# knex

Monday, December 14, 2015 11:11 AM

## Altering a column to have a different type or default (reference)

Let's say you have a bunch of rows in a database and you want to add a "status" column to those. You want all existing rows to get a status of "STARTED", but all future rows to have to define their own "status" explicitly. Thus, you can't just set a default and leave it, otherwise future rows would get that default, but you also can't say "notNullable" or existing rows would violate that property.

Here's how to do this in Knex:

```
function alterSeasonsTable(knex) {
  return knex.schema.alterTable('seasons', (table) => {
    // Because there's no "status" column to begin with and seasons may
exist,
    // we want to set a default just to give the existing season a
value.
    table
      .enu('status', _.keys(SeasonStatus))
      .notNullable()
      .defaultTo(SeasonStatus.STARTED);

    // However, we don't want a default to STAY on this column, so we
alter it
    // again.
    table
      .enu('status', _.keys(SeasonStatus))
      .notNullable()
      .alter();
  });
}
```

## Transactions with async/await

Here's an example of the syntax for this (make sure your Node version is higher than 5 or whatever it was that added async/await):

```
        return this.knex.transaction(async (trx) => {
            const dbResp = await trx('users')
              .select('name')
              .where('id', '=', 1);

              console.log('dbResp: ' + JSON.stringify(dbResp));
        });
```

Note: if you wrap an "await" with try/catch that the internal catch isn't going to rollback a transaction; you need to throw for that: https://bpaste.net/show/ce6cc58b4f31 Alternatively: you can call "trx.rollback" if you're going to manually catch, but it's probably just best to throw and let Knex handle it.

## Primary index

I've gotten the advice that each database table should have a primary index. I had a table like this:

```
+-----------------+---------------------+------+-----+---------+-------+
| Field           | Type                | Null | Key | Default | Extra |
+-----------------+---------------------+------+-----+---------+-------+
| user_id         | bigint(20) unsigned | NO   | MUL | <null>  |       |
| is_attack       | tinyint(1)          | NO   |     | <null>  |       |
| highest_rating  | float(8,2)          | NO   |     | <null>  |       |
| lowest_rating   | float(8,2)          | NO   |     | <null>  |       |
| starting_rating | float(8,2)          | NO   |     | <null>  |       |
+-----------------+---------------------+------+-----+---------+-------+
```

Rows should be unique here by user_id and is_attack, so for me to do that, I call this Knex:

```
function setupForeignKeysForUsersLeagues(knex) {
  return knex.schema.table('users_leagues', (table) => {
    table
      .foreign('user_id')
      .references('id')
      .inTable('users');

    // This should be unique since users_leagues only represents the
current
    // season.
    table.primary(
      ['user_id', 'is_attack'],
      'users_leagues_user_id_and_is_attack_index'
    );
  });
}
```

That produces this table (notice the "PRI" next to user_id and is_attack):

```
+-----------------+---------------------+------+-----+---------+-------+
| Field           | Type                | Null | Key | Default | Extra |
+-----------------+---------------------+------+-----+---------+-------+
| user_id         | bigint(20) unsigned | NO   | PRI | <null>  |       |
| is_attack       | tinyint(1)          | NO   | PRI | <null>  |       |
| highest_rating  | float(8,2)          | NO   |     | <null>  |       |
| lowest_rating   | float(8,2)          | NO   |     | <null>  |       |
| starting_rating | float(8,2)          | NO   |     | <null>  |       |
+-----------------+---------------------+------+-----+---------+-------+
```

## Creating a stored procedure

This uses knex.raw, but it's worth noting that you can't use DELIMITER from knex (as it only seems to exist in particular client programs):

```
const procedure = `
CREATE PROCEDURE \`getAllUserNames\` (IN unusedInParameter INT)
BEGIN
SELECT name
FROM users;
END
`;
knex.schema
  .raw(procedure)
  .then(() => {
    console.log("Success");
  })
```

```
  .catch(error => {
    console.log("Failure", error);
  });
```

## Default a datetime to now

The documentation doesn't talk about this in depth, but you can do something like this:

```
table
  .dateTime('created_at')
  .notNullable()
  .defaultTo(knex.fn.now());
```

Note that when I tried this, I ended up getting my local time put into the database, whereas in JavaScript, if I specified a "new Date()", I'd get UTC (which I think was based on specifying "timezone":"z" in the connection string).

## where

Small note: "=" is assumed as the operator:

```
knex.where('id', userId); // fine and short
knex.where('id', '=', userId); // fine, but more verbose
```

## as

To select a column aliased to another name, just use "as" directly in the "select":

```
knex('ongoing_matches')
  .join('matches', 'matches.id', '=', 'ongoing_matches.match_id')
  .select('ongoing_matches.id', 'matches.defender_id as match_defender')
  .where('matches.attacker_id', '=', 1)
  .first()
```

Result: {"id":9,"match_defender":4}

### Joins

If you don't do this with "as" and you instead just "select()" (i.e. "select *"), then conflicting names from joins will be overwritten. In this case, my ongoing_matches and matches tables both have an "id" column, so every once in a while I would end up getting the wrong column.

Your solutions here are to alias the entire table as shown here or to select specific columns as shown in the code snippet above. It's apparently an antipattern to select * from a table in MySQL (probably for reasons like this and also performance/memory usage).

Note: MySQL has no problem returning two values with the same name (although you wouldn't be able to differentiate between the two):

```
SELECT `ongoing_matches`.`id`,
    `matches`.`id`
FROM `ongoing_matches`
INNER JOIN `matches` ON `matches`.`id` = `ongoing_matches`.`match_id`
WHERE `matches`.`attacker_id` = 1 LIMIT 1

+----+----+
| id | id |
```

```
+----+----+
| 12 | 42 |
+----+----+
```

JavaScript will choke on that though because object keys need to be unique.

This can be fixed by using a query like this:

```
SELECT `ongoing_matches`.`id` AS `ongoing_matches_id`,
    `matches`.`id` AS `matches_id`
FROM `ongoing_matches`
INNER JOIN `matches` ON `matches`.`id` = `ongoing_matches`.`match_id` LIMIT 1
```

Note: there are two ways to do this with Knex:

1. knex.raw with "as"

```
knex("ongoing_matches")
  .select([
    knex.raw("matches.id as match_id"),
    knex.raw("ongoing_matches.id as ongoing_match_id")
  ])
  .join("matches", "ongoing_matches.match_id", "=", "matches.id");
```

2. You can also use "identifier syntax" (reference)

```
knex("ongoing_matches")
  .select({
    match_id: "matches.id",
    ongoing_match_id: "ongoing_matches.id",
    start_time: "start_time"
  })
  .join("matches", "ongoing_matches.match_id", "=", "matches.id");
```

## ".first"

Let's say you have a function like this:

```
getUserByEmailAddress(emailAddress) {
    return this.knex('users')
        .select()
        .where('email_address', '=', emailAddress)
        .then((dbResp) => {
            // Return the user object.
            return dbResp[0];
        });
}
```

Instead of returning "dbResp[0];", you can use ".first()", which not only has the "dbResp[0]" built-in, but also adds a "limit 1" to the query (which could improve performance for certain requests):

```
knex('users').select().first().toString()
    'select * from `users` limit 1'
```

Thus, the final function looks like this:

```
getUserByEmailAddress(emailAddress) {
    return this.knex('users')
```

```
            .select()
            .where('email_address', '=', emailAddress)
            .first();
    }
```

## Query basics

10/25/2016

You can optionally add to a query like this if you want since it's just the builder pattern [where ".then" coerces it into a Promise]:

```
    getScriptsForUser(userId, scriptIds = undefined) {
            const query = this.knex('scripts')
                .andWhere('user_id', '=', userId);

            if (util.exists(scriptIds)) {
                query.whereIn('id', scriptIds);
            }

            return query;
        }
```

## INSERT INTO … SELECT

I didn't actually try this out, but here's an issue on how to do it:
https://github.com/tgriesser/knex/issues/1056

## knex.raw

10/17/2016

This can be helpful when you're trying to do something that doesn't exactly fit the paradigm that knex lays out for you. For example, when a user logged in, I wanted to update their number of logins and their last login time. I thought it would have taken an "update" and an "increments" in the query builder, but that didn't do what I expected (I think it only did the increments) **(UPDATE: update+increments doesn't work only because of a bug).** Instead, I ended up with this:

```
return this.knex('users')
  .where('id', '=', userId)
  .update({
    last_login_date: new Date(),
    num_logins: this.knex.raw('num_logins + 1')
  });
```

volatilemajesty: if you mark variables as ":id", ":instanceId"instead of "?", you can pass in an object. cleans things up

**Be very careful not to open yourself up to injection attacks by using proper bindings (reference)**
Here's an example using named bindings and some extra functions so that I can concatenate with a "log" string that keeps getting built up from null (hence the "COALESCE" (reference)):

```
    const details = 'you were banned';
    yield trx('bans')
      .whereIn('user_id', ids)
      .update({
        ban_details: trx.raw("CONCAT(COALESCE(ban_details, ''), :details)",
```

```
{
      details,
    }),
  });
```

## Upsert (reference)

This doesn't exist in Knex and the creators don't want it to exist (apparently because of how particular each database vendor works with upsert semantics). From what I can see, it seems like using "knex.raw" combined with "ON DUPLICATE KEY" (MySQL) or "ON CONFLICT" (PostgreSQL) is the best way to go.

*MySQL knex upsert example (this code is based on the code suggested here for PostgreSQL)*

```
/**
 * Inserts or updates in a single query.
 * @param  {string} options.table  - the name of the table to upsert
into
 * @param  {Object} options.object - the object to insert or update
 * @param  {?Knex} knex - if specified, this should represent an
existing
 * transaction if you want this to be rolled back/committed as part of
 * another set of queries, otherwise it will fall back to the instance-
level
 * Knex object.
 * @return {Promise}
 */
upsert({ table, object }, trx = this.knex) {
  // This specific setup will only work with MySQL/Maria due to the raw
SQL
  // syntax.
  assert(
    trx.client instanceof knexClientMySQL ||
      trx.client instanceof knexClientMaria
  );

  const insert = trx(table).insert(object);
  const update = trx
    .queryBuilder()
    .update(object)
    .toString();

  // In MySQL, "update" typically pairs with "SET", but it does NOT when
doing
  // an upsert, so we have to remove the "set".
  const mySqlUpdate = update.replace(/update\s+set/i, 'update ');

  return trx.raw(`? ON DUPLICATE KEY ${mySqlUpdate}`, [insert]);
}

const objToUpsert = {
  user_id: 1,
  is_attack: true,
  games_played: 6,
};

await upsert({
```

```
    table: 'users_leagues',
    object: objToUpsert
  });
```

Note: the return value from "upsert" looks something like this:

```
dbResp: [
 {
  "fieldCount": 0,
  "affectedRows": 2,
  "insertId": 0,
  "serverStatus": 2,
  "warningCount": 0,
  "message": "",
  "protocol41": true,
  "changedRows": 0
 },
 null
]
```

- insertId apparently only gets set if you have an auto-incremented primary key
- affectedRows is 2 in the case of an update (rather than an insert) due to this

### *Multiple queries in a transaction*

Originally, I thought that maybe I should do something like this:

```
const insertOrUpdateObject = { key, value: JSON.stringify(value) };

return this.knex.transaction(async (trx) => {
  const dbRowsAffected = await trx('overseer_config')
    .update(insertOrUpdateObject)
    .where({
      key,
    });

  if (dbRowsAffected === 0) {
    await trx('overseer_config').insert(insertOrUpdateObject);
  }
});
```

However, there's a potential race condition here even though there's a transaction in use. It is possible that each connection calls "update", gets back 0 rows, then each connection tries inserting, which would result in an error on the second transaction to run.

## Migrations

12/14/2015

- Note: it's a really good idea to take a back-up of a database before running migrations. Typically, migrations are going to alter tables in addition to adding new ones, and altering a table can't be rolled back in MySQL (reference). To back up a database, check out this note.
- knex migrations:
  - First, make your knexfile.js
    - knex init
  - Modify the knexfile.js to include your environments and connection information.
  - Set up your database. I never found out whether this is something you should do through knex or through the command line, so I just did it through the command line.

```
CREATE DATABASE knex_migration_db;
USE knex_migration_db;
GRANT ALL PRIVILEGES ON knex_migration_db.* TO 'Adam'@'localhost';
```

- Make the knex migration file (see next section for more notes on the structure of this)
  - knex migrate:make migration_name
    - If this fails with ENOENT, then you probably just need to make the "migrations" folder.
    - The "migration_name" is used in the file name; it's appended to the datetime-stamp of creation. I believe it's just a moniker for you so that you can say something like "v1_to_v2" or something.
    - **For Adam:** if you need to make a new migration for Bot Land locally, first call "run.cmd" in the Account Server to set all of the database connection options, then ctrl+C it and do the "knex migrate:make migration_name" command from the "database" folder.
- Modify the migration file that was created to include any setup you need, e.g. making tables, inserting rows, etc.
- Run the migrations
  - knex migrate:latest --env development
  - Remember: there is a configuration value in knexfile.js called migrations: {tableName: 'knex_migrations'}. This will create a knex_migrations table in your database if it doesn't already exist. That table keeps track of which migration files were run so that it can know where to pick up next time.
  - Note: for Bot Land, I wanted to use ES6, so I wrote an NPM script in the Account Server to do this:
    - "knex": "babel-node node_modules/knex/bin/cli.js --knexfile database/knexfile.js",
    - That means I have to run the migrations from the Account Server folder with "npm run knex -- migrate:latest"

- Migration files
  - You should define functions like this:
    - exports.up = function(knex, Promise) {};
    - exports.down = function(knex, Promise) {};
    - The "up" function defines how you upgrade from the last migration file to this one.
    - The "down" function lets you specify rollback behavior if you want. This is not necessarily an inverse of the "up" function, e.g. you don't need to drop a table that you created in "up". Instead, you can use this for custom functionality like creating a backup table, filling that with data, and THEN dropping the original table so that you've rolled back correctly.
  - Migrations are automatically run inside of transactions, but keep in mind that altering tables in MySQL cannot be rolled back (reference). That means that if you're going to be altering tables, and hit a failure, you would have to manually undo your changes just to be able to re-run the file. This is why it's good to take a back-up beforehand.

## ES6 migrations (reference)

I originally tried to use node-babel for this via an NPM script in package.json, but I ran into issues trying to use the migrations from the API in test code (knex.migrate.latest()), so I had to switch to babel-register:

1. npm install --save-dev babel-register
2. Modify knexfile.js to include require('babel-register');
3. Add this NPM script
   a. "knex": "node node_modules/knex/bin/cli.js --knexfile database/knexfile.js",
   b. Note that you can probably just say "knex --knexfile database/knexfile.js", but I have

"knex.cmd" in the same directory and Windows is acting very strange when it comes to writing it that way.

4. Make sure you have a babelrc
5. Make sure babel-register shows up in your test code somewhere since I think calling knex.migrate.latest directly bypasses the knexfile.js (not sure about this).

~~Based on what I read at the reference and what tollus says below, it sounds like I need to use Babel and then change the command line that gets invoked (which would involve changing Ansible scripts for Bot Land):~~

~~10:15 tollus: if you want to use es6/imports you have to call babel first, i did it with a npm script called 'knex' with 'babel-node node_modules/knex/bin/cli.js'~~
~~10:15 tollus: then i run 'npm run knex -- migrate:xyz'~~

~~What I did for this:~~

~~npm install --save-dev babel-cli~~

~~Add this to package.json (note: I had to specify the knexfile.js only because it was in a different folder):~~

~~"knex": "babel-node node_modules/knex/bin/cli.js --knexfile database/knexfile.js",~~

~~Modified Ansible to have the command look something like this:~~

~~command: npm run knex -- migrate:latest --env development~~
~~chdir={{ accountServerDir }}/account_server~~

## DateTime guidelines

1/21/2016

Knex handles conversions between MySQL DateTime and JavaScript Date objects. However, you should make sure you connect with timezone set to 'Z':

```
knex = require('knex')({
  client: 'mysql',
  connection: {
    host: databaseHost,
    user: databaseUser,
    password: databasePassword,
    timezone: 'Z',
    database: databaseName
  }
});
```

If you DID want a function to convert from JavaScript --> MySQL, you can use this code that I adapted from a StackOverflow answer:

```
/**
 * This is mostly used by convertJSDateToMySQLDateTime to pad number strings.
 * @param  {number} d
 * @return {string}
 */
function twoDigits(d) {
  if (d >= 0 && d < 10) {
    return '0' + d;
  }
```

```
    if (d < 0 && d > -10) {
      return '-0' + (-1 * d);
    }

    return d.toString();
  }

  /**
   * Converts a JavaScript Date object into the MySQL format for DateTime.
   *
   * E.g. the epoch turns into '1970-01-01 00:00:00'.
   * @param  {Date} date
   * @return {string}
   */
  export function convertJSDateToMySQLDateTime(date) {
    const year = date.getUTCFullYear();
    const month = twoDigits(1 + date.getUTCMonth());
    const dayOfMonth = twoDigits(date.getUTCDate());
    const hours = twoDigits(date.getUTCHours());
    const minutes = twoDigits(date.getUTCMinutes());
    const seconds = twoDigits(date.getUTCSeconds());

    return `${year}-${month}-${dayOfMonth} ${hours}:${minutes}:${seconds}`;

    // return `${date.getUTCFullYear()}-${twoDigits(1 + date.getUTCMonth())}` +
    //     `-${twoDigits(date.getUTCDate())} ${twoDigits(date.getUTCHours())}` +
    //     `:${twoDigits(date.getUTCMinutes())}:${twoDigits(date.getUTCSeconds())}`;
  };
```

## Comparing DateTime

Just do something like **this**:

```
    return this.knex
      .select(['id, def_estimated_skill'])
      .from('users')

      // They need to have a defense
      .whereNotNull('serialized_defense')

      // They need to have logged in recently
      .andWhere('last_login_date', '>', new Date(Date.now() - 30 * 24 * 60 * 60 * 1000))

      // Find users with the lowest difference in skill.
      .orderByRaw(`ABS(def_estimated_skill - ${atkEstimatedSkill}) ASC`)

      // We set the limit very high so that it's likely to find someone
      // who they haven't played against before. Ideally, this function
      // won't be called many times because we should cache the results
      // from every function like this in the Matchmaker.
      .limit(500);
```

# Troubleshooting

### ER_CANNOT_ADD_FOREIGN

If you run into errors like this:

> Error: ER_CANNOT_ADD_FOREIGN: Cannot add foreign key constraint

Type this into MySQL:

> SHOW ENGINE INNODB STATUS;

You'll see a section like this:

> ---
> LATEST FOREIGN KEY ERROR
> ---
> 2017-02-11 14:13:58 0x1034 Error in foreign key constraint of table botland/#sql-abc_146:
> foreign key (`game_version`) references `game_versions` (`id`) on delete SET NULL:
> You have defined a SET NULL condition though some of the
> columns are defined as NOT NULL.

In this particular case, I had "ON DELETE SET NULL" on a column that couldn't be null.

## Transactions are **not** "all or nothing" due to locking tables

As far as I can tell, this is just how MySQL works (reference). If you have something like this code, it will persist to disk immediately due to locking semantics with transactions:

```
async function transactionTest() {
  await knex.transaction(async trx => {
    await trx.raw('LOCK TABLES users WRITE');

    // Kick off some promises

    // Because an error is thrown, you would expect the whole
transaction to
    // roll back, but the lock causes certain things to be committed
anyway.
    throw new Error('fake error');

    // TODO: unlock tables if you ever make it past this error in non-
example code
  });
}
```

As for a solution, I didn't actually come up with one. I bypassed this by avoiding table-level locks altogether (which itself could only be done by never running the particular queries unless the database couldn't be modified via user-controlled actions (i.e. the servers are essentially offline)).