This crate offers:

- MySql database driver in pure rust;
- connection pool.

Features:

- macOS, Windows and Linux support;
- LS support via **nativetls** or **rustls**;
- MySql text protocol support, i.e. support of simple text queries and text result sets;
- MySql binary protocol support, i.e. support of prepared statements and binary result sets;
- support of multi-result sets;
- support of named parameters for prepared statements;
- optional per-connection cache of prepared statements;
- buffer pool;
- support of MySql packets larger than 2^24;
- support of Unix sockets and Windows named pipes;
- support of custom LOCAL INFILE handlers;
- support of MySql protocol compression;
- support of auth plugins:
    - **mysql_native_password** - for MySql prior to v8;
    - **caching_sha2_password** - for MySql v8 and higher.

Installation

Put the desired version of the crate into the `dependencies` section of your `Cargo.toml`:

```
[dependencies]
mysql = "*"
```

Example

```
use mysql::*;
use mysql::prelude::*;

#[derive(Debug, PartialEq, Eq)]
struct Payment {
    customer_id: i32,
    amount: i32,
    account_name: Option<String>,
}

fn main() -> std::result::Result<(), Box<dyn std::error::Error>> {
    let url = "mysql://root:password@localhost:3306/db_name";
    let pool = Pool::new(url)?;

    let mut conn = pool.get_conn()?;

    // Let's create a table for payments.
    conn.query_drop(
        r"CREATE TEMPORARY TABLE payment (
            customer_id int not null,
            amount int not null,
            account_name text
        )")?;

    let payments = vec![
        Payment { customer_id: 1, amount: 2, account_name: None },
        Payment { customer_id: 3, amount: 4, account_name: Some("foo".into()) },
        Payment { customer_id: 5, amount: 6, account_name: None },
        Payment { customer_id: 7, amount: 8, account_name: None },
        Payment { customer_id: 9, amount: 10, account_name: Some("bar".into()) },
    ];

    // Now let's insert payments to the database
    conn.exec_batch(
        r"INSERT INTO payment (customer_id, amount, account_name)
          VALUES (:customer_id, :amount, :account_name)",
        payments.iter().map(|p| params! {
            "customer_id" => p.customer_id,
            "amount" => p.amount,
            "account_name" => &p.account_name,
        })
    )?;

    // Let's select payments from database. Type inference should do the trick here.
    let selected_payments = conn
        .query_map(
            "SELECT customer_id, amount, account_name from payment",
            |(customer_id, amount, account_name)| {
```

```
                Payment { customer_id, amount, account_name }
            },
        )?;

    // Let's make sure, that `payments` equals to `selected_payments`.
    // Mysql gives no guaranties on order of returned rows
    // without `ORDER BY`, so assume we are lucky.
    assert_eq!(payments, selected_payments);
    println!("Yay!");

    Ok(())
}
```

Crate Features

- crate's features:
  - **native-tls** (enabled by default) – specifies **native-tls** as the TLS backend
  - **rustls-tls** (disabled by default) – specifies **rustls** as the TLS backend
  - **buffer-pool** (enabled by default) – enables buffer pooling
- external features enabled by default:
  - for the **flate2** crate (please consult **flate2** crate documentation for available features):
    - **flate2/zlib** (necessary) – **zlib** backend is chosed by default.
  - for the **mysql_common** crate (please consult **mysql_common** crate documentation for available features):
    - **mysql_common/bigdecimal03** – the **bigdecimal03** is enabled by default
    - **mysql_common/rust_decimal** – the **rust_decimal** is enabled by default
    - **mysql_common/time03** – the **time03** is enabled by default
    - **mysql_common/uuid** – the **uuid** is enabled by default
    - **mysql_common/frunk** – the **frunk** is enabled by default

Please note, that you'll need to reenable external features if you are using **default-features = false**:

```
[dependencies]
mysql = { version = "*", default-features = false, features = ["minimal", "rustls-tls"] }
mysql_common = { version = "*", default-features = false, features = ["bigdecimal03", "time03", "uuid"]}
```

Basic structures

Opts

This structure holds server host name, client username/password and other settings, that controls client behavior.

URL-based connection string

Note, that you can use URL-based connection string as a source of an **Opts** instance. URL schema must be **mysql**. Host, port and credentials, as well as query parameters, should be given in accordance with the RFC 3986.

Examples:

```
# mysql::doctest_wrapper!(__result, {
# use mysql::Opts;
let _ = Opts::from_url("mysql://localhost/some_db")?;
let _ = Opts::from_url("mysql://[::1]/some_db")?;
let _ = Opts::from_url("mysql://user:pass%20word@127.0.0.1:3307/some_db?")?;
# });
```

Supported URL parameters (for the meaning of each field please refer to the docs on **Opts** structure in the create API docs):

- **prefer_socket: true | false** - defines the value of the same field in the **Opts** structure;
- **tcp_keepalive_time_ms: u32** - defines the value (in milliseconds) of the **tcp_keepalive_time** field in the **Opts** structure;
- **tcp_keepalive_probe_interval_secs: u32** - defines the value of the **tcp_keepalive_probe_interval_secs** field in the **Opts** structure;

- **tcp_keepalive_probe_count: u32** - defines the value of the **tcp_keepalive_probe_count** field in the **Opts** structure;
- **tcp_connect_timeout_ms: u64** - defines the value (in milliseconds) of the **tcp_connect_timeout** field in the **Opts** structure;
- **tcp_user_timeout_ms** - defines the value (in milliseconds) of the **tcp_user_timeout** field in the **Opts** structure;
- **stmt_cache_size: u32** - defines the value of the same field in the **Opts** structure;
- **compress** - defines the value of the same field in the **Opts** structure. Supported value are:
  - **true** - enables compression with the default compression level;
  - **fast** - enables compression with "fast" compression level;
  - **best** - enables compression with "best" compression level;
  - **1..9** - enables compression with the given compression level.
- **socket** - socket path on UNIX, or pipe name on Windows.

OptsBuilder

It's a convenient builder for the **Opts** structure. It defines setters for fields of the **Opts** structure.

```
# mysql::doctest_wrapper!(__result, {
# use mysql::*;
let opts = OptsBuilder::new()
    .user(Some("foo"))
    .db_name(Some("bar"));
let _ = Conn::new(opts)?;
# });
```

Conn

This structure represents an active MySql connection. It also holds statement cache and metadata for the last result set.

Conn's destructor will gracefully disconnect it from the server.

Transaction

It's a simple wrapper on top of a routine, that starts with **START TRANSACTION** and ends with **COMMIT** or **ROLLBACK**.

```
use mysql::*;
use mysql::prelude::*;

let pool = Pool::new(get_opts())?;
let mut conn = pool.get_conn()?;

let mut tx = conn.start_transaction(TxOpts::default())?;
tx.query_drop("CREATE TEMPORARY TABLE tmp (TEXT a)")?;
tx.exec_drop("INSERT INTO tmp (a) VALUES (?)", ("foo",))?;
let val: Option<String> = tx.query_first("SELECT a from tmp")?;
assert_eq!(val.unwrap(), "foo");
// Note, that transaction will be rolled back implicitly on Drop, if not committed.
tx.rollback();

let val: Option<String> = conn.query_first("SELECT a from tmp")?;
assert_eq!(val, None);
```

Pool

It's a reference to a connection pool, that can be cloned and shared between threads.

```
use mysql::*;
use mysql::prelude::*;

use std::thread::spawn;

let pool = Pool::new(get_opts())?;

let handles = (0..4).map(|i| {
    spawn({
        let pool = pool.clone();
        move || {
            let mut conn = pool.get_conn()?;
            conn.exec_first::<u32, _, _>("SELECT ? * 10", (i,))
                .map(Option::unwrap)
        }
    })
});

let result: Result<Vec<u32>> = handles.map(|handle| handle.join().unwrap()).collect();

assert_eq!(result.unwrap(), vec![0, 10, 20, 30]);
```

Statement

Statement, actually, is just an identifier coupled with statement metadata, i.e an information about its parameters and columns. Internally the `Statement` structure also holds additional data required to support named parameters (see bellow).

```
use mysql::*;
use mysql::prelude::*;

let pool = Pool::new(get_opts())?;
let mut conn = pool.get_conn()?;

let stmt = conn.prep("DO ?")?;

// The prepared statement will return no columns.
assert!(stmt.columns().is_empty());

// The prepared statement have one parameter.
let param = stmt.params().get(0).unwrap();
assert_eq!(param.schema_str(), "");
assert_eq!(param.table_str(), "");
assert_eq!(param.name_str(), "?");
```

Value

This enumeration represents the raw value of a MySql cell. Library offers conversion between `Value` and different rust types via `FromValue` trait described below.

FromValue trait

This trait is reexported from **mysql_common** create.

Trait offers conversion in two flavours:

- `from_value(Value) -> T` - convenient, but panicking conversion.
  Note, that for any variant of `Value` there exist a type, that fully covers its domain, i.e. for any variant of `Value` there exist `T: FromValue` such that `from_value` will never panic. This means, that if your database schema is known, than it's possible to write your application using only `from_value` with no fear of runtime panic.
- `from_value_opt(Value) -> Option<T>` - non-panicking, but less convenient conversion.
  This function is useful to probe conversion in cases, where source database schema is unknown.

```
use mysql::*;
use mysql::prelude::*;

let via_test_protocol: u32 = from_value(Value::Bytes(b"65536".to_vec()));
let via_bin_protocol: u32 = from_value(Value::UInt(65536));
assert_eq!(via_test_protocol, via_bin_protocol);

let unknown_val = // ...

// Maybe it is a float?
let unknown_val = match from_value_opt::<f64>(unknown_val) {
    Ok(float) => {
        println!("A float value: {}", float);
        return Ok(());
    }
    Err(FromValueError(unknown_val)) => unknown_val,
};

// Or a string?
let unknown_val = match from_value_opt::<String>(unknown_val) {
    Ok(string) => {
        println!("A string value: {}", string);
        return Ok(());
    }
    Err(FromValueError(unknown_val)) => unknown_val,
};

// Screw this, I'll simply match on it
match unknown_val {
    val @ Value::NULL => {
        println!("An empty value: {:?}", from_value::<Option<u8>>(val))
    },
    val @ Value::Bytes(..) => {
        // It's non-utf8 bytes, since we already tried to convert it to String
        println!("Bytes: {:?}", from_value::<Vec<u8>>(val))
    }
    val @ Value::Int(..) => {
        println!("A signed integer: {}", from_value::<i64>(val))
    }
    val @ Value::UInt(..) => {
        println!("An unsigned integer: {}", from_value::<u64>(val))
    }
    Value::Float(..) => unreachable!("already tried"),
    val @ Value::Double(..) => {
        println!("A double precision float value: {}", from_value::<f64>(val))
    }
    val @ Value::Date(..) => {
        use time::PrimitiveDateTime;
        println!("A date value: {}", from_value::<PrimitiveDateTime>(val))
    }
    val @ Value::Time(..) => {
        use std::time::Duration;
```

```
        println!("A time value: {:?}", from_value::<Duration>(val))
    }
}
```

Row

Internally **Row** is a vector of **Value**s, that also allows indexing by a column name/offset, and stores row metadata. Library offers conversion between **Row** and sequences of Rust types via **FromRow** trait described below.

FromRow trait

This trait is reexported from **mysql_common** create.

This conversion is based on the **FromValue** and so comes in two similar flavours:

- **from_row(Row) -> T** - same as **from_value**, but for rows;
- **from_row_opt(Row) -> Option<T>** - same as **from_value_opt**, but for rows.

Queryable trait offers implicit conversion for rows of a query result, that is based on this trait.

```
use mysql::*;
use mysql::prelude::*;

let mut conn = Conn::new(get_opts())?;

// Single-column row can be converted to a singular value:
let val: Option<String> = conn.query_first("SELECT 'foo'")?;
assert_eq!(val.unwrap(), "foo");

// Example of a mutli-column row conversion to an inferred type:
let row = conn.query_first("SELECT 255, 256")?;
assert_eq!(row, Some((255u8, 256u16)));

// The FromRow trait does not support to-tuple conversion for rows with more than 12 columns,
// but you can do this by hand using row indexing or `Row::take` method:
let row: Row = conn.exec_first("select 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12", ())?.unwrap();
for i in 0..row.len() {
    assert_eq!(row[i], Value::Int(i as i64));
}

// Another way to handle wide rows is to use HList (requires `mysql_common/frunk` feature)
use frunk::{HList, hlist, hlist_pat};
let query = "select 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15";
type RowType = HList!(u8, u16, u32, u8, u8, u8, u8, u8, u8, u8, u8, u8, u8, u8, u8, u8);
let first_three_columns = conn.query_map(query, |row: RowType| {
    // do something with the row (see the `frunk` crate documentation)
    let hlist_pat![c1, c2, c3, ...] = row;
    (c1, c2, c3)
});
assert_eq!(first_three_columns.unwrap(), vec![(0_u8, 1_u16, 2_u32)]);

// Some unknown row
let row: Row = conn.query_first(
    // ...
    # "SELECT 255, Null",
)?.unwrap();

for column in row.columns_ref() {
    // Cells in a row can be indexed by numeric index or by column name
    let column_value = &row[column.name_str().as_ref()];

    println!(
        "Column {} of type {:?} with value {:?}",
        column.name_str(),
        column.column_type(),
        column_value,
    );
}
```

Params

Represents parameters of a prepared statement, but this type won't appear directly in your code because binary protocol API will ask for **T: Into<Params>**, where **Into<Params>** is implemented:

- for tuples of \`Into<Value>\` types up to arity 12;
  Note: singular tuple requires extra comma, e.g. **("foo",)**;
- for **IntoIterator<Item: Into<Value>>** for cases, when your statement takes more than 12 parameters;
- for named parameters representation (the value of the **params!** macro, described below).

```
use mysql::*;
use mysql::prelude::*;

let mut conn = Conn::new(get_opts())?;

// Singular tuple requires extra comma:
let row: Option<u8> = conn.exec_first("SELECT ?", (0,))?;
assert_eq!(row.unwrap(), 0);

// More than 12 parameters:
```

```
let row: Option<u8> = conn.exec_first(
    "SELECT CONVERT(? + ? + ? + ? + ? + ? + ? + ? + ? + ? + ? + ? + ? + ? + ? + ?, UNSIGNED)",
    (0..16).collect::<Vec<_>>(),
)?;
assert_eq!(row.unwrap(), 120);
```

Note: Please refer to the **mysql_common** crate docs for the list of types, that implements **Into<Value>**.

Serialized, Deserialized

Wrapper structures for cases, when you need to provide a value for a JSON cell, or when you need to parse JSON cell as a struct.

```
use mysql::*;
use mysql::prelude::*;

/// Serializable structure.
#[derive(Debug, PartialEq, Serialize, Deserialize)]
struct Example {
    foo: u32,
}

// Value::from for Serialized will emit json string.
let value = Value::from(Serialized(Example { foo: 42 }));
assert_eq!(value, Value::Bytes(br#"{"foo":42}"#.to_vec()));

// from_value for Deserialized will parse json string.
let structure: Deserialized<Example> = from_value(value);
assert_eq!(structure, Deserialized(Example { foo: 42 }));
```

QueryResult

It's an iterator over rows of a query result with support of multi-result sets. It's intended for cases when you need full control during result set iteration. For other cases **Queryable** provides a set of methods that will immediately consume the first result set and drop everything else.

This iterator is lazy so it won't read the result from server until you iterate over it. MySql protocol is strictly sequential, so **Conn** will be mutably borrowed until the result is fully consumed.

```
use mysql::*;
use mysql::prelude::*;

let mut conn = Conn::new(get_opts())?;

// This query will emit two result sets.
let mut result = conn.query_iter("SELECT 1, 2; SELECT 3, 3.14;")?;

let mut sets = 0;
while let Some(result_set) = result.iter() {
    sets += 1;

    println!("Result set columns: {:?}", result_set.columns());
    println!(
        "Result set meta: {}, {:?}, {} {}",
        result_set.affected_rows(),
        result_set.last_insert_id(),
        result_set.warnings(),
        result_set.info_str(),
    );

    for row in result_set {
        match sets {
            1 => {
                // First result set will contain two numbers.
                assert_eq!((1_u8, 2_u8), from_row(row?));
            }
            2 => {
                // Second result set will contain a number and a float.
                assert_eq!((3_u8, 3.14), from_row(row?));
            }
            _ => unreachable!(),
        }
    }
}

assert_eq!(sets, 2);
```

Text protocol

MySql text protocol is implemented in the set of **Queryable::query** methods. It's useful when your query doesn't have parameters.

Note: All values of a text protocol result set will be encoded as strings by the server, so **from_value** conversion may lead to additional parsing costs.

Examples:

```
let pool = Pool::new(get_opts())?;
```

```
let val = pool.get_conn()?.query_first("SELECT POW(2, 16)")?;

// Text protocol returns bytes even though the result of POW
// is actually a floating point number.
assert_eq!(val, Some(Value::Bytes("65536".as_bytes().to_vec())));
```

The `TextQuery` trait.

The **TextQuery** trait covers the set of **Queryable::query** methods from the perspective of a query, i.e. **TextQuery** is something, that can be performed if suitable connection is given. Suitable connections are:

- **&Pool**
- **Conn**
- **PooledConn**
- **&mut Conn**
- **&mut PooledConn**
- **&mut Transaction**

The unique characteristic of this trait, is that you can give away the connection and thus produce **QueryResult** that satisfies **'static**:

```
use mysql::*;
use mysql::prelude::*;

fn iter(pool: &Pool) -> Result<impl Iterator<Item=Result<u32>>> {
    let result = "SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3".run(pool)?;
    Ok(result.map(|row| row.map(from_row)))
}

let pool = Pool::new(get_opts())?;

let it = iter(&pool)?;

assert_eq!(it.collect::<Result<Vec<u32>>>()?, vec![1, 2, 3]);
```

Binary protocol and prepared statements.

MySql binary protocol is implemented in **prep**, **close** and the set of **exec*** methods, defined on the **Queryable** trait. Prepared statements is the only way to pass rust value to the MySql server. MySql uses **?** symbol as a parameter placeholder and it's only possible to use parameters where a single MySql value is expected. For example:

```
let pool = Pool::new(get_opts())?;
let val = pool.get_conn()?.exec_first("SELECT POW(?, ?)", (2, 16))?;

assert_eq!(val, Some(Value::Double(65536.0)));
```

Statements

In MySql each prepared statement belongs to a particular connection and can't be executed on another connection. Trying to do so will lead to an error. The driver won't tie statement to its connection in any way, but one can look on to the connection id, contained in the **Statement** structure.

```
let pool = Pool::new(get_opts())?;

let mut conn_1 = pool.get_conn()?;
let mut conn_2 = pool.get_conn()?;

let stmt_1 = conn_1.prep("SELECT ?")?;

// stmt_1 is for the conn_1, ..
assert!(stmt_1.connection_id() == conn_1.connection_id());
assert!(stmt_1.connection_id() != conn_2.connection_id());

// .. so stmt_1 will execute only on conn_1
assert!(conn_1.exec_drop(&stmt_1, ("foo",)).is_ok());
assert!(conn_2.exec_drop(&stmt_1, ("foo",)).is_err());
```

Statement cache

Conn will manage the cache of prepared statements on the client side, so subsequent calls to prepare with the same statement won't lead to a client-server roundtrip. Cache size for each connection is determined by the **stmt_cache_size** field of the **Opts** structure. Statements, that are out of this boundary will be closed in LRU order.

Statement cache is completely disabled if **stmt_cache_size** is zero.

Caveats:

- disabled statement cache means, that you have to close statements yourself using **Conn::close**, or they'll exhaust server limits/resources;
- you should be aware of the **max_prepared_stmt_count** option of the MySql server. If the number of active connections times the value of **stmt_cache_size** is greater, than you could receive an error while prepareing another statement.

Named parameters

MySql itself doesn't have named parameters support, so it's implemented on the client side. One should use **:name** as a placeholder syntax for a named parameter. Named parameters uses the following naming convention:

- parameter name must start with either _ or **a..z**
- parameter name may continue with _, **a..z** and **0..9**

Named parameters may be repeated within the statement, e.g **SELECT :foo, :foo** will require a single named parameter **foo** that will be repeated on the corresponding positions during statement execution.

One should use the **params!** macro to build parameters for execution.

Note: Positional and named parameters can't be mixed within the single statement.

Examples:

```
let pool = Pool::new(get_opts())?;

let mut conn = pool.get_conn()?;
let stmt = conn.prep("SELECT :foo, :bar, :foo")?;

let foo = 42;

let val_13 = conn.exec_first(&stmt, params! { "foo" => 13, "bar" => foo })?.unwrap();
// Short syntax is available when param name is the same as variable name:
let val_42 = conn.exec_first(&stmt, params! { foo, "bar" => 13 })?.unwrap();

assert_eq!((foo, 13, foo), val_42);
assert_eq!((13, foo, 13), val_13);
```

Buffer pool

Crate uses the global lock-free buffer pool for the purpose of IO and data serialization/deserialization, that helps to avoid allocations for basic scenarios. You can control it's characteristics using the following environment variables:

- **RUST_MYSQL_BUFFER_POOL_CAP** (defaults to 128) – controls the pool capacity. Dropped buffer will be immediately deallocated if the pool is full. Set it to `0` to disable the pool at runtime.
- **RUST_MYSQL_BUFFER_SIZE_CAP** (defaults to 4MiB) – controls the maximum capacity of a buffer stored in the pool. Capacity of a dropped buffer will be shrunk to this value when buffer is returned to the pool.

To completely disable the pool (say you are using jemalloc) please remove the **buffer-pool** feature from the set of default crate features.

BinQuery and BatchQuery traits.

BinQuery and **BatchQuery** traits covers the set of **Queryable::exec*** methods from the perspective of a query, i.e. **BinQuery** is something, that can be performed if suitable connection is given.

As with the **TextQuery** you can give away the connection and acquire **QueryResult** that satisfies **'static**.

BinQuery is for prepared statements, and prepared statements requires a set of parameters, so **BinQuery** is implemented for **QueryWithParams** structure, that can be acquired, using **WithParams** trait.

Example:
```
use mysql::*;
use mysql::prelude::*;

let pool = Pool::new(get_opts())?;

let result: Option<(u8, u8, u8)> = "SELECT ?, ?, ?"
    .with((1, 2, 3)) // <- WithParams::with will construct an instance of QueryWithParams
    .first(&pool)?;  // <- QueryWithParams is executed on the given pool

assert_eq!(result.unwrap(), (1, 2, 3));
```

The **BatchQuery** trait is a helper for batch statement execution. It's implemented for **QueryWithParams** where parameters is an iterator over parameters:
```
use mysql::*;
use mysql::prelude::*;

let pool = Pool::new(get_opts())?;
let mut conn = pool.get_conn()?;

"CREATE TEMPORARY TABLE batch (x INT)".run(&mut conn)?;
"INSERT INTO batch (x) VALUES (?)"
    .with((0..3).map(|x| (x,))) // <- QueryWithParams constructed with an iterator
    .batch(&mut conn)?;         // <- batch execution is preformed here

let result: Vec<u8> = "SELECT x FROM batch".fetch(conn)?;

assert_eq!(result, vec![0, 1, 2]);
```

Queryable

The **Queryable** trait defines common methods for **Conn**, **PooledConn** and **Transaction**. The set of basic methods consts of:

- **query_iter** - basic methods to execute text query and get **QueryResult**;
- **prep** - basic method to prepare a statement;
- **exec_iter** - basic method to execute statement and get **QueryResult**;
- **close** - basic method to close the statement;

The trait also defines the set of helper methods, that is based on basic methods. These methods will consume only the first result set, other result sets will be dropped:
{query|exec} - to collect the result into a **Vec<T: FromRow>**;
{query|exec}_first - to get the first **T: FromRow**, if any;
{query|exec}_map - to map each **T: FromRow** to some **U**;
{query|exec}_fold - to fold the set of **T: FromRow** to a single value;
{query|exec}_drop - to immediately drop the result.

The trait also defines the **exec_batch** function, which is a helper for batch statement execution.

SSL Support

SSL support comes in two flavors:

1. Based on **native-tls –** this is the default option, that usually works without pitfalls.
2. Based on **rustls –** TLS backend written in Rust. Please use the **rustls-tls** crate feature to enable it.
   Please also note a few things about **rustls**:
   - it will fail if you'll try to connect to the server by its IP address, hostname is required;
   - it, most likely, won't work on windows, at least with default server certs, generated by the MySql installer.