LuloQuery: una herramienta poderosa para Lulo

Introducción

¿Cansado de leer `dbLoadAll`? ¿Harto del formato de consultas de `DB`? Con LuloQuery podrás generar consultas sencillas que eviten ese formato engorroso y con cierto “aroma” a Django,

Este documento describe una de las funcionalidades más potentes de Lulo, su herramienta de generación de consultas LuloQuery. Vamos a ver de qué partes está compuesto este documento que echará luz sobre este asunto.

Primero describiremos el objetivo de este trabajo, ¿para qué tener dos sistemas de consultas distintos?

Después describiremos su uso para que puedas directamente comenzar a trabajar en tus proyectos con LuloQuery.

Luego hablaremos del diseño de LuloQuery, de qué partes internas está compuesto y las decisiones que se han tomado en su construcción. Esta explicación nos servirá para poder introducir más tarde la siguiente sección: cómo extender LuloQuery.

Justo después de explicar cómo extender LuloQuery mostraremos posibles mejoras futuras que se contemplan como ampliaciones en un futuro.

Por último, terminaremos con un epílogo en el que pondremos punto final a este documento.

Antes de que comiences a leer este manual, quiero que te pares un momento y pienses que este sistema no es perfecto, está diseñado para ser lo más útil posible pero seguramente tenga sus fallos, por lo que habrá que ir puliéndolo con el tiempo, y de hecho, ése es mi objetivo, que entre todos mejoremos este sistema con el objetivo de mejorar el rendimiento de nuestras aplicaciones web, la eficiencia y usabilidad en el desarrollo de software.

Objetivo

¿Por qué ?

De forma algo humorística en la introducción de este documento indico que el sistema de generación de consultas es algo “chapucero” y además extremadamente ineficiente.

Usabilidad

Solemos pensar que la usabilidad es algo que sólo ha de estar presente en las aplicaciones finales, pero, ¿no somos los desarrolladores a fin de cuentas usuarios también? Puede que no usemos aplicaciones, pero usamos software y si ese software es incómodo, no-placentero y nos obliga a estar constantemente mirando la documentación (porque recordarlo es imposible) no es funcional.

En especial, el crear consultas con una sintaxis basada en arrays es complicado y engorroso. De hecho, para consultas complejas, al final yo optaba por escribir código SQL que (irónicamente) era mucho más limpio.

Eficiencia

Por otro lado, desde el punto de vista de la eficiencia, el sistema que usa DBHelper para la carga masiva de datos es muy poco eficiente. Se trae todos los datos para su procesado directamente. Es decir, no tiene carga diferida, y no, no me refiero a lo que hice de DeferredSet en “los modelos”, sino al uso de un cursor (recordset) que vaya obteniendo de forma eficiente cada una de las tuplas.

Este tipo de carga “de una vez” es muy ineficiente para algunas URLs y de hecho, el servidor de base de datos se resiente ante algunas cargas dado que hay mucha transferencia de golpe. De hecho, ha habido caídas debido a consultas de un gran número de tuplas que han ocasionado un pico de carga muy alto en el sistema.

Uso

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

Este sistema permite la consulta a partir del método `objects` de los modelos que hereden de ROModel, o la consulta

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.

Métodos

select

Permite seleccionar una serie de campos determinados. Por defecto no debería usarse, ya que se supone que se van a traer todos los atributos del modelo.

aggregate

Permite realizar agregaciones (`COUNT`, `SUM`, etc.) sobre un campo. acepta como parámetro un array de objetos `Aggregation` en los que el primer parámetro es la operación de agregación y el segundo es el alias (siempre obligatorio).

Así, cada una de las agregaciones se añadirá como atributo dinámico a cada uno de los objetos.

Nótese que has de tener cuidado cuando hagas la agregación para que ésta haga exactamente lo que quieres.

Ejemplo de uso: cuenta los objetos ExampleUserTag que no tienen por nombre “amigos”.

```
$tags = Tag::objects()->filter(["stack" => "depto_constitucional"])->exclude(["name" => "amigos"])->aggregate([new luloquery\Aggregation("count", "cuenta") ]);
```

select_aggregate

Si no quieres obtener los resultados como atributos de un modelo o simplemente sólo te interesa el valor de la cuenta, puedes directamente obtenerlo llamando a `select_aggregate` en vez de `aggregate`.

limit

Limita el número de objetos obtenidos. Acepta un parámetro (número de elementos obtenidos) o dos (inicio y número de elemento obtenidos desde inicio).

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>`.

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])`:
  - (A=1) OR (B=3 AND C=4)
- `filter(["X"=>2, "Y"=>3], ["W"=>5])`:
  - (X=2 AND Y=3) OR (W=5)

Si la sintaxis de parámetros te resulta engorrosa, también se acepta pasar todas las condiciones OR como un único array que contiene las condiciones AND, esto es, Forma Normal Disyuntiva.

Por ejemplo:

`filter([["stack"=>"alumni", "active"=>0], ["phone"=>null]]):`
- (stack="alumni" AND active=0) OR (phone IS NULL)

exclude

Condiciones de exclusión de la consulta a realizar. Tienen exactamente 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.

Por ejemplo:

```
// Obtención de las etiquetas con fecha de creación menor al 1 de Enero del 2015

Tag::objects()->filter([["creation_datetime__lte" => "2015-01-01"],
```

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[1] y el nombre del campo del modelo destino[2].

Por ejemplo:

```
// Obtención de todas las etiquetas con usuarios cuyo nombre contenga "Prueba" pero
// que no sean las etiquetas con el nombre "amigos",
// quedándonos con las 100 primeras y ordenando por orden ascendente de id
```

[1] 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.
[2] Aquí no hay confusión posible, es el nombre del campo de la tabla destino.

```
$tags = Tag::objects()
        ->filter(["users::first_name__contains" => "Prueba"])
        ->exclude(["name" => "amigos"])
        ->limit(100)
        ->order(["id" => "asc"]);
```

Comparación con valor actual de la tupla

A veces, queremos hacer una comparación con el valor que tiene la tupla, lo haremos mediante la envoltura del valor en una clase, `TupleValue`. Esta clase lo único que hace es dar una semántica distinta al campo que se le pasa en el constructor, indicando que la comparación no viene de una variable, sino del valor que tuviera la tupla concreta.

Ejemplos:

```
// Obtener todos los objetos ExampleUserTag que tengan fecha de finalización
// mayor que fecha de inicio
$tags = Tag::objects()->filter(["end_datetime__lte" =>TupleValue::n("start_datetime") ]);
```

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 una fecha 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`: operador mayor que.
- `gte`: operador mayor o igual que.

Métodos finales

Los métodos finales son aquellos que no devuelven una referencia a this y por tanto no permiten el encadenamiento de filtros y exclusiones, y por supuesto no permiten la obtención de un `recordset`[3] que permita ciclar por ellos.

Update

Ejecuta una actualización sobre los datos. Para poder ejecutar una actualización, se ha de pasar un array de pares (clave => valor) donde las claves son atributos y los valores son los valores que tomará ese atributo.

Ejemplo:

```
// Actualización por un valor concreto
```

[3] Un recordset es la abstracción con la que llama AdoDB a los cursores de las bases de datos con las que trabaja.

```
User::objects()->filter(["slug" => "djrl"])->update(["name" => "Diego J."]);


// Actualización por un valor referenciado de una tabla

// Lo primero es hacer un ALIAS de TupleValue al inicio del fichero .PHP

use \lulo\query\TupleValue as L_V ;

// ...

// Actualiza el nombre de cada tupla por su slug

User::objects()->update(["name" => L_V::n("slug")]);


// Actualización de un valor

// Actualiza el número de visitas incrementándolo en 1, nótese el uso del segundo

// parámetro que indica que lo que se pase al TupleValue está en modo crudo y no

// se ha de modificar, escapar ni nada.

User::objects()->update(["visits" => L_V::n("visits + 1", $raw=true)]);
```

Delete

Ejecuta una eliminación para las tuplas seleccionadas por el filtro.

Ejemplo:

```
// Eliminación del objeto User con id = 12

User::objects()->filter(["stack" => "depto_constitucional", "id" => 12])->delete();
```

Nótese que la eliminación puede eliminar los objetos relacionados en relaciones ForeignKey en las que este modelo sea su padre y las relaciones ManyToMany. Nótese que esta eliminación es una única consulta por lo que es **realmente eficiente** desde el punto de vista de la aplicación.

A continuación indicamos cómo indicar a ROModel que la eliminación ha de ser en cascada.

Padre en una relación ForeignKey de otro modelo

Para que se eliminen los objetos hijo, se ha de indicar: `"on_master_deletion" = true.` en la relación ForeignKey del modelo hijo.

Relación ManyToMany con otro modelo

Para que se eliminen las tuplas de la tabla nexo entre los dos modelos, se ha de indicar: `"on_master_deletion" = true.` en esta relación.

trans

Marca una operación como transacción. Nótese que la transacción sólo envuelve la operación sobre la que se ejecuta y no el resto, para envolver varias operaciones usa la clase Transaction (dentro del namespace luloquery).

sql

Obtiene el código SQL generado para la consulta. Útil para depurar.

sql_for_delete

Obtiene el código SQL de la consulta de eliminación. Útil para depurar.

sql_for_update

Obtiene el código SQL de la consulta de eliminación. Útil para depurar. Nótese que este método requiere como parámetro un array con los campos que se desean actualizar, exactamente igual que el método `update`.

Sobrecarga de operadores

Las LuloQuery tienen implementadas dos interfaces, `Iterator` y `ArrayAccess`.

Iterator

El implementar esta interfaz permite ciclar sobre objetos Query en bucles `foreach`. Esto permite implementar una consulta compleja y obtener los resultados y ciclar por ellos olvidándonos de la carga explícita de los objetos o de si son muchos objetos los que se trae la consulta[4].

Además, tiene la carga perezosa implementada mediante el uso de un cursor, por lo que es más eficiente que traerse todos los datos e ir ciclando por ellos.

Si has usado alguna vez los QuerySet de Django, todo esto te sonará familiar.

ArrayAccess

Permite el acceso mediante un índice entre corchetes (`[]`) a elementos concretos de un objeto Query.

No lo recomiendo y personalmente no me gusta, porque es más intuitivo ciclar usando `foreach` y no acceder mediante índices, pero bueno, ahí está la funcionalidad por si se necesita usarla.

Ejemplos completos

Estarás preguntándote que dónde hay ejemplos reales que puedas verlos. Pues bien, en el namespace `lulo\tests` hay una clase con el nombre `Tester`. En esa clase hay bastantes ejemplos descritos y comentados que pueden ser de utilidad.

Los ejemplos hacen uso de los modelos de ejemplo (aquellos que comienzan por Example) y que están en el stack `_lulo_examples`.

Estos tests se ejecutan desde `la sección tests`.

[4] Hasta cierto punto, es cierto que el servidor de base de datos tendrá menos carga, pero habrá que ver si el servidor de aplicaciones soporta al trabajar con ese número de objetos en la instanciación de una plantilla, etc.

Diseño

Introducción

Lo primero que debes pensar es que se ha diseñado el sistema lo más sencillo y claro posible, de manera que luego se pueda mejorar y ampliar, pero sin perder eficiencia a la hora de generar las consultas.

Por otro lado, yo diría que el hecho de que la lógica esté por una parte y la generación de código SQL esté por otra, le confiere una belleza especial que hemos de respetar y mantener en toda ampliación posterior.

Modelo de datos

Básicamente el modelo de datos se basa en tres clases relacionadas de esta forma:

Query

Este modelo es el modelo principal de este paquete software. Representa una consulta sobre la base de datos, independientemente del tipo de ésta.

Atributos

- **db**: conector de la base de datos. Nombre de la clase que implementa la interfaz DBHelper que se va a usar. Cadena.
- **is_distinct**: booleano que indica si se le ha de aplicar DISTINCT a la consulta SELECT. Booleano.
- **model**: nombre del modelo que se va a consultar. Cadena.
- **model_table**: tabla del modelo que se va a consultar. Cadena.
- **selected_fields**: campos que se van a consultar en el SELECT. Por defecto es null y el sistema por defecto selecciona todos los campos que no son BLOB del modelo.
- **related_models**: ahí vamos almacenando los modelos que aparecen nombrados en las operaciones `filter`, `exclude` o en el `order_by` del Query. Es decir, se almacenan los modelos que aparecerán en la consulta de entre los relacionados con el modelo principal. Array.
- **related_tables**: lo mismo que related_models, pero sólo en el caso de las tablas. Se usa para mejorar la eficiencia de la consulta y permitir la depuración de forma más sencilla. Array.
- **relationships**: las relaciones en sí mismas que han de considerarse a la hora de montar la consulta porque han sido nombradas en un filtro u orden de la consulta. Array.
- **aggregations**: array con los objetos Aggregate que representan a cada una de las agregaciones que ha seleccionado el desarrollador. Array.
- **has_group_by**: indica si tiene la cláusula `GROUP BY` o no, se usa para distinguir consultas `COUNT(*)` de `<atributo1>, <atributo2>, COUNT(<atributoN>)` donde hay que incluir `GROUP BY <atributo1>, <atributo2>`.
- **is_transaction**: indica si la consulta está envuelta en una transacción. Nótese que las transacciones no son anidadas, así que si marcas una Query como con transacción y la vista en sí misma contiene una transacción, la transacción de la Query ejecutará un COMMIT implícito de la transacción de la vista.
- **filters**: array de objetos `ConjunctionCondition`. Cuando un desarrollador llama a `filter` le pasa como parámetro un OR de condiciones AND, y lo puede llamar de dos formas:
  - `filter([[...], [...], [...]])`
  - `filter([...], [...], [...])`

  $Condición_1$ OR $Condición_2$ OR $Condición_3$

  Donde cada $Condición_i$ es una conjunción de comparaciones de atributos del modelo y de los modelos relacionados.

  Sea como sea, ambas formas son equivalentes y cada una de las condiciones se convertirá en un objeto `ConjunctionCondition`, que no es más que un Y lógico de proposiciones.

- **order**: orden de la consulta. El formato es un array de pares `atributo del modelo => asc|desc`.
- **currentIndex**: índice por el que se va recorriendo el Query. Usado en la implementación de `Iterator`.
- **recordSet**: objeto que contiene la abstracción que hace AdoDB de los cursores. A fin de cuentas es lo que se va a recorrer. Object.
- **recordSetSize**: tamaño del recordSet. Útil para saber el número de registros obtenidos al realizar la consulta.
- **results**: almacén-caché de los elementos para evitar tener que consultar constantemente usando el cursor, de manera que a partir del índice actual no haya que consultar con la base de datos. Object Collection.

La mayoría de los métodos son encadenables, en el sentido de que devuelven `$this`, y por tanto pueden seguir aplicándose restricciones y cambios en propiedades sobre la misma consulta. Obviamente, consultas finales como count, delete o update no devuelven `$this`, sino el número de elementos en el primer caso y nada en el segundo y tercero.

ConditionConjunction

Representa una conjunción de proposiciones lógicas, por ejemplo:

A=1 AND B=2 AND C=5

En cada consulta tenemos una lista de conjunciones de condiciones. Se considera que todas las conjunciones de conjunciones de la lista forman una disyunción. Esto es, que en el Query tenemos una Forma Normal Disyuntiva.

Por ejemplo, si tenemos las condiciones:

A=1 AND B=2 AND C=5

D=10 AND X=34

Z=8

Tendríamos las siguiente expresión

(A=1 AND B=2 AND C=5) OR (D=10 AND X=34) OR (Z=8)

Atributos

- **luloquery**: objeto Query del que depende. Se usa un patrón observador en los objetos Condition que dependen de cada ConditionConjunction de forma que puedan indicarle al Query padre que una de las condiciones usa una relación y por tanto, hay que tenerla en cuenta en la generación de la consulta.
- **queryCondition**: grupo de condiciones tal y como las introdujo el desarrollador. Se usa para depurar. Array.
- **conditions**: array de objetos Condition con cada una de las condiciones. Array.
- **conditionsByModel**: array de objetos Condition con cada una de las condiciones agrupadas por modelo. Lo usamos para poder saber qué relaciones con modelos externos incluir en la consulta. Array.

Condition

Contiene cada una de las condiciones. Esto es, el nivel más bajo de toda esta estructura. Básicamente es un atributo, un operador (por defecto =) y un valor (que puede ser el propio valor de la tupla).

- **conditionConjunction**: objeto ConditionConjunction del que depende la condición actual. Object.
- **model**: modelo sobre el que se ejecuta la condición. Se guarda porque es práctico indicar claramente si el modelo es el modelo actual o un modelo remoto. String.

- **table**: tabla a la que pertenece el atributo. Es útil para luego usarlo en la plantilla de la consulta. String.
- **table_alias**: en realidad, en la mayoría de los casos, no se usa el nombre de las tablas sino el alias de la tabla. En el caso de que sea la tabla principal, será siempre `"main_table"` y en el caso de que sea una tabla relacionada, será el nombre de la relación. String.
- **field**: nombre del campo que se quiere consultar. Nótese que el campo es de la tabla **table**, que no tiene porqué ser la tabla principal, claro está. String.
- **operator**: operador que relaciona el campo field con el valor. El operador por defecto es = (igualdad) pero si se pasa alguno de éstos:
    - `contains`: indica si contiene el valor (`LIKE "%valor%"`).
    - `notcontains`: indica si no contiene el valor (`NOT LIKE "%valor%"`).
    - `startswith`: indica si comienza por el valor (`LIKE "valor%"`)
    - `endswith`: indica si termina por el valor (`LIKE "%valor"`)
    - `in`: indica si contiene el valor (`IN "item`$_1$`", …, "item`$_N$`"`). Donde `valor = ["item`$_1$`", …, "item`$_N$`"]`.
    - `range`: indica si tupla está en un rango. Es decir es mayor que un valor y menor que otro.
    - `eq`: igualdad (`= "valor"`).
    - `noteq`: no igualdad (`<> "valor"`).
    - `lt`: menor estricto que (`< "valor"`).
    - `lte`: menor o igual que (`<= "valor"`).
    - `gt`: mayor estricto que (`> "valor"`).
    - `gte`: mayor o igual que (`>= "valor"`).

  se obtendrá correctamente el operador SQL asociado como se ha indicado.

- **value**: valor que se desea comparar, si es un `TupleValue`, entonces se generará la condición con el valor actual de cada una de la tupla. String o Object.

Flujo de ejecución

Construcción del Query

Internamente, el objeto Query se construye a partir del `ROModel`, éste le pasa su nombre en el constructor e inmediatamente el objeto Query rellena:

- Conector db
- Nombre del modelo
- Tabla del modelo

Pone los valores iniciales de los filtros, un array vacío ya que no hay filtros.

Las relaciones que se usan en la consulta están vacías, porque inicialmente no sabemos si se van a usar las relaciones para con este modelo.

Eso sí, preparamos el Query para obtener los atributos que no sean BLOB del modelo.

Aggregate y select_aggregate

Los agregadores son las operaciones COUNT, SUM… Si quieres, puedes añadir tantos agregadores como quieras a una consulta.

Automáticamente se crearán los GROUP BY para todo atributo que tengas seleccionado.

Estos métodos lo único que hacen es añadir las operaciones de agregación al Query, para luego que en la plantilla, se incluyan éstos.

Para que los *aggregates* funcionen, has de pasarle un array de objetos Aggregation, que no son más que un objeto contenedor en el que se indica la operación (COUNT, SUM…) y el alias que quieres darle para que luego aparezca como tal.

Collection

No deberías usar este método si lo que quieres es ciclar por los resultados, pero ¿cómo obtener una colección a partir de un Query? Este método ignora la carga "perezosa" y te devuelve una colección con todos los objetos que se obtienen de base de datos.

El uso de este método hace que el propósito principal de LuloQuery se desvanezca, por lo que antes de usar `collection`, pregúntate si de verdad lo necesitas y si lo vas a usar en un contexto correcto. Lo más probable es que no te sea necesario.

Count

Construye un agregado estilo COUNT(*) y obtiene el resultado.

Es útil para obtener cuentas de elementos de forma eficiente.

Filter

¿Qué hace filter? Va procesando cada uno de los conjuntos de condiciones y construyendo un `ConditionConjunction` con cada uno de los conjuntos.

ConditionConjunction

Cada `ConditionConjunction` que se construye lo que hace es trocearse en condiciones y construir objetos `Condition`.

Condition

Los objetos `Condition` lo primero que hacen es ver si tienen un campo remoto. Si lo tienen, llaman al método `Query::addRelatedModel` para indicarle al Query que en las condiciones debe incluir un join con el modelo relacionado. Ese método ya se encarga de comprobar el tipo de relación que es, si tiene tabla de nexo, etc.

Luego, ha de convertir los operadores en formato LuloQuery a operadores SQL estándar. Esto se hace en `getSqlOperator`.

Por último, tienen un método llamado `sql` que convierte la condición en código SQL, lo que es perfecto para su uso en las plantillas. Este método hace llamadas a la obtención del campo, valor y operador correcto para SQL.

Devolución

Para que se permita el encadenamiento de métodos, filter devuelve `$this`.

Delete

Ejecuta una sentencia de DELETE, en realidad, genera un SQL a partir de la plantilla. Nótese que en la plantilla, se hereda de la plantilla de SELECT, para poder aprovechar algunos fragmentos.

El código SQL es un DELETE con JOIN, de manera que si están marcados las relaciones como

Exclude

Exclude hace exactamente lo mismo que filter, pero marca la nueva condición como condición negativo, esto es, que va precedida por un `NOT`, dado que estamos buscando que **no cumplan** con esa condición.

Limit

Inicia el atributo limit con un array en el que se indica el número de la primera tupla y el número de tuplas a obtener.

Cualquiera que haya trabajado con SQL dialecto MySQL sabrá que a fin de cuentas, luego en la plantilla, se genera un código SQL `LIMIT (<inicio>, <número de tuplas>)`.

Trans

El método `trans` pone un flag en el objeto Query a TRUE que se lee desde las plantillas SQL y si está activo, se envuelve la plantilla con el SQL generado en una transacción.

Nótese que no se permiten transacciones anidadas, de manera que no puedes hacer una transacción y luego llamar a un Query con trans.

Update

Ejecuta una sentencia de UPDATE, en realidad, genera un SQL a partir de la plantilla. Nótese que en la plantilla, se hereda de la plantilla de SELECT, para poder aprovechar algunos fragmentos.

Generación de consultas

Todas las consultas se basan en plantillas de TWIG. Esto se ha hecho para separar completamente lo que es la generación del código SQL de la lógica de la aplicación de construcción de las estructuras de información necesarias para interpretar las operaciones del usuario.

Por otro lado, es importante destacar que el código SQL generado para cada una de las consultas, genera una única consulta. Esto es, tal y como hemos dicho antes, se evita hacer un conjunto de consultas, sino que se ejecuta una única consulta logrando mayor eficiencia en cuanto a número de accesos a la base de datos.

SELECT

La consulta SELECT se basa en una plantilla con un SELECT. Esta plantilla incluye dos plantillas que generan de forma automática las condiciones de unión con los modelos relacionados.

UPDATE

Plantillas que hereda de la plantilla del SELECT, sólo que cambia de orden las condiciones y sobrescribe el SELECT, sustituyéndolo por un UPDATE <TABLA> SET.

DELETE

De forma similar a como hago en UPDATE, heredo de la plantilla del SELECT y sobrescribo lo necesario para obtener una consulta de eliminación que permita condiciones con las relaciones.

Extensión de LuloQuery

Para extender LuloQuery seguiremos con el método de trabajo de Lulo. Hemos de recordar que estamos en un equipo diverso, en el que la forma de trabajar de cada uno de distinta. Pero, por otro lado, debemos poner unos estándares de calidad para poder colaborar entre nosotros y que no nos ocurra “una torre de Babel”.

Discusión de mejoras

Cuando tengas alguna mejora en mente, piénsala primero (recomiendo que la escribas). Luego la hablamos entre nosotros. Quizás puede que haya otra forma de hacer lo que pides, o que ya esté implementada.

También, claro está, puede que sea realmente necesaria. En ese caso, vemos la mejor forma de ponerla en práctica discutiendo sobre la forma mejor de hacerla (hablo sobre esto en el siguiente punto).

También es cierto, que conociendo el contexto de uso, es más fácil entender una funcionalidad.

Seguimiento de buenas prácticas y estándares

Vamos a seguir buenas prácticas, heurísticas y estándares de la Ingeniería del Software. ¿Por qué? Para mantener la **deuda técnica**[5] al mínimo.

[5] Deuda Técnica es un término que se usa para indicar una baja calidad del software que impide mantener y/o ampliar fácilmente un software. Más información en esta entrada de Javier Garzás.

Es preferible no implementar una mejora hasta que no tengamos el diseño software claro que implementarla y ver que falla o que no es ampliable (o mantenible).

Intentemos adelantarnos a los problemas y seamos muy estrictos en cuanto a documentación y diseño. No queremos volver a los errores de la clase Tablón[6], o al generador de modelos antiguo que hice hace año. Buscamos un proyecto que perdure durante años y permita el desarrollo rápido de aplicaciones web.

Divide y vencerás

Antes de abarcar un gran cambio de golpe, hemos de trocear la funcionalidad, de manera que podamos abarcarla de forma más sencilla, poco a poco.

Prototipos

El uso de prototipos es importante. De hecho, esta versión de LuloQuery que ves aquí es la segunda que hago, la primera se hizo de una forma completamente distinta y con un diseño menos flexible.

Los prototipos rápidos de desarrollar son una buena herramienta para poder comprobar

El prototipo puede convertirse en el producto final, pero tendrá que ser refactorizado y limpiado de manera que el "código sucio" desaparezca y sea reemplazado totalmente.

En definitiva, buscamos inicialmente que funcione, pero luego lo convertimos a software de alta calidad.

Código limpio

El código es la parte fundamental del software, es cierto que la documentación es crucial, pero lo que al final se ejecuta, es código.

Por tanto, el código ha de seguir también normas, de manera que se lea de forma **entendible y placentera**.

Buenas prácticas

Las buenas prácticas en cuanto a código no deben ser olvidadas:

- Baja complejidad ciclomática.
- Principio de única responsabilidad.
- Nombres de variables claros.

Comentarios

Cada grupo de sentencias que realicen una acción debe estar detallada mediante un comentario. Da igual repetir información, porque lo que para nosotros puede ser obvio, para otro desarrollador no lo es tanto.

No quiero tener que ir a la cabecera de una función para saber qué hace una función, quiero que se pueda entender como si fuera un texto legible lo que se está haciendo.

Un código bien comentado es una herramienta muy importante de ayuda a los desarrolladores a la hora de mantener una aplicación.

Tests

Hay una clase **Tester** que contiene una serie de pruebas. Esas pruebas deben ir incrementándose según la nueva funcionalidad vaya introduciéndose.

Si un cambio en Query no pasa las pruebas, no se subirá nunca a producción.

[6] Por si no la conoces, la aplicación de Tablón 1.0 fue una de las primeras aplicaciones que se hizo para UniWeb y era tan baja la calidad del software, que ampliarla o modificarla se convertía en una tarea difícil y que había que realizar con extremo cuidado, para evitar romper la funcionalidad existente. Trabajar sobre esa clase, incrementaba el estrés en los desarrolladores, afectando de forma negativa a la productividad.

Mejoras futuras

Independizarlo de MySQL

Ahora mismo las plantillas TWIG con la generación del código SQL están preparadas para su uso con MySQL, dado que la sintaxis del código generado es particular. Bien es cierto que se ha intentado usar código SQL lo más estándar posible, pero el uso de LIMIT y otras características no forman parte del estándar.

Se debería especializar cada una de las plantillas de manera que hubiera versiones para cada una de los sistemas de bases de datos que usamos.

SELECT FOR UPDATE

Actualmente no hay forma de bloquear unas tuplas concretas. MySQL-MyISAM no proporciona esa funcionalidad, pero MySQL-InnoDB sí que permite bloquear por tuplas, lo que puede ser útil para un desarrollador a la hora de no permitir que otros clientes puedan editar esa tupla concreta hasta que temine la transacción.

Esto es,

```
START TRANSACTION
    SELECT *
    FROM <TABLA>
    WHERE <CONDICIÓN>
    FOR UPDATE
COMMIT
```

De manera que todas las tuplas que hayan sido seleccionadas mediante <CONDICIÓN> estarán bloqueadas hasta la finalización de la transacción, cuando se ejecute el COMMIT.

Epílogo

Espero que hayas disfrutado, tú desarrollador que lees estas páginas y hayas visto aquí una herramienta potente que te sirva para el desarrollo de nuevos proyectos sobre intelliweb.

Cualquier sugerencia es bienvenida, pero recuerda que este proyecto es un proyecto de la comunidad, en el que Diego J. es, ni más, ni menos, que el coordinador.

Lo dicho, espero que hayas pasado un buen rato y hayas descubierto algo nuevo que te haga el trabajo más fácil.