd/mm/yyyy.

[4] Este es uno de los pocos lugares que dejan lugar a la ampliación en los modelos de Lulo.

    - `ForeignKey`: este atributo actúa como clave externa. En caso de que así sea, necesitará los siguientes atributos:
        - Atributos obligatorios:
            - `name`: nombre de la relación de clave externa.
            - `on`: modelo padre y atributo con el que está asociado. Con el formato "`<Modelo>.<atributo_padre>`".
        - Atributos opcionales:
            - `verbose_name`: nombre legible por humanos de la relación.
            - `related_name`: nombre de la relación inversa.
            - `related_verbose_name`: nombre legible por humanos de la relación inversa.
            - `nullable`: ¿puede ser null?
            - `readonly`: si la relación es de solo lectura
            - `on_maste_deletion`[5]: indica qué hay que hacer en caso de eliminación del padre. Puede tomar los siguiente valores:
                - Establecer un valor constante a una serie de atributos del objeto de la clase actual. Se define como un array: `["set"=>[atributo1=>valor1, ... atributoN=>valorN]]`
                - Poner el atributo a null: `null`.
                - Eliminar el objeto actual: "`delete`".
- `access`: tipo de acceso al atributo. Opcional (se presupone acceso público). Puede tener los siguiente valores:
    - `readonly|ro`: sólo legible desde fuera del ámbito del objeto. Si se intenta editar el atributo, se lanzará una excepción.
    - `rw`: editable desde fuera del ámbito del objeto.
    - `writable`: igual que rw.
- `default`: valor por defecto. Opcional.
- `null`: indica si el atributo puede tener el valor nulo. Opcional.
- `verbose_name`: descripción del atributo. Obligatorio.
- `length`: si el campo es una cadena, se puede incluir la longitud de éste. Opcional.
- `auto`: el campo se rellena de forma automática. Bien porque tiene un valor por defecto en BD o por otro motivo.

Todo atributo que no tenga un valor por defecto o no pueda ser nulo, se asume obligatorio.

Notemos que aquí los atributos pueden llamarse como el desarrollador quiera, no se tienen que respetar las convenciones de `_stack` sea el atributo del stack y el id sea el identificador único. Eso lo definiremos en la aplicación que haga uso de este modelo.

Ascensión de atributos

Un concepto clave cuando hablamos de atributos de un modelo, es el de ascensión de atributos.

Todo atributo que tenga un `subtype`, puede ser devuelto como (ascendido a) objeto.

[5] Actualmente, esta opción no está implementada.

Cómo se hace esto, mediante la llamada al método `a`, pasándole a este método el nombre del atributo.

Ahora mismo sólo está implementado para los subtipos de fechas, que devolverán un objeto de clase `Datetime` de PHP.

Clave primaria

La clave primaria se define como un array en el atributo estático `$PK_ATTRIBUTES`. Este array contiene los nombres de los atributos que forman la clave primaria.

Modelos relacionados

Es obligatorio que todo modelo sepa qué relaciones tiene con los otros modelos. Este array contiene los nombres de las clases que hacen referencia a él o a los que él hace referencia[6].

Como él se conoce a sí mismo, no hace falta que incluya su nombre en el array.

El atributo estático se llama `$RELATED_MODELS`.

Los modelos pueden no sobrescribir este atributo, pero entonces sólo existirán relaciones unidireccionales. Si se quieren relaciones bidireccionales se ha de indicar con qué otros modelos tiene relación.

Relaciones

Las relaciones se definen como arrays de arrays en el atributo de clase `$RELATIONSHIPS`. Toda relación ha de ser de uno de los siguientes tipos:

- **ForeignKey**: relación de clave externa. O lo que es igual Muchos-a-1. Descritas como metaatributos de atributo, también se pueden incluir en la zona de relaciones (`$RELATIONSHIPS`).
- **ManyToMany**: relación de muchos a muchos. Requiere de al menos una tabla intermedia.
- **OneToMany**: relación de maestro-esclavo. O lo que es igual 1-a-Muchos. Se generarán de forma automática, no se te ocurra crear una relación de este tipo.

Después del tipo, se ha de incluir el modelo. Es importante destacar que **el modelo ha de incluirse como una cadena que contenga el nombre del modelo** y no como una llamada al atributo CLASS_NAME del modelo, ¿Por qué? Porque si escribimos <Modelo>::CLASS_NAME, el cargador de clases intentará cargar <Modelo> y se producirá una dependencia cíclica que, si bien no da un error, no permite cargar correctamente las clases. Este es el precio que hay que pagar por la inclusión de la carga dinámica de clases.

```
// Si es relación de muchos a muchos

'<relationship_name>' => [

    // Tipo de la relación (muchos a muchos)
```

[6] Se puede ver un ejemplo de esto en `User` y `Photo`.

```
"type" => "ManyToMany",
// Modelo con el que está relacionado
"model" => "<model_name>",
// Tablas de nexo.
// ATENCIÓN: sólo se permite la edición si existe una única tabla nexo
"junctions" => <array con la lista de las tablas intermedias>,
// Lista con la lista de todas las relaciones entre nexos
"conditions" => [
  // Cada una de las relaciones entre tablas
  // Se asume que
  // 1.- La tabla 1 (T1) es la tabla origen (la tabla de este modelo)
  // 2.- Las tablas 2 hasta la N-1 (ambas incluidas) son las tablas nexo
   // (tablas listadas en el atributo "junctions" de esta relación)
   // 3.- La tabla N es la tabla remota
   // (tabla del modelo descrito en el atributo "model" de esta relación)
  // Relaciones:
  // Relación entre tabla 1 (T1) y tabla 2 (T2)
   // (M1 es el número de atributos a emparejar entre la T1 y la T2)
  [<T1_attr1> => <T2_attr1>, ..., <T1_attrM1> => <T2_attrM1>],
  // Relación entre tabla 2 (T2) y tabla 3 (T3)
   // (M2 es el número de atributos a emparejar entre la T2 y la T3)
  [<T2_attr1> => <T3_attr1>, ..., <T2_attrM2> => <T3_attrM2>],
  ...
  // Relación entra tabla N-1 T[N-1] y tabla N (TN)
   // (Mn es el número de atributos a emparejar entre la T[N-1] y la TN)
  [<T[N-1]_attr1> => <TN_attr1>, ..., <T[N-1]_attr_Mn> => <TN_attrMn>],
],
// Para determinar si la relación es de sólo lectura
"readonly" => true|false,
// Para determinar si se han de incluir los atributos de la relación
// como atributos dinámicos de los objetos relacionados
```

```
        // los atributos han de tener un nombre distinto a los atributos propios
        // del objeto remoto. Si comparten nombre, no se incluirán los del nexo sino
        // los del objeto remoto
        "include_nexii_attributes" => true|false,
    ],

    // Si es relación de clave externa con otra modelo (tabla)
    '<relationship_name>' => [
        // Tipo de la relación (clave externa)
        'type' => "ForeignKey",
        // Nombre del modelo con el que se relaciona
        'model' => "<model_name>",
        // Condición de la relación, donde los atributos locales
        // son los del modelo actual, y los remotos los del modelo remoto
        "condition" => [
          // Relaciones entre atributos de este modelo y los del modelo remoto
          <local_attribute_1> => <remote_attribute_1>,
          <local_attribute_2> => <remote_attribute_2>,
          ...
          <local_attribute_N> => <remote_attribute_N>,
        ],
        // Para determinar si la relación es de sólo lectura
        "readonly" => true|false,
    ],
```

Fíjate que no hemos incluido las relaciones OneToMany. Esto se debe a que aunque existen, se insertarán de forma automática como relaciones inversas de las ForeignKey. Es decir, sólo se ha de especificar la relación en el lado “muchos” (salvo en el caso ManyToMany, que obviamente, da igual el extremo en el que la especifiquemos).

Validación del nuevo objeto

Ante la creación de un objeto, podemos querer validar si sus datos cumplen una condición. Para ello, se sobrescribe el método `cleanObjectAttributes`. Este método tomará un array con los datos y devolverá un array con los datos convertidos. Si falta algún dato, ha de lanzar una excepción.

Este método `cleanObjectAttributes` será llamado en cada llamada al constructor (a menos que se indique lo contrario en su segundo parámetro).

Hemos de tener en cuenta que sólo se llama en la construcción de objetos nuevos, no al cargar objetos de base de datos. Esto es, si los datos en base de datos son incorrectos, no se comprobarán.

LuloQuery

De forma similar a como hace Django con el QuerySet, se ha desarrollado un objeto que encapsula las consultas llamado LuloQuery.

Este sistema de generación de consultas está explicado en otro documento llamado LuloQuery en este mismo directorio. Lo que viene a continuación no es más que una somera descripción de la funcionalidad de este subsistema.

Se le llama mediante el método objects, que devuelve un LuloQuery al que se le pueden añadir condiciones, límites y un orden al resultado de la consulta. Es importante destacar que los métodos de filtrado, exclusión, límite y orden devuelven siempre una referencia al objeto `$this`, permitiendo encadenar las operaciones.

Relaciones

A los modelos ROModel se les añade un método dinámico con el nombre de la relación que devuelve un objeto LuloQuery, de manera que se pueden realizar las mismas operaciones de consulta sabiendo automáticamente que se traerá los atributos relacionados.

Además, permite que se le pase parámetros que se tomarán como parámetros de un filter, por lo que se compacta todavía más la llamada a los objetos relacionados.

Es importante destacar que esta funcionalidad funciona de igual forma para las relaciones ForeignKey y ManyToMany, además de sus relaciones inversas correspondientes.

Eso sí, no se permite la adición (todavía) de objetos a la relación haciendo uso del método add (se hará en un futuro).

Ejemplos

```
// Obtención de la etiqueta padre de una etiqueta

$parentTag = $tag->parent_tag()->get();

// Obtención de las etiquetas (que continen la palabra "amigos")

// relacionadas con un usuario

$relatedTags = $user->tags(["name__contains"=>"amigos"]);
```

Hay más ejemplos en los tests, por lo que no te preocupes si no entiendes al 100% esta parte.

Por supuesto, estos métodos mágicos no suplen a los ya conocidos dbLoadRelated, sino que los complementan y hacen el desarrollo de software más sencillo (o eso espero).

Métodos

- `order`: permite aplicar un orden a la consulta, acepta un array de (clave=>valor) donde la clave es el campo por el que se ordena y el valor el tipo de orden (`ASC` o `DESC`). La clave puede ser un campo de la tabla o un campo de una tabla relacionada con la sintaxis `<nombre relación>::<campo modelo relacionado>`.
- `limit`: limita el número de objetos obtenidos. Acepta un parámetro (número de elementos obtenidos) o dos (inicio y número de elemento obtenidos desde inicio).
- `filter`: filtra por condiciones. Acepta como tantos parámetros como sea necesario, siendo cada uno de ellos un array. Las condiciones que se pasan en arrays se unen mediante AND. Los arrays que se ponen como otros parámetros se unen mediante OR. Ejemplo:
  - `filter(["A"=>1, "B"=>2], ["B"=>3, "C"=>4])`:
    - (A=1 AND B=2) OR (B=3 AND C=4)
  - `filter(["A"=>1], ["B"=>3, "C"=>4])->`
    `filter(["X"=>2, "Y"=>3], ["W"=>5])`:
    - ((A=1) OR (B=3 AND C=4)) AND ((X=2 AND Y=3) OR W=5)
- `exclude`: condiciones de exclusión de la consulta a realizar. Tienen la misma estructura que que los parámetros de filter, pero habrá que tener en cuenta que la condición se niega con un NOT.
  - `filter(["A"=>1, "B"=>2], ["B"=>3, "C"=>4])->`
    `exclude(["X"=>5, "Y"=>6])`:
    - ((A=1 AND B=2) OR (B=3 AND C=4)) AND NOT(X=5 AND Y=6)

Campos de búsqueda

Uno de los aspectos interesantes de LuloQuery es su posibilidad de realizar operaciones de consulta basándose en operadores especiales y campos externos. La sintaxis de cada campo es la siguiente:

Campo interno

- `<campo> => <valor>`: operación de igualdad.
- `<campo>__<operador> => <valor>`: operación específica.

Donde `<campo>` es el nombre de un campo en la base de datos, `<valor>` es un valor legal para ese campo de la tabla y si hay un operador, `<operador>` identifica de forma única la operación de comparación a realizar.

Campo externo (campo de un modelo relacionado con el actual)

- `<nombre relación>::<campo modelo relacionado> => valor`
- `<nombre relación>::<campo modelo relacionado>__<operador> => valor`

De forma similar al caso de un campo interno, tenemos dos tipos de comparaciones. En la primera el operador es el de igualdad, en la segunda éste puede especificarse. En ambos tipos de comparaciones aparece el nombre de la relación que es el nombre de la relación

desde el punto de vista del modelo que estamos cargando[7] y el nombre del campo del modelo destino[8].

Operadores

Hay varios operadores de búsqueda:

- `contains`: indica si el valor indicado está incluido en el valor del campo de la tabla.
- `notcontains`: lo contrario de contains.
- `startswith`: indica si el valor forma parte del inicio del valor del campo.
- `endswith`: indica si el valor forma parte del final del valor del campo.
- `in`: indica si el campo de la tabla es uno de los valores en un array. Es decir, el valor que aparece a la derecha de la asignación (clave=>valor) es un array.
- `range`: indica si un valor se encuentra en un intervalo. De forma similar a lo que ocurre con in, el valor del hash es un array de dos elementos.
- `eq`: operador de igualación.
- `noteq`: operador distinto.
- `lt`: : operador menor que.
- `lte`: : operador menor o igual que.
- `gt`: operdor mayor que.
- `gte`: operador mayor o igual que.

Ejemplos

```
// Obtiene todos los usuarios con el username "diegoj"
$users = User::objects()->
filter(["username"=>"diegoj"])->
limit(5)->
order(["username"=>"asc"]);

// Obtiene todos los usuarios creados antes de hoy menos
// el del username "diegoj"
$today = new DateTime('NOW')
$users = User::objects()->
filter(["creation_datetime__lt"=>$today])->
exclude(["username"=>"diegoj"])->
limit(5)->
order(["username"=>"asc"]);
```

Programación por contrato

Los ROModel permiten programación por contrato a nivel de objeto. Cuando se construye un objeto, el constructor admite dos parámetros:

- atributos del objeto.
- booleano que indica si se ha de aplicar este validación (por defecto activada).

[7] Es decir, que si es una relación directa (Modelo->ModeloDestino) será la clave en el array `$RELATIONSHIPS`, en caso de que sea una relación inversa (Modelo<-ModeloDestino), será el valor de `related_name` incluido en la relación desde el punto de vista del ModeloDestino.

[8] Aquí no hay confusión posible, es el nombre del campo de la tabla destino.

Para ejecutar la validación del objeto, no hay más que sobrescribir el método `cleanObjectAttributes`.

Este método ha de devolver una excepción por cada error que encuentre en la creación del objeto. Obviamente también se pueden implementar conversiones de datos, comprobaciones avanzadas e incluso comprobaciones no sólo con el objeto actual, sino con el objeto a crear y el resto de objetos que existen en el sistema.

Nótese que este método **no se le llama en la carga de objetos** de la base de datos por motivos de eficiencia. Así, si hay objetos guardados en base de datos que no cumplen el contrato, no darán un error.

RWModel

Este modelo hereda de `ROModel` y permite (si se hereda de él) realizar operaciones de inserción, actualización y eliminación.

Este modelo es el padre de `LuloModel`.

Métodos sobrescribibles

cleanCreation

Proporciona una conversión entre los datos obtenidos en crudo del formulario de creación de un nuevo objeto y los datos que tomará el método `factoryFromArray` del modelo.

Si se sobrescribe este método, se ha de llamar a `cleanCreation` de la clase padre si se desea que las relaciones de clave externa se actualicen. En todas las clases de ejemplo puedes ver cómo se usa.

Este método será llamado de forma automática por la clase de ayuda de creación de formularios `ModelForm`.

cleanEdition

Proporciona una conversión entre los datos obtenidos en crudo en el formulario de edición del objeto y los datos que tomará el método `setFromArray`.

Si se sobrescribe este método, se ha de llamar a `cleanEdition` de la clase padre si se desea que las relaciones de clave externa se actualicen. En todas las clases de ejemplo puedes ver cómo se usa.

Este método será llamado de forma automática por la clase de ayuda de creación de formularios `ModelForm`.

defaultFormValue

Este método devuelve el valor por defecto de un campo para el formulario de creación o de edición creado por `ModelForm`, (en función de si no le pasamos un objeto o sí se lo pasamos, respectivamente). Es útil para definir valores por defecto en las relaciones.

Mira `Tag::defaultFormValue` por ejemplo para saber de qué estamos hablando.

formValues

Valores de los campos que son de tipo select y multiselect del formulario de creación y edición para este modelo. Al igual que en `defaultFormValue`, se usa el parámetro `$object` para distinguir si estamos en un formulario de creación (si `$object` es nulo) o en un formulario de edición (si `$object` es un objeto de ese modelo).

Mira `Tag::defaultFormValue` por ejemplo para saber cómo se asignan los posibles valores en relaciones.

formValidation

Este método tiene las llamadas a los métodos de validación de cada uno. Se devuelve un array en el que cada elemento es un array con la estructura de las validaciones de WizardForm:

```
$validatorArray = [

  "function" => <Nombre del método estático público validador>,

  "error_message" => "<mensaje de error si la validación no se cumple>",

  "error_fields" => <array con una lista de los campos erróneos>

];
```

Es importante destacar que si `$object` no es nulo se le pasa a este método ANTES de habérsele asignado a `$object` los valores que vienen del formulario.

El método `User::formValidation` devuelve un conjunto de validadores para el formulario. Échale un vistazo.

clean

Método de objeto para validar un objeto. Se pasa como parámetro un array con los objetos `DBBlobReader`.

Si no se desea comprobar si el objeto existe en base de datos, se pueden utilizar los métodos de objeto:

- `cleanNew`: validar y limpiar un objeto nuevo.
- `cleanOld`: validar y limpiar un objeto que ya existía en la base de datos.

Se considerará que el objeto es correcto si no se lanza una excepción.

LuloModel

`LuloModel` hereda de `RWModel`, por lo que es hijo de éste y nieto de `ROModel`.

Este modelo fuerza a trabajar sobre la base de datos de IntelliWeb[9]. De ahí que sobrescriba la constante `DBHelper` con el valor "`DBHelper`" (nótese que es una cadena con el texto DBHelper).

Esta clase contiene los métodos de tratamiento con `dbBlobReader` y `dbBlobWriter`.

dbBlobUpdate

Este método escribe un blob cuyo nombre de atributo es el primer parámetro (`$blobName`), el segundo parámetro (`$blob`) es el propio blob, que puede ser:

1. Un objeto `DBBlobReader`.
2. Una cadena, que es el binario del fichero.
3. Un array, y tiene la clave "`path`" que es una cadena con la ruta de un fichero.
4. Un descriptor de fichero, cuyo contenido se volcará en el campo `$blobName`.

dbBlobLoad

Carga el blob con el nombre del primer parámetro (`$blobName`), que está en esta tabla como un LONGBLOB, en un objeto de tipo `DBBLOBReader`.

[9] Teóricamente se podría reimplementar `DBBlobReader` y `DBBlobWriter` para su acceso a otras bases de datos, pero habría que pasarles un nuevo conector, cambiar el estilo de SQL, etc.

Ejemplo

Este ejemplo se basa en una supuesta red social en la que se etiquetan a los usuarios y se suben fotos. No hay funcionalidad para comentar y no hay permisos. Obviamente es un ejemplo y sólo mostramos las vistas de administración. Ten en cuenta que estas vistas de administración se generan de forma automática para todo modelo `LuloModel`.

Tablas

Hay 5 tablas, una para cada entidad y una tabla de nexo. Recordemos que habrá 4 entidades usuarios, fotografías de usuarios, entradas de blog de usuarios y etiquetas de usuarios.

Fíjate como los nombres de las tablas son los que el desarrollador desee y no tienen porqué tener relación con los nombres de los modelos generados.

```
-- --------------------------------------------------------

--

-- Estructura de tabla para la tabla `user`

--

CREATE TABLE IF NOT EXISTS `user` (

  `stack` varchar(256) CHARACTER SET ascii NOT NULL,

  `id` int(11) NOT NULL AUTO_INCREMENT,

  `first_name` varchar(256) COLLATE utf8_spanish_ci NOT NULL,

  `last_name` varchar(256) COLLATE utf8_spanish_ci NOT NULL,

  `email` varchar(256) COLLATE utf8_spanish_ci NOT NULL,

  `phone` varchar(256) COLLATE utf8_spanish_ci DEFAULT NULL,

  `username` varchar(64) COLLATE utf8_spanish_ci NOT NULL,

  `sha1_password` varchar(256) COLLATE utf8_spanish_ci NOT NULL,

  `main_photo` longblob NOT NULL,

  `main_photo_mimetype` varchar(64) COLLATE utf8_spanish_ci NOT NULL,

  `main_photo_filename` varchar(64) COLLATE utf8_spanish_ci NOT NULL,

  `last_update_datetime` datetime NOT NULL,

  `creation_datetime` datetime NOT NULL,

  PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci COMMENT='Tabla de ejemplo para LULO' AUTO_INCREMENT=16 ;
```

```
-- --------------------------------------------------------

--

-- Estructura de tabla para la tabla `photo`

--

CREATE TABLE IF NOT EXISTS `photo` (

  `stack` varchar(128) CHARACTER SET ascii NOT NULL,

  `id` int(11) NOT NULL AUTO_INCREMENT,

  `user_id` int(11) DEFAULT NULL,

  `order_in_gallery` int(11) NOT NULL,

  `photo` longblob NOT NULL,

  `photo_mimetype` varchar(128) COLLATE utf8_unicode_ci NOT NULL,

  `photo_filename` varchar(128) COLLATE utf8_unicode_ci NOT NULL,

  `creation_datetime` datetime NOT NULL,

  `last_update_datetime` datetime NOT NULL,

  PRIMARY KEY (`id`)

) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci COMMENT='Imágenes de
los usuarios' AUTO_INCREMENT=5 ;


-- --------------------------------------------------------

--

-- Estructura de tabla para la tabla `user_tag`

--

CREATE TABLE IF NOT EXISTS `user_tag` (

  `stack` varchar(128) CHARACTER SET ascii NOT NULL,

  `id` int(11) NOT NULL AUTO_INCREMENT,

  `name` varchar(128) COLLATE utf8_unicode_ci NOT NULL,

  `description` longtext COLLATE utf8_unicode_ci NOT NULL,

  `parent_id` int(11) DEFAULT NULL,

  `creation_datetime` datetime NOT NULL,

  `last_update_datetime` datetime NOT NULL,
```

```
  PRIMARY KEY (`id`),
  KEY `id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci COMMENT='Etiquetas para cada uno de los usuarios' AUTO_INCREMENT=38 ;

-- --------------------------------------------------------
--
-- Estructura de tabla para la tabla `tag`
--
CREATE TABLE IF NOT EXISTS `tag` (
  `stack` varchar(256) CHARACTER SET ascii NOT NULL,
  `user_id` int(11) NOT NULL,
  `usertag_id` int(11) NOT NULL,
  PRIMARY KEY (`stack`,`user_id`,`usertag_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci COMMENT='Relación muchos a muchos entre user y usertag';

-- --------------------------------------------------------
--
-- Estructura de tabla para la tabla `post`
--
CREATE TABLE `post` (
        `stack` varchar(128) CHARACTER SET ascii NOT NULL,
        `id` int(11) NOT NULL AUTO_INCREMENT,
        `owner_id` int(11) NOT NULL,
        `title` varchar(128) COLLATE utf8_unicode_ci NOT NULL,
        `title_slug` varchar(128) CHARACTER SET ascii NOT NULL,
        `content` longtext COLLATE utf8_unicode_ci NOT NULL,
        `creation_datetime` datetime NOT NULL,
        `last_update_datetime` datetime NOT NULL,
        PRIMARY KEY (`id`)
```

```
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci
COMMENT='Entradas'
```

Modelos

Por motivos de espacio no se ha incluido el código de cada uno de los modelos, pero puedes encontrarlo en `/lulo/tests/models`. En esa carpeta verás varias clases. Esas clases implementa los modelos que se describen a continuación.

User

Este modelo define un supuesto usuario de un sistema. Este usuario tendrá fotografías y estará etiquetado por una serie de etiquetas que lo clasificarán en función a las preferencias de un supuesto administrador

Photo

Fotografía de perfil de cada uno de los usuarios. Dependiente del usuario, un usuario puede tener muchas fotografías.

Tag

Este modelo define una etiqueta que se le puede aplicar a los usuarios. Todo usuario puede tener varias etiquetas y cada una de éstas puede aplicarse a varios usuarios.

Post

Este modelo supone que el usuario tiene un blog asociado y que cada entrada se almacena en este modelo. Los atributos `title` y `content` representan al título y contenido de la entrada de blog (resp.) y ambos son traducibles.

Preguntas más frecuentes

En esta sección se incluyen las preguntas más frecuentes agrupadas por secciones. Esta sección se irá ampliando conforme los desarrolladores vayan utilizando este sistema.

Autoría, nomenclatura y generalidades

¿Quién es el autor de esto?

El desarrollo y la idea original son míos (Diego J. Romero López), y, obviamente me he basado en todo el desarrollo anterior realizado por todo el equipo de intelligenia en el desarrollo de IntelliWeb.

Atributos especiales

¿Por qué no insertas atributos de forma automática como _stack, _uuid, etc?

No soy amigo de hacer un sistema tan dependiente de este tipo de características que no permita el uso de estos modelos sobre otras vistas o tablas ya existentes, como de hecho hice en el proyecto de la API para la UGR.

¿Podrías al menos forzar a trabajar sólo con objetos del stack actual?

Esa funcionalidad existe, se puede implementar el método `implicitBaseCondition` y devolver un array con la asignación de las condiciones base en toda obtención de objetos de la base de datos. Por ejemplo, esta implementación en una clase hija de `LuloModel` con el atributo `stack`:

```
public static function implicitBaseCondition(){

  return [ "stack" => EWSTACK, ];

}
```

fuerza a que en los métodos de carga, edición y eliminación (`dbLoad`, `dbLoadByPk`, `dbLoadAll`, `dbDeleteAll`) y derivados se carguen siempre objetos del stack actual (debido a que `EWSTACK` se define de forma automática en la configuración en cada stack con el identificador del stack en cuestión).

¿Dónde están los atributos de fecha de creación y de fecha de actualización?

En ningún sitio. Si quieres, defínete un `LuloModel` genérico y úsalo en tus proyectos sobreescribiendo los métodos clean y de validación si lo ves necesario. Todas las clases de ejemplo `User`, `Tag` y `Photo` tienen fechas de creación y de última actualización y no me he visto en ningún poblema para asignarle valores. Échales un vistazo para ver cómo se pueden usar.

¿Se pueden realizar borrados lógicos?

Sí, para explicarlo, vamos a suponer que vas a usar el atributo `is_erased` para indicar si el objetos está borrado lógicamente (`1`) o no (`0`).

Lo primero es indicar en implicitBaseCondition que sólo queremos trabajar sobre objetos con `is_erased=0`, por tanto:

```
public static function implicitBaseCondition(){

    return [ "stack" => EWSTACK, "is_erased"=>0  ];

}
```

Puede ser una buena implementación de ese método.

Después, hemos de sobrescribir el método protegido `dbDeleteAction`. Este método contiene la acción de eliminación de un objeto. Por defecto, consiste en una eliminación real de la tupla en la base de datos. Nosotros tendremos que sobrescribir su implementación en nuestro modelo, de manera que actualice el atributo `is_erased`, poniéndolo a `0`.

Por último, hemos de indicar que las relaciones entre objetos no han de eliminarse. Para ello hemos de sobrescribir la constante de clase que indica este hecho `UPDATE_RELATIONS_ON_OBJECT_DELETION_BY_DEFAULT = false`. Así, cuando se produzca una llamada a `dbDelete`, no se eliminarán las relaciones sino que se marcará el objeto actual como borrado.

Desarrollo

He visto que el framework X hace Y, ¿por qué no lo hace Lulo?

Puede que no lo conozca, o puede simplemente que esa filosofía no encaje con la filosofía de Lulo. Lo podemos hablar.

¿Son ampliables estas clases?

Sí, pero **no de forma arbitraria**. La complejidad de estas clases y la posibilidad de equivocarse es tal que pido a los desarrolladores que antes de modificar cualquier parte de esta librería me lo comuniquen. Ten en cuenta que un cambio en `ROModel`, `RWModel` o `LuloModel` rompería todas las aplicaciones que hagan uso de modelos Lulo, por lo tanto, todo cambio deberá ser aprobado por Diego J. antes de publicarse. **Recuerda, Lulo es un esfuerzo colaborativo, pero ordenado.**

¿Puedo desarrollar una librería basada en este miniframework?

Por supuesto, si en tus aplicaciones utilizas una serie de convenciones y quieres seguir repitiéndolas, la mejor forma es que heredes del modelo que te interese y lo uses como modelo padre del resto de tus modelos.

Por ejemplo, supongamos que todos tus modelos tienen los siguientes atributos:

- `stack`: nombre del *stack* (cadena). Constante según la aplicación.
- `id`: identificador único del objeto (entero autoincrementado). Único por tabla (modelo).
- `last_update_datetime`: fecha de última actualización. `Datetime`.
- `creation_datetime`: fecha de creación. `Datetime`.

No podemos definir los atributos en el modelo padre y luego ampliarlo, pero sí podemos modificar los métodos `cleanCreation`, `cleanEdition`, `dbSave`, etc. Según las convenciones que estamos adoptando en nuestra aplicación.

Mi proyecto requiere que trabaje con otra base de datos distinta, ¿cómo lo hago?

Los modelos de Lulo tienen un atributo llamado DBHelper que es sobrescribible por todos los modelos que hereden de `ROModel`, `RWModel` y `LuloModel`. Sólo tienes que tener una capa de abstracción de base de datos con la misma API, el resto funciona exactamente igual.

Documentación

¿Dónde puedo ver la API de estas clases?

Hay un documento PDF en el que está la API, extraída por medio de `phpdocumentor`. Puedes acceder al documento en este mismo directorio compartido.

Operaciones

¿Cómo se trabaja con los blobs?

Hay métodos que permiten la actualización de blobs desde varias fuentes, mira la documentación de `dbBlobUpdate`, `dbBlobLoad` y `dbSave`. Si tienes más dudas, puedes ver cómo trabaja `LuloAdmin` con los blobs.

¿Dónde está el método dbEdit?

Ya no existe. En los modelos `LuloModel` sólo hay un método `dbSave` que será el encargado de guardar o actualizar el objeto. Si el objeto no existe en la tabla, insertará una nueva tupla en la tabla y si existe, actualizará sus datos.

¿Hay operaciones de conversión de datos?

No, eso no pertenece al modelo y no debería pertenecer. Lamentablemente PHP no tiene métodos de cadenas ni de enteros (espero que algún día los añadan). No se van a insertar modificadores de tipo como atributos de los modelos.

Relaciones

¿Son las relaciones unidireccionales?

No. Las relaciones son bidireccionales. Están las relaciones directas que son las que han sido definidas por el desarrollador y las relaciones inversas que son las definidas de forma automática por el sistema. Para que se creen las relaciones inversas **se ha de llamar al método estático del modelo `init` justo después de la definición de la clase**. Mira los ejemplos para más información.

Es importante destacar que sólo se mostrarán las relaciones directas en los formularios automáticos.

¿Hay relaciones de modelos a tablas?

Están en proyecto. Esto permitirá la obtención de atributos de tablas de nexo o de vistas que no queremos convertir en modelo.