Добавление поддержки СУБД Firebird в фреймворк Laravel

Автор: Денис Симонов
28.08.2016

Оглавление

Добавление поддержки СУБД Firebird в фреймворк Laravel 3
Класс подключения FirebirdConnection 3
Класс грамматики для построения DML запросов 6
Класс грамматики для построения DDL запросов 8
Добавление поддержки последовательностей 19
Заключение 23

Добавление поддержки СУБД Firebird в фреймворк Laravel

Во время написания примера (позже будет ссылка) веб-приложения на PHP с использованием СУБД Firebird возник вопрос выбора фреймворка для разработки с использованием архитектурной модели MVC. Выбор фреймворков под PHP очень большой, но наиболее удобным, простым и легко расширяемым показался Laravel. Однако этот фреймворк не поддерживал из коробки СУБД Firebird. Laravel использует для работы с базой данных драйвера PDO. Поскольку для Firebird существует драйвер PDO, то это натолкнуло меня на мысль, что можно с некоторыми усилиями заставить работать Laravel с Firebird.

Laravel — бесплатный веб-фреймворк с открытым кодом, предназначенный для разработки с использованием архитектурной модели MVC (англ. Model View Controller — модель-представление-контроллер). Это очень удобный и легко расширяемый фреймворк для построения ваших веб-приложений. Из коробки фреймворк Laravel поддерживает 4 СУБД: MySQL, Postgres, SQLite и MS SQL Server. В этой статье я расскажу как добавить ещё одну СУБД Firebird.

Класс подключения FirebirdConnection

Каждый раз, когда вы подключаетесь к базе данных, с помощью фабрики Illuminate\Database\Connectors\ConnectionFactory создаётся конкретный экземпляр подключения в зависимости от типа СУБД, который реализует интерфейс Illuminate\Database\ConnectionInterface. Кроме того, фабрика создаёт некий коннектор, который на основе параметров конфигурации формирует строку подключения и передаёт её в конструктор подключения PDO. Коннектор создаётся в методе createConnector. Немного модифицируем его так, чтобы он мог создавать коннектор для Firebird.

```
public function createConnector(array $config)
{
    if (! isset($config['driver'])) {
        throw new InvalidArgumentException('A driver must be specified.');
    }

    if ($this->container->bound($key = "db.connector.{$config['driver']}")) {
        return $this->container->make($key);
    }

    switch ($config['driver']) {
        case 'mysql':
            return new MySqlConnector;
        case 'pgsql':
            return new PostgresConnector;
        case 'sqlite':
            return new SQLiteConnector;
        case 'sqlsrv':
            return new SqlServerConnector;
        case 'firebird': // Add support Firebird
            return new FirebirdConnector;
    }

    throw new InvalidArgumentException("Unsupported driver [{$config['driver']}]");
}
```

Сам класс коннектора Illuminate\Database\Connectors\FirebirdConnector определим чуть позже. А пока модифицируем метод createConnection предназначенный для создания подключения реализующего интерфейс Illuminate\Database\ConnectionInterface.

```
protected function createConnection($driver, $connection, $database, $prefix = '',
                                    array $config = [])
{
    if ($this->container->bound($key = "db.connection.{$driver}")) {
        return $this->container->make($key, [$connection, $database, $prefix, $config]);
    }

    switch ($driver) {
        case 'mysql':
            return new MySqlConnection($connection, $database, $prefix, $config);
        case 'pgsql':
            return new PostgresConnection($connection, $database, $prefix, $config);
        case 'sqlite':
            return new SQLiteConnection($connection, $database, $prefix, $config);
        case 'sqlsrv':
            return new SqlServerConnection($connection, $database, $prefix, $config);
        case 'firebird': // Add support Firebird
            return new FirebirdConnection($connection, $database, $prefix, $config);
    }

    throw new InvalidArgumentException("Unsupported driver [$driver]");
 }
```

Теперь перейдём к созданию коннектора для Firebird – класса Illuminate\Database\Connectors\FirebirdConnector. За основу можно взять любой из существующих коннекторов, например коннектор для Postgres и переделаем его под Firebird.

Для начала изменим метод для формирования строки подключения в соответствии с форматом строки описанной в документации:

```
protected function getDsn(array $config) {
    $dsn = "firebird:dbname=";
    if (isset($config['host'])) {
        $dsn .= $config['host'];
    }
    if (isset($config['port'])) {
        $dsn .= "/" . $config['port'];
    }
    $dsn .= ":" . $config['database'];
    if (isset($config['charset'])) {
        $dsn .= ";charset=" . $config['charset'];
    }
    if (isset($config['role'])) {
        $dsn .= ";role=" . $config['role'];
    }

    return $dsn;
 }
```

Метод, создающий PDO подключение, в данном случае будет максимально упрощён

```
public function connect(array $config) {
    $dsn = $this->getDsn($config);

    $options = $this->getOptions($config);

    // We need to grab the PDO options that should be used while making the brand
    // new connection instance. The PDO options control various aspects of the
    // connection's behavior, and some might be specified by the developers.
    $connection = $this->createConnection($dsn, $config, $options);

    return $connection;
}
```

Сами подключения для различных СУБД, используемые в Laravel, наследуют класс Illuminate\Database\Connection. Именно этот класс инкапсулирует в себе все возможности по работе с БД, которые используются в Laravel, в частности в ORM Eloquent. В классах наследниках (под каждый тип СУБД) реализованы методы возвращающие экземпляр класса с описанием грамматики, который требуются при построении DML запросов, и экземпляр грамматики для DDL запросов (используется в миграции), также помощники для добавления имени схемы к таблице. Этот класс будет выглядеть следующим образом:

```
namespace Illuminate\Database;

use Illuminate\Database\Query\Processors\FirebirdProcessor;
use Doctrine\DBAL\Driver\PDOFirebird\Driver as DoctrineDriver;
use Illuminate\Database\Query\Grammars\FirebirdGrammar as QueryGrammar;
use Illuminate\Database\Schema\Grammars\FirebirdGrammar as SchemaGrammar;

class FirebirdConnection extends Connection
{

    /**
     * Get the default query grammar instance.
     *
     * @return \Illuminate\Database\Query\Grammars\FirebirdGrammar
     */
    protected function getDefaultQueryGrammar()
    {
        return $this->withTablePrefix(new QueryGrammar);
    }

    /**
     * Get the default schema grammar instance.
     *
     * @return \Illuminate\Database\Schema\Grammars\FirebirdGrammar
     */
    protected function getDefaultSchemaGrammar()
    {
        return $this->withTablePrefix(new SchemaGrammar);
    }

    /**
     * Get the default post processor instance.
     *
     * @return \Illuminate\Database\Query\Processors\FirebirdProcessor
     */
    protected function getDefaultPostProcessor()
    {
        return new FirebirdProcessor;
    }

    /**
     * Get the Doctrine DBAL driver.
     *
     * @return \Doctrine\DBAL\Driver\PDOFirebird\Driver
     */
    protected function getDoctrineDriver()
    {
        return new DoctrineDriver;
    }
}
```

Метод для возврата экземпляра драйвера Doctrine формально требуется, поэтому мы вводим его, но у меня не было цели работать с Doctrine (только с Eloquent), поэтому реализацию я не делал. При желании вы можете это сделать самостоятельно.

Постпроцессор `Illuminate\Database\Query\Processors\FirebirdProcessor` предназначен для дополнительной обработки результатов запросов, в частности он помогает извлекать идентификатор записи из INSERT запроса. Определение его

реализация полностью скопирована из `Illuminate\Database\Query\Processors\PostgresProcessor`.

Класс грамматики для построения DML запросов

Теперь переходим к самому интересному и важному, а именно к описанию грамматики для построения DML запросов. Именно этот класс отвечает за преобразование выражения записанного для построителя запросов Laravel в диалект языка SQL, используемой в вашей СУБД. Зная, что делает определённая конструкция в одной СУБД, вы без труда можете записать эту конструкцию для Firebird. На самом деле DML часть языка SQL достаточно стандартна и не значительно отличается для различных СУБД, по крайней мере, с точки зрения тех запросов, которые можно построить с помощью Laravel.

Грамматики DML запросов наследуются от класса Illuminate\Database\Query\Grammars\Grammar. В защищённом свойстве $selectComponents перечислены части SELECT запроса, из которых собирается запрос построителем \Illuminate\Database\Query\Builder. В методе compileComponents происходит обход этих частей и для каждой из них вызывается метод с именем, которое состоит из имени части запроса и префикса compile.

```
/**
 * Compile the components necessary for a select clause.
 *
 * @param  \Illuminate\Database\Query\Builder  $query
 * @return array
 */
protected function compileComponents(Builder $query)
{
    $sql = [];

    foreach ($this->selectComponents as $component) {
        // To compile the query, we'll spin through each component of the query and
        // see if that component exists. If it does we'll just call the compiler
        // function for the component which is responsible for making the SQL.
        if (! is_null($query->$component)) {
            $method = 'compile'.ucfirst($component);

            $sql[$component] = $this->$method($query, $query->$component);
        }
    }

    return $sql;
}
```

Зная этот факт, становиться понятно, что и где надо модифицировать. Теперь настало время создать наследник класса Illuminate\Database\Query\Grammars\Grammar для определения грамматики Firebird - Illuminate\Database\Query\Grammars\FirebirdGrammar. Определим основные отличительные особенности Firebird.

Для простого INSERT запроса основное отличие заключается в возможности возврата только что добавленной строки с помощью предложения RETURNING. В Laravel это используется для возврата идентификатора, только что добавленной строки. Однако MySQL не имеет такой возможности, а потому метод compileInsertGetId выглядит по-разному для разных СУБД. Firebird поддерживает предложение RETURNING, как и СУБД Postgres, а поэтому этот метод можно взять из грамматики для Postgres. Выглядеть он будет следующим образом:

```
    /**
     * Compile an insert and get ID statement into SQL.
     *
     * @param  \Illuminate\Database\Query\Builder  $query
     * @param  array   $values
     * @param  string  $sequence
     * @return string
     */
    public function compileInsertGetId(Builder $query, $values, $sequence) {
        if (is_null($sequence)) {
            $sequence = 'id';
        }

        return $this->compileInsert($query, $values) . ' returning ' . $this->wrap($sequence);
    }
```

Пожалуй, самым основным отличием для SELECT запроса является ограничение на количество записей возвращаемых запросом, которое часто используется при постраничной навигации. Например, следующее выражение

```
DB::table('goods')->orderBy('name')->skip(10)->take(20)->get();
```

В различных СУБД будет выглядеть на языке SQL очень по разному.

В MySQL это выглядит так:

```
SELECT *
FROM goods
ORDER BY name
LIMIT 10, 20
```

в Postgres так:

```
SELECT *
FROM goods
ORDER BY name
LIMIT 20 OFFSET 10
```

В Firebird сразу три варианта:

начиная с версии 1.5

```
SELECT FIRST(10) SKIP(20) *
FROM goods
ORDER BY name
```

начиная с версии 2.0 добавляется ещё одна конструкция

```
SELECT *
FROM goods
ORDER BY name
ROWS 21, 30
```

начиная с версии 3.0 ещё добавлена конструкция из стандарта SQL-2011

```
SELECT *
FROM color
ORDER BY name
OFFSET 20 ROWS
FETCH FIRST 10 ROWS ONLY
```

Самым удобным и правильным вариантом, конечно, является вариант из стандарта, но хотелось бы чтобы Laravel поддерживал одновременно Firebird 2.5 и 3.0, поэтому мы

выберем второй вариант. В этом случае методы compileLimit и compileOffset будут выглядеть следующим образом:

```
/**
 * Compile the "limit" portions of the query.
 *
 * @param  \Illuminate\Database\Query\Builder  $query
 * @param  int  $limit
 * @return string
 */
protected function compileLimit(Builder $query, $limit) {
    if ($query->offset) {
        $first = (int) $query->offset + 1;
        return 'rows ' . (int) $first;
    } else {
        return 'rows ' . (int) $limit;
    }
}

/**
 * Compile the "offset" portions of the query.
 *
 * @param  \Illuminate\Database\Query\Builder  $query
 * @param  int  $offset
 * @return string
 */
protected function compileOffset(Builder $query, $offset) {
    if ($query->limit) {
        if ($offset) {
            $end = (int) $query->limit + (int) $offset;
            return 'to ' . $end;
        } else {
            return '';
        }
    } else {
        $begin = (int) $offset + 1;
        return 'rows ' . $begin . ' to 2147483647';
    }
}
```

Следующее чем отличаются запросы так это извлечением частей даты, это делается с помощью метода dateBasedWhere. В Firebird для этого используется стандартная функция EXTRACT. С учётом этого наш метод будет выглядеть следующим образом:

```
/**
 * Compile a date based where clause.
 *
 * @param  string  $type
 * @param  \Illuminate\Database\Query\Builder  $query
 * @param  array  $where
 * @return string
 */
protected function dateBasedWhere($type, Builder $query, $where) {
    $value = $this->parameter($where['value']);

    return 'extract(' . $type . ' from ' . $this->wrap($where['column']) . ') '
         . $where['operator'] . ' ' . $value;
}
```

Вот собственно и всё, все основные отличительные особенности были учтены. Полностью реализованный класс Illuminate\Database\Query\Grammars\FirebirdGrammar вы можете найти в исходных кодах приложенных к статье.

Класс грамматики для построения DDL запросов

Теперь переходим к более сложной грамматики, используемой при построении схем БД. Эта грамматика используется для так называемых миграций (в терминах Laravel). Здесь различий между различными СУБД гораздо больше, начиная от типов данных, заканчивая автоинкрементными полями. Кроме того, здесь потребуется здесь потребуется переписать запросы к ряду системных таблиц для определения наличия таблицы или столбца.

Грамматики DDL запросов наследуются от класса Illuminate\Database\Schema\Grammars\Grammar. Создадим свою грамматику Illuminate\Database\Schema\Grammars\FirebirdGrammar. В защищённом свойстве $modifiers перечислены модификаторы полей таблицы, для каждого из модификаторов, перечисленных в массиве должны быть метод, который начинается с modify после чего идёт имя модификатора. Строим эти методы по аналогии с грамматикой для MySQL, но с учётом специфики Firebird.

```
class FirebirdGrammar extends Grammar {

    /**
     * The possible column modifiers.
     *
     * @var array
     */
    protected $modifiers = ['Charset', 'Collate', 'Increment', 'Nullable', 'Default'];

    /**
     * The columns available as serials.
     *
     * @var array
     */
    protected $serials = ['bigInteger', 'integer',
        'mediumInteger', 'smallInteger', 'tinyInteger'];

// …… пропущено ……

    /**
     * Get the SQL for a character set column modifier.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $column
     * @return string|null
     */
    protected function modifyCharset(Blueprint $blueprint, Fluent $column)
    {
        if (! is_null($column->charset)) {
            return ' character set '.$column->charset;
        }
    }

    /**
     * Get the SQL for a collation column modifier.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $column
     * @return string|null
     */
    protected function modifyCollate(Blueprint $blueprint, Fluent $column)
    {
        if (! is_null($column->collation)) {
            return ' collate '.$column->collation;
        }
    }

    /**
     * Get the SQL for a nullable column modifier.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $column
     * @return string|null
     */
    protected function modifyNullable(Blueprint $blueprint, Fluent $column) {
        return $column->nullable ? '' : ' not null';
    }
```

```
    /**
     * Get the SQL for a default column modifier.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $column
     * @return string|null
     */
    protected function modifyDefault(Blueprint $blueprint, Fluent $column) {
        if (!is_null($column->default)) {
            return ' default ' . $this->getDefaultValue($column->default);
        }
    }

    /**
     * Get the SQL for an auto-increment column modifier.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $column
     * @return string|null
     */
    protected function modifyIncrement(Blueprint $blueprint, Fluent $column) {
        if (in_array($column->type, $this->serials) && $column->autoIncrement) {
            return ' primary key';
        }
    }

// …… пропущено ……

}
```

В массиве $serials перечислены типы (доступные в Laravel) для которых доступен модификатор Increment (автоинкрементный столбец). На типах доступных в Laravel стоит остановиться отдельно. Типы доступные в Laravel перечислены в документации по миграции параграф «Доступные типы столбцов». Для преобразования типа доступного в Laravel в тип данных конкретной СУБД внутри грамматики используются методы, начинающиеся со слова type после которого следует имя типа.

```
    /**
     * Create the column definition for a char type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeChar(Fluent $column) {
        return "char({$column->length})";
    }

    /**
     * Create the column definition for a string type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeString(Fluent $column) {
        return "varchar({$column->length})";
    }

    /**
     * Create the column definition for a text type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeText(Fluent $column) {
        return 'BLOB SUB_TYPE TEXT';
    }

    /**
     * Create the column definition for a medium text type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeMediumText(Fluent $column) {
```

```
        return 'BLOB SUB_TYPE TEXT';
    }

    /**
     * Create the column definition for a long text type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeLongText(Fluent $column) {
        return 'BLOB SUB_TYPE TEXT';
    }

    /**
     * Create the column definition for a integer type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeInteger(Fluent $column) {
        return $column->autoIncrement ? 'INTEGER GENERATED BY DEFAULT AS IDENTITY' : 'INTEGER';
    }

    /**
     * Create the column definition for a big integer type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeBigInteger(Fluent $column) {
        return $column->autoIncrement ? 'BIGINT GENERATED BY DEFAULT AS IDENTITY' : 'BIGINT';
    }

    /**
     * Create the column definition for a medium integer type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeMediumInteger(Fluent $column) {
        return $column->autoIncrement ? 'INTEGER GENERATED BY DEFAULT AS IDENTITY' : 'INTEGER';
    }

    /**
     * Create the column definition for a tiny integer type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeTinyInteger(Fluent $column) {
        return $column->autoIncrement ? 'SMALLINT GENERATED BY DEFAULT AS IDENTITY' : 'SMALLINT';
    }

    /**
     * Create the column definition for a small integer type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeSmallInteger(Fluent $column) {
        return $column->autoIncrement ? 'SMALLINT GENERATED BY DEFAULT AS IDENTITY' : 'SMALLINT';
    }

    /**
     * Create the column definition for a float type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeFloat(Fluent $column) {
        return $this->typeDouble($column);
    }

    /**
     * Create the column definition for a double type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeDouble(Fluent $column) {
```

```
    return 'double precision';
}

/**
 * Create the column definition for a decimal type.
 *
 * @param  \Illuminate\Support\Fluent  $column
 * @return string
 */
protected function typeDecimal(Fluent $column) {
    return "decimal({$column->total}, {$column->places})";
}

/**
 * Create the column definition for a boolean type.
 *
 * @param  \Illuminate\Support\Fluent  $column
 * @return string
 */
protected function typeBoolean(Fluent $column) {
    return 'boolean';
}

/**
 * Create the column definition for an enum type.
 *
 * @param  \Illuminate\Support\Fluent  $column
 * @return string
 */
protected function typeEnum(Fluent $column) {
    $allowed = array_map(function ($a) {
        return "'" . $a . "'";
    }, $column->allowed);

    return "varchar(255) check (\"{$column->name}\" in (" . implode(', ', $allowed) . '))';
}

/**
 * Create the column definition for a json type.
 *
 * @param  \Illuminate\Support\Fluent  $column
 * @return string
 */
protected function typeJson(Fluent $column) {
    return 'varchar(8191)';
}

/**
 * Create the column definition for a jsonb type.
 *
 * @param  \Illuminate\Support\Fluent  $column
 * @return string
 */
protected function typeJsonb(Fluent $column) {
    return 'varchar(8191)';
}

/**
 * Create the column definition for a date type.
 *
 * @param  \Illuminate\Support\Fluent  $column
 * @return string
 */
protected function typeDate(Fluent $column) {
    return 'date';
}

/**
 * Create the column definition for a date-time type.
 *
 * @param  \Illuminate\Support\Fluent  $column
 * @return string
 */
protected function typeDateTime(Fluent $column) {
    return 'timestamp';
}

/**
 * Create the column definition for a date-time type.
 *
```

```
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeDateTimeTz(Fluent $column) {
        return 'timestamp';
    }

    /**
     * Create the column definition for a time type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeTime(Fluent $column) {
        return 'time';
    }

    /**
     * Create the column definition for a time type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeTimeTz(Fluent $column) {
        return 'time';
    }

    /**
     * Create the column definition for a timestamp type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeTimestamp(Fluent $column) {
        if ($column->useCurrent) {
            return 'timestamp default CURRENT_TIMESTAMP';
        }

        return 'timestamp';
    }

    /**
     * Create the column definition for a timestamp type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeTimestampTz(Fluent $column) {
        if ($column->useCurrent) {
            return 'timestamp default CURRENT_TIMESTAMP';
        }

        return 'timestamp';
    }

    /**
     * Create the column definition for a binary type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeBinary(Fluent $column) {
        return 'varchar(8191) CHARACTER SET OCTETS';
    }

    /**
     * Create the column definition for a uuid type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeUuid(Fluent $column) {
        return 'char(36)';
    }

    /**
     * Create the column definition for an IP address type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
```

```
     */
    protected function typeIpAddress(Fluent $column) {
        return 'varchar(45)';
    }

    /**
     * Create the column definition for a MAC address type.
     *
     * @param  \Illuminate\Support\Fluent  $column
     * @return string
     */
    protected function typeMacAddress(Fluent $column) {
        return 'varchar(17)';
    }
```

Замечание об автоинкрементных столбцах

Identity столбцы доступны начиная с Firebird 3.0. До версии 3.0 для аналогичной функциональности применяли создание генератора и BEFORE INSERT триггер. Пример:

```
CREATE TABLE USERS (
    ID INTEGER GENERATED BY DEFAULT AS IDENTITY,
…
);
```

Аналогичную функциональность можно получить так:

```
CREATE TABLE USERS (
    ID INTEGER,
…
);

CREATE SEQUENCE SEQ_USERS;

CREATE TRIGGER TR_USERS_BI FOR USERS
ACTIVE BEFORE INSERT
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = NEXT VALUE FOR SEQ_USERS;
END
```

Миграции Laravel не поддерживают никаких объектов схемы кроме таблиц. Т.е. создание и модификация последовательностей, а тем более триггеров не поддерживается. Тем не менее, последовательности являются частью функционала довольно большого количества СУБД, в том числе Postgres и MS SQL (начиная с 2012). Как добавить поддержку последовательностей в миграции Laravel будет описано позже в данной статье.

Теперь добавим два метода, которые возвращает запрос для проверки существования таблицы и столбца внутри таблицы.

```
/**
  * Compile the query to determine if a table exists.
  *
  * @return string
  */
public function compileTableExists() {
```

```
    return 'select * from RDB$RELATIONS where RDB$RELATION_NAME = ?';
}


/**
  * Compile the query to determine the list of columns.
  *
  * @param  string  $table
  * @return string
  */
public function compileColumnExists($table) {
    return "select TRIM(RDB\$FIELD_NAME) AS \"column_name\" from RDB\$RELATION_FIELDS where
RDB\$RELATION_NAME = '$table'";
}
```

Добавим метод compileCreate предназначенный для создания оператора CREATE TABLE. Этот же метода используется для создания временных таблиц GTT. Довольно странно, но даже для СУБД Postgres создаётся только один тип GTT - ON COMMIT DELETE ROWS, мы же реализуем поддержку сразу обоих типов GTT.

```
    /**
     * Compile a create table command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileCreate(Blueprint $blueprint, Fluent $command) {
        $columns = implode(', ', $this->getColumns($blueprint));

        $sql = $blueprint->temporary ? 'create temporary' : 'create';

        $sql .= ' table ' . $this->wrapTable($blueprint) . " ($columns)";

        if ($blueprint->temporary) {
            if ($blueprint->preserve) {
                $sql .= ' ON COMMIT DELETE ROWS';
            } else {
                $sql .= ' ON COMMIT PRESERVE ROWS';
            }
        }

        return $sql;
    }
```

Класс Illuminate\Database\Schema\Blueprint не содержит свойства $preserve, поэтому добавим его, а так же метод для его установки. Класс Blueprint предназначен для генерирования запроса или набора запросов для поддержки создания, модификации и удаления метаданных таблицы.

```
class Blueprint
{
// …… Пропущено

    /**
     * Whether a temporary table such as ON COMMIT PRESERVE ROWS
     *
     * @var bool
     */
    public $preserve = false;

// …… Пропущено

    /**
     * Indicate that the temporary table as ON COMMIT PRESERVE ROWS.
     *
     * @return void
     */
    public function preserveRows() {
        $this->preserve = true;
    }
```

```
// …… Пропущено

}
```

Вернёмся к классу грамматики Illuminate\Database\Schema\Grammars\ FirebirdGrammar. Добавим в него метод для создания оператора удаления таблицы.

```
    /**
     * Compile a drop table command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileDrop(Blueprint $blueprint, Fluent $command) {
        return 'drop table ' . $this->wrapTable($blueprint);
    }
```

В Laravel есть ещё один метод, который пытается удалить таблицу, только если она существует. Это делается с помощью SQL оператора DROP TABLE IF EXISTS. В Firebird нет оператора с подобным функционалом, однако мы можем эмулировать его с помощью анонимного блока (EXECUTE BLOCK + EXECUTE STATEMENT).

```
    /**
     * Compile a drop table (if exists) command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileDropIfExists(Blueprint $blueprint, Fluent $command) {
        $sql = 'EXECUTE BLOCK' . "\n";
        $sql .= 'AS' . "\n";
        $sql .= 'BEGIN' . "\n";
        $sql .= "  IF (EXISTS(select * from RDB\$RELATIONS where RDB\$RELATION_NAME = '" .
$blueprint->getTable() . "')) THEN" . "\n";
        $sql .= "    EXECUTE STATEMENT 'DROP TABLE " . $this->wrapTable($blueprint) . "';" .
"\n";
        $sql .= 'END';
        return $sql;
    }
```

Теперь добавим методы для добавления и удаления столбцов, ограничений и индексов.

```
    /**
     * Compile a column addition command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileAdd(Blueprint $blueprint, Fluent $command) {
        $table = $this->wrapTable($blueprint);

        $columns = $this->prefixArray('add column', $this->getColumns($blueprint));

        return 'alter table ' . $table . ' ' . implode(', ', $columns);
    }

    /**
     * Compile a primary key command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compilePrimary(Blueprint $blueprint, Fluent $command) {
        $columns = $this->columnize($command->columns);
```

```
        return 'alter table ' . $this->wrapTable($blueprint) . " add primary key ({$columns})";
    }

    /**
     * Compile a unique key command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileUnique(Blueprint $blueprint, Fluent $command) {
        $table = $this->wrapTable($blueprint);

        $index = $this->wrap($command->index);

        $columns = $this->columnize($command->columns);

        return "alter table $table add constraint {$index} unique ($columns)";
    }

    /**
     * Compile a plain index key command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileIndex(Blueprint $blueprint, Fluent $command) {
        $columns = $this->columnize($command->columns);

        $index = $this->wrap($command->index);

        return "create index {$index} on " . $this->wrapTable($blueprint) . " ({$columns})";
    }

    /**
     * Compile a drop column command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileDropColumn(Blueprint $blueprint, Fluent $command) {
        $columns = $this->prefixArray('drop column', $this->wrapArray($command->columns));

        $table = $this->wrapTable($blueprint);

        return 'alter table ' . $table . ' ' . implode(', ', $columns);
    }

    /**
     * Compile a drop primary key command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileDropPrimary(Blueprint $blueprint, Fluent $command) {
        $table = $blueprint->getTable();

        $index = $this->wrap("{$table}_pkey");

        return 'alter table ' . $this->wrapTable($blueprint) . " drop constraint {$index}";
    }

    /**
     * Compile a drop unique key command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileDropUnique(Blueprint $blueprint, Fluent $command) {
        $table = $this->wrapTable($blueprint);

        $index = $this->wrap($command->index);

        return "alter table {$table} drop constraint {$index}";
    }
```

```
    /**
     * Compile a drop index command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileDropIndex(Blueprint $blueprint, Fluent $command) {
        $index = $this->wrap($command->index);

        return "drop index {$index}";
    }

    /**
     * Compile a drop foreign key command.
     *
     * @param  \Illuminate\Database\Schema\Blueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileDropForeign(Blueprint $blueprint, Fluent $command) {
        $table = $this->wrapTable($blueprint);

        $index = $this->wrap($command->index);

        return "alter table {$table} drop constraint {$index}";
    }
```

Вы можете проверить работоспособность наших классов, создав и запустив миграцию со следующим содержимым:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('users');
    }
}
```

В результате запуска с помощью команды

```
php artisan migrate
```

будет создана таблица со следующим DDL:

```
CREATE TABLE "users" (
    "id"              INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    "name"            VARCHAR(255) NOT NULL,
```

```
    "email"           VARCHAR(255) NOT NULL,
    "password"        VARCHAR(255) NOT NULL,
    "remember_token"  VARCHAR(100),
    "created_at"      TIMESTAMP,
    "updated_at"      TIMESTAMP
);

ALTER TABLE "users" ADD CONSTRAINT "users_email_unique" UNIQUE ("email");
```

Откатить миграцию можно с помощью команды

```
php artisan migrate:reset
```

Добавление поддержки последовательностей

По аналогии с классом Blueprint, предназначенным для генерирования запроса или набора запросов для поддержки создания, модификации и удаления метаданных таблицы, создадим класс SequenceBlueprint для поддержки тех же действий для последовательностей. Этот класс достаточно простой, приводить целиком я его не буду, только основные отличия от аналогичного класса Blueprint.

Последовательности в Firebird имеют следующие атрибуты: имя последовательности, начальное значение и инкремент, который используется оператором NEXT VALUE FOR. Последний атрибут доступен начиная с версии 3.0. Таким образом, наш класс будет содержать следующие свойства:

```
    /**
     * The sequence the blueprint describes.
     *
     * @var string
     */
    protected $sequence;


    /**
     * Initial sequence value
     *
     * @var int
     */
    protected $start_with = 0;

    /**
     * Increment for sequence
     *
     * @var int
     */
    protected $increment = 1;

    /**
     * Restart flag that indicates that the sequence should be reset
     *
     * @var bool
     */
    protected $restart = false;
```

Методы для получения значений и установки этих свойств элементарны, поэтому мы не будем приводить их. Приведу лишь метод restart, который будет использован при генерации предложения RESTART в операторе ALTER SEQUENCE.

```
    /**
     * Restart sequence and set initial value
     *
     * @param int $startWith
```

```
     */
    public function restart($startWith = null) {
        $this->restart = true;
        $this->start_with = $startWith;
    }
```

По аналогии с классом Blueprint если не даны команды create или drop, то выполняется команда alter sequence.

```
    /**
     * Determine if the blueprint has a create command.
     *
     * @return bool
     */
    protected function creating()
    {
        foreach ($this->commands as $command) {
            if ($command->name == 'createSequence') {
                return true;
            }
        }

        return false;
    }

    /**
     * Determine if the blueprint has a drop command.
     *
     * @return bool
     */
    protected function dropping()
    {
        foreach ($this->commands as $command) {
            if ($command->name == 'dropSequence') {
                return true;
            }
            if ($command->name == 'dropSequenceIfExists') {
                return true;
            }
        }

        return false;
    }

    /**
     * Add the commands that are implied by the blueprint.
     *
     * @return void
     */
    protected function addImpliedCommands()
    {
        if (($this->restart || ($this->increment !== 1)) &&
                ! $this->creating() &&
                ! $this->dropping()) {
            array_unshift($this->commands, $this->createCommand('alterSequence'));
        }
    }

    /**
     * Get the raw SQL statements for the blueprint.
     *
     * @param  \Illuminate\Database\Connection  $connection
     * @param  \Illuminate\Database\Schema\Grammars\Grammar  $grammar
     * @return array
     */
    public function toSql(Connection $connection, Grammar $grammar)
    {
        $this->addImpliedCommands();

        $statements = [];

        // Each type of command has a corresponding compiler function on the schema
        // grammar which is used to build the necessary SQL statements to build
        // the sequence blueprint element, so we'll just call that compilers function.
        foreach ($this->commands as $command) {
            $method = 'compile'.ucfirst($command->name);
```

```
            if (method_exists($grammar, $method)) {
                if (! is_null($sql = $grammar->$method($this, $command, $connection))) {
                    $statements = array_merge($statements, (array) $sql);
                }
            }
        }

        return $statements;
    }
```

Полный код класса Illuminate\Database\Schema\ SequenceBlueprint вы можете найти в исходных текстах приложенных к статье.

Теперь настало время вернуться к классу грамматики Illuminate\Database\Schema\Grammars\ FirebirdGrammar и добавить в него методы для создания операторов {CREATE | ALTER | DROP} SEQUENCE.

```
    /**
     * Compile a create sequence command.
     *
     * @param  \Illuminate\Database\Schema\SequenceBlueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileCreateSequence(SequenceBlueprint $blueprint, Fluent $command) {
        $sql = 'create sequence ';
        $sql .= $this->wrapSequence($blueprint);
        if ($blueprint->getInitialValue() !== 0) {
            $sql .= ' start with ' . $blueprint->getInitialValue();
        }
        if ($blueprint->getIncrement() !== 1) {
            $sql .= ' increment by ' . $blueprint->getIncrement();
        }
        return $sql;
    }

    /**
     * Compile a alter sequence command.
     *
     * @param  \Illuminate\Database\Schema\SequenceBlueprint  $blueprint
     * @param  \Illuminate\Support\Fluent  $command
     * @return string
     */
    public function compileAlterSequence(SequenceBlueprint $blueprint, Fluent $command) {
        $sql = 'alter sequence ';
        $sql .= $this->wrapSequence($blueprint);
        if ($blueprint->isRestart()) {
            $sql .= ' restart';
            if ($blueprint->getInitialValue() !== null) {
                $sql .= ' with ' . $blueprint->getInitialValue();
            }
        }
        if ($blueprint->getIncrement() !== 1) {
            $sql .= ' increment by ' . $blueprint->getIncrement();
        }
        return $sql;
    }

    /**
     * Compile a drop sequence command.
     *
     * @param \Illuminate\Database\Schema\SequenceBlueprint $blueprint
     * @param \Illuminate\Support\Fluent $command
     * @return string
     */
    public function compileDropSequence(SequenceBlueprint $blueprint, Fluent $command) {
        return 'drop sequence ' . $this->wrapSequence($blueprint);
    }

    /**
     * Compile a drop sequence command.
     *
     * @param \Illuminate\Database\Schema\SequenceBlueprint $blueprint
     * @param \Illuminate\Support\Fluent $command
     * @return string
```

```
     */
    public function compileDropSequenceIfExists(SequenceBlueprint $blueprint, Fluent $command) {
        $sql = 'EXECUTE BLOCK' . "\n";
        $sql .= 'AS' . "\n";
        $sql .= 'BEGIN' . "\n";
        $sql .= "  IF (EXISTS(select * from RDB\$GENERATORS where RDB\$GENERATOR_NAME = '" .
$blueprint->getSequence() . "')) THEN" . "\n";
        $sql .= "    EXECUTE STATEMENT 'DROP SEQUENCE " . $this->wrapSequence($blueprint) . "';"
. "\n";
        $sql .= 'END';
        return $sql;
    }

    /**
     * Wrap a sequence in keyword identifiers.
     *
     * @param  mixed   $sequence
     * @return string
     */
    public function wrapSequence($sequence) {
        if ($sequence instanceof SequenceBlueprint) {
            $sequence = $sequence->getSequence();
        }

        if ($this->isExpression($sequence)) {
            return $this->getValue($sequence);
        }

        return $this->wrap($this->tablePrefix . $sequence, true);
    }
```

Ну вот, теперь добавление поддержки последовательностей в миграции Laravel завершена. Давайте посмотрим, как это работает, для этого слегка модифицируем миграцию приведённую ранее.

```
class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        // выполнит оператор
        // CREATE SEQUENCE "seq_users_id"
        Schema::createSequence('seq_users_id');
        // выполнит операторы
        // CREATE TABLE "users" (
        //   "id"              INTEGER NOT NULL,
        //   "name"            VARCHAR(255) NOT NULL,
        //   "email"           VARCHAR(255) NOT NULL,
        //   "password"        VARCHAR(255) NOT NULL,
        //   "remember_token"  VARCHAR(100),
        //   "created_at"      TIMESTAMP,
        //   "updated_at"      TIMESTAMP
        // );
        // ALTER TABLE "users" ADD PRIMARY KEY ("id");
        // ALTER TABLE "users" ADD CONSTRAINT "users_email_unique" UNIQUE ("email");
        Schema::create('users', function (Blueprint $table) {
            //$table->increments('id');
            $table->integer('id')->primary();
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
        // выполнит оператор
        // ALTER SEQUENCE "seq_users_id" RESTART WITH 10 INCREMENT BY 5
        Schema::sequence('seq_users_id', function (SequenceBlueprint $sequence) {
            $sequence->increment(5);
            $sequence->restart(10);
        });
    }
```

```
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        // выполнит оператор
        // DROP SEQUENCE "seq_users_id"
        Schema::dropSequence('seq_users_id');
        // выполнит оператор
        // DROP TABLE "users"
        Schema::drop('users');
    }
}
```

Можно пойти дальше и сделать поддержку создания BEFORE INSERT триггера и последовательности (генератора) для поддержки автоинкрементных полей в Firebird 2.5 и ниже. Но в большинстве случаев достаточно получать следующее значение последовательности и передавать её в INSERT запрос.

Заключение

Приведённых изменений достаточно для разработки веб приложений с использованием СУБД Firebird с использование фреймворка Laravel. Конечно, было бы хорошо оформить это как пакет и подключать его для расширения функционала, так как это делается с остальными модулями Laravel.

В следующей статье я расскажу о том, как создать небольшое приложение с использованием Laravel и СУБД Firebird. Если вас заинтересовала интеграция Firebird в фрймворк Laravel или вы нашли ошибку, пишите в личку обязательно отвечу. Изменённые и добавленные в Laravel файлы для поддержки Firebird вы можете скачать отсюда.

> **Замечание**
>
> На момент написания статьи мне было не известно о существовании пакета github.com/jacquestvanzuydam/laravel-firebird. Я надеюсь, что статья всё равно полезна для понимания того что происходит внутри Laravel.