

ПРОГРАММИРОВАНИЕ В INTERNET

ORM. SEQUELIZE. PRISMA

ORM =
Object Relational Mapping

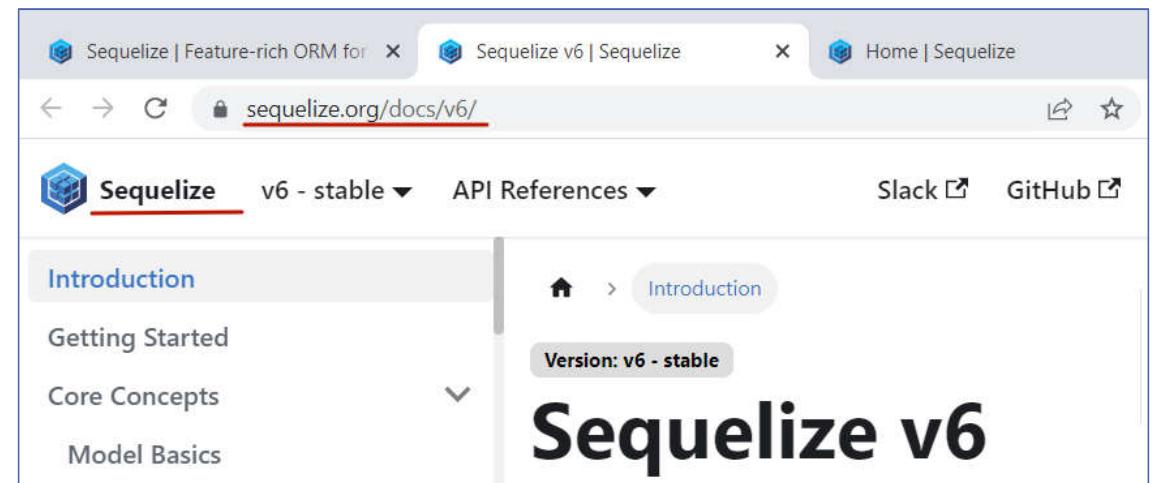
технология программирования,
которая позволяет работать с
SQL-базой данных, **как с**
набором программных
объектов.

Mapping: база данных – объект
контекста, таблица – коллекция
объектов, строка в таблице –
объект, структура таблицы –
класс.

Sequelize

=

это **ORM-библиотека**, основанная на промисах, **для приложений на Node.js**, которая осуществляет сопоставление таблиц в БД и отношений между ними с классами. Может применяться для: Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server, Oracle, DB2, Snowflake. Поддерживает TS и JS.



[документация](#)

Для работы с sequelize необходимо установить 2 пакета:

- пакет sequelize;

```
npm install --save sequelize
```

- драйвер базы данных.

```
# One of the following:  
$ npm install --save pg pg-hstore # Postgres  
$ npm install --save mysql2  
$ npm install --save mariadb  
$ npm install --save sqlite3  
$ npm install --save tedious # Microsoft SQL Server  
$ npm install --save oracledb # Oracle Database
```

Установка и закрытие соединения с БД

```
const Sequelize = require('sequelize');      // npm install sequelize, npm install tedious
const sequelize = new Sequelize('NodeJSTest','student','fitfit',{host:'172.16.193.223',dialect:'mssql'});

sequelize.authenticate()                      // проверка соединения
.then(()  => {
  console.log('Соединение с базой данных установлено');

  sequelize.close();
})
.catch(err => {console.log('Ошибка при соединении с базой данных', err);});
```

Для проверки соединения можно использовать метод **authenticate()**.

Sequelize будет держать соединение открытым по умолчанию и использовать одно и то же соединение для всех запросов. Если нужно **закрыть соединение**, то необходимо вызвать метод **close()**

Для подключения к базе данных прежде всего необходимо **создать объект Sequelize**.

Для создания объекта sequelize используется класс Sequelize, конструктор которого принимает ряд параметров:

constructor(database, username, password[, options])

Параметр options – объект, который может содержать следующие свойства: host, port, dialect, dialectOptions, define (freezeTableName, modelName, scopes, timestamps, hooks), schema, sync, logging, ssl, pool (max, min,...), hooks,...

Модель

=

это абстракция, представляющая **таблицу** в базе данных.

Модель сообщает несколько вещей о сущности, которую она представляет, например, **имя таблицы** в базе данных и какие **столбцы** у нее есть (и **их типы данных**). У модели есть имя. Это имя не обязательно должно совпадать с именем таблицы, которую она представляет в базе данных. Обычно модели имеют имена в единственном числе (например, User), а таблицы – имена во множественном числе (например, Users).

Есть 2 способа создания модели:

- 1) расширение **Model** и вызов метода **init (attributes[, options])**

```
const Sequelize = require('sequelize');      // npm install sequelize, npm install tedious
const Model    = Sequelize.Model;
const sequelize = new Sequelize('SeqTest','student','fitfit',{host:'172.16.193.223',dialect:'mssql'});

class Faculty extends Model{};
Faculty.init(
{
  faculty: {type: Sequelize.STRING, allowNull:false, primaryKey: true},
  faculty_name:{type: Sequelize.STRING, allowNull:false}
},{  
  sequelize,  
  modelName:'Faculty',           // имя модели  
  tableName:'Faculty',          // имя таблицы, по умолчанию был бы 'faculties'  
  timestamps: false             // не использовать updateAt insertAt
})
```

По умолчанию, если имя таблицы не указано, **Sequelize автоматически формирует множественное число для имени модели и использует его в качестве имени таблицы**. Имя можно заморозить **options.freezeTableName: true** (название таблицы == имени модели), а можно задать явно в **options.tableName**.

Есть 2 способа создания модели:

- 2) вызов `sequelize.define (modelName, attributes[, options])`

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('SeqTest', 'student', 'fitfit', { host: '172.16.193.223', dialect: 'mssql' })

const Faculty = sequelize.define('Faculty', { // имя модели
    faculty: { type: Sequelize.STRING, allowNull: false, primaryKey: true },
    faculty_name: { type: Sequelize.STRING, allowNull: false }
}, {
    sequelize,
    tableName: 'Faculty', // имя таблицы
    timestamps: false // не использовать updateAt, insertAt
});
```

Внутри `sequelize.define` вызывает `Model.init`, поэтому оба подхода по сути эквивалентны.

Внутри `sequelize.define` вызывает `Model.init`, поэтому оба подхода по сути эквивалентны.

Значение параметра attributes.column (описание столбца таблицы)

- **allowNull** – если false, столбец будет иметь ограничение NOT NULL.
- **defaultValue** – значение по умолчанию для столбца.
- **unique** – если true, то есть все значения в столбце должны быть уникальными. Если указана строка, столбец будет частью составного уникального индекса.
- **primaryKey** – если true, этот атрибут будет помечен как первичный ключ.
- **field** – если установлено, sequelize сопоставит имя столбца с именем в базе данных.
- **autoIncrement** – если true, то для этого столбца будет установлено автоинкрементирование.

Значение параметра `attributes.column` (описание столбца таблицы)

- `comment` – комментарий к этому столбцу.
- `references` – объект с конфигурацией связей на другие модели (`model`, `key`).
- `onUpdate` – что должно произойти при обновлении связанного ключа. Варианты: CASCADE, RESTRICT, SET DEFAULT, SET NULL или NO ACTION.
- `onDelete` – что должно произойти, когда связанный ключ будет удален. Варианты: CASCADE, RESTRICT, SET DEFAULT, SET NULL или NO ACTION.
- `validate` – объект проверки для этого столбца при каждом сохранении модели.

[Подробнее здесь](#)

Значение параметра options

- **sequelize** – экземпляр sequelize, чтобы присоединить его к новой модели.
- **modelName** – название модели. По умолчанию это то же самое, что и имя класса.
- **defaultScope** – scope по умолчанию для модели.
- **scopes** – дополнительные scope для модели.
- **timestamps** – добавляет временные метки createdAt и updatedAt в модель.
- **paranoid** – если true, то вызов destroy не удалит модель, а вместо этого установит временную метку deleteAt.
- **freezeTableName** – если true, то sequelize не будет пытаться изменить имя модели, чтобы получить имя таблицы, то есть имя модели и имя таблицы будут одинаковыми, в противном случае в качестве названия таблицы будет использоваться название модели во множественном числе.

Значение параметра options

- **indexes** – определение индексов.
- **createdAt / updatedAt / deletedAt** – если указана строка, то переопределяется имя столбца createdAt, или, если false, этот столбец не добавляется.
- **tableName** – в качестве имени таблицы будет использоваться то название, что здесь указано.
- **schema** – имя схемы.
- **charset** – кодировка таблицы модели.
- **comment** – комментарий к таблице модели.
- **hooks** – объект с hook-функциями, которые вызываются до и после определенных событий жизненного цикла.
- **validate** – объект широкой проверки модели. Валидации имеют доступ ко всем значениям модели через this.

Использование типов данных sequelize

Для использования типов данных можно:

- 1) Использовать DataTypes

```
const { DataTypes } = require('sequelize');

sequelize.define('model', {
  column: DataTypes.INTEGER
})
```

- 2) Использовать полностью подключенный пакет sequelize

```
const sequelize = require('sequelize');

sequelize.define('model', {
  column: sequelize.INTEGER
})
```

Типы данных (DataTypes)

Строки (STRING, STRING.BINARY, TEXT, CITEXT)

```
Sequelize.STRING          // VARCHAR(255)
Sequelize.STRING(1234)    // VARCHAR(1234)
Sequelize.STRING.BINARY   // VARCHAR BINARY
Sequelize.TEXT            // TEXT
Sequelize.TEXT('tiny')    // TINYTEXT
Sequelize.CITEXT          // CITEXT      PostgreSQL and SQLite only.
```

Дата (DATE, DATEONLY)

```
Sequelize.DATE           // DATETIME for mysql / sqlite, TIMESTAMP WITH TIME ZONE for postgres
Sequelize.DATE(6)         // DATETIME(6) for mysql 5.6.4+. Fractional seconds support with up to 6
Sequelize.DATEONLY        // DATE without time.
```

Логический (BOOLEAN)

```
Sequelize.BOOLEAN         // TINYINT(1)
```

Типы данных (DataTypes)

Числа (INTEGER, BIGINT, FLOAT, REAL, DOUBLE, DECIMAL)

```
Sequelize.INTEGER          // INTEGER  
Sequelize.BIGINT          // BIGINT  
Sequelize.BIGINT(11)       // BIGINT(11)
```

```
Sequelize.FLOAT            // FLOAT  
Sequelize.FLOAT(11)        // FLOAT(11)  
Sequelize.FLOAT(11, 10)     // FLOAT(11,10)
```

```
Sequelize.REAL              // REAL      PostgreSQL only.  
Sequelize.REAL(11)          // REAL(11)   PostgreSQL only.  
Sequelize.REAL(11, 12)       // REAL(11,12) PostgreSQL only.
```

```
Sequelize.DOUBLE            // DOUBLE  
Sequelize.DOUBLE(11)        // DOUBLE(11)  
Sequelize.DOUBLE(11, 10)     // DOUBLE(11,10)
```

```
Sequelize.DECIMAL           // DECIMAL  
Sequelize.DECIMAL(10, 2)     // DECIMAL(10,2)
```

Типы данных (DataTypes)

Перечисления (ENUM)

```
Sequelize.ENUM('value 1', 'value 2') // An ENUM with allowed values 'value 1' and 'value
```

Типы, специфичные для PostgreSQL

```
Sequelize.ARRAY(Sequelize.TEXT)      // Defines an array. PostgreSQL only.  
Sequelize.ARRAY(Sequelize.ENUM)      // Defines an array of ENUM. PostgreSQL only.  
  
Sequelize.BLOB                      // BLOB (bytea for PostgreSQL)  
Sequelize.BLOB('tiny')               // TINYBLOB (bytea for PostgreSQL. Other options are medium and large)  
  
Sequelize.UUID                      // UUID datatype for PostgreSQL and SQLite, CHAR(36) BINARY for MySQL  
  
Sequelize.CIDR                      // CIDR datatype for PostgreSQL  
Sequelize.INET                      // INET datatype for PostgreSQL  
Sequelize.MACADDR                   // MACADDR datatype for PostgreSQL  
  
Sequelize.RANGE(Sequelize.INTEGER)   // Defines int4range range. PostgreSQL only.  
Sequelize.RANGE(Sequelize.BIGINT)    // Defines int8range range. PostgreSQL only.  
Sequelize.RANGE(Sequelize.DATE)     // Defines tstzrange range. PostgreSQL only.  
Sequelize.RANGE(Sequelize.DATEONLY)  // Defines daterange range. PostgreSQL only.  
Sequelize.RANGE(Sequelize.DECIMAL)   // Defines numrange range. PostgreSQL only.  
  
Sequelize.GEOMETRY                  // Spatial column. PostgreSQL (with PostGIS) or MySQL only.  
Sequelize.GEOMETRY('POINT')         // Spatial column with geometry type. PostgreSQL (with PostGIS) or MySQL only.  
Sequelize.GEOMETRY('POINT', 4326)    // Spatial column with geometry type and SRID. PostgreSQL (with PostGIS) only.
```

(ARRAY, BLOB, UUID,
CIDR, INER,
MACADDR, RANGE,
GEOMETRY)

Ассоциации

Sequelize поддерживает стандартные ассоциации: One-To-One, One-To-Many и Many-To-Many.

Для этого Sequelize предоставляет **четыре типа ассоциаций**, которые следует комбинировать для их создания:

- **HasOne**
- **BelongsTo**
- **HasMany**
- **BelongsToMany**

Создание ассоциаций

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

AhasOne(B, { /* options */ });
AbelongsTo(B, { /* options */ });
AhasMany(B, { /* options */ });
A.belongsToMany(B, { through: 'C', /* options */ }); // через промежуточную таблицу C
```

А является **исходной моделью (source)**, а В — **целевой моделью (target)**.

A.hasOne(B) означает, что между А и В существует отношение «один к одному».

A.belongsTo(B) означает, что между А и В существует отношение «один к одному».

AhasMany(B) означает, что между А и В существует связь «один ко многим».

A.belongsToMany(B, {through: 'C'}) означает, что между А и В отношение «многие ко многим», использующее таблицу С в качестве соединительной таблицы (создастся, если нет), которая будет иметь внешние ключи (aId и bId).

has... => внешний ключ в **целевой модели**

belongTo... => внешний ключ **в исходной модели**

Создание стандартных ассоциаций

Обычно **ассоциации** Sequelize
используются параметрами для создания
стандартных ассоциаций:

- для создания отношения «один к одному» ассоциации **hasOne + belongsTo**;
- для создания отношения «один ко многим» ассоциации **hasMany + belongsTo**;
- для создания отношения «многие ко многим» используются **два вызова belongsToMany**.

```
// один-ко-многим
Foo.hasOne(Bar, {
  foreignKey: 'myFooId', // по умолчанию название исх. модели + Id
  onDelete: 'RESTRICT',
  onUpdate: 'RESTRICT', // RESTRICT, CASCADE, NO ACTION, SET DEFAULT and SET NULL
  as: 'foobar'          // псевдоним ассоциации, если нет вн.кл., то будет foobarId
});

Bar.belongsTo(Foo, {
  foreignKey: {
    name: 'myFooId', // вн. ключ в Foo, по умолч. название цел. модели + Id
    type: DataTypes.UUID,
    allowNull: false,
    ...
  }
});
```

Когда связь определена между двумя моделями, экземпляры этих моделей получают специальные методы для взаимодействия со своими связанными моделями.

`FoohasOne(Bar)`

`Foo.belongs(Bar)`

- `foolInstance.getBar()`
- `foolInstance.setBar()`
- `foolInstance.createBar()`

`FoohasMany(Bar)`

`Foo.belongsToMany(Bar, { through: Baz })`

- `foolInstance.getBars()`
- `foolInstance.countBars()`
- `foolInstance.hasBar()`
- `foolInstance.hasBars()`
- `foolInstance.setBars()`
- `foolInstance.addBar()`
- `foolInstance.addBars()`
- `foolInstance.removeBar()`
- `foolInstance.removeBars()`
- `foolInstance.createBar()`

Модели для примеров

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define('Faculty', {
    faculty: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      primaryKey: true
    },
    faculty_name: {
      type: DataTypes.STRING(50),
      allowNull: true
    }
  }, {
    sequelize,
    tableName: 'Faculty',
    timestamps: false
  });
};
```

Faculty.js

```
... module.exports = function(sequelize, DataTypes) {
  return sequelize.define('Pulpit', {
    pulpit: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      primaryKey: true
    },
    pulpit_name: {
      type: DataTypes.STRING(100),
      allowNull: true
    },
    faculty: {
      type: DataTypes.CHAR(10),
      allowNull: true,
      references: {
        model: 'Faculty',
        key: 'faculty'
      }
    },
    {
      sequelize,
      tableName: 'Pulpit',
      timestamps: false
    });
};
```

Pulpit.js

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define('Teacher', {
    teacher: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      primaryKey: true
    },
    teacher_name: {
      type: DataTypes.STRING(50),
      allowNull: true
    },
    pulpit: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      references: {
        model: 'Pulpit',
        key: 'pulpit'
      }
    },
    {
      sequelize,
      tableName: 'Teacher',
      timestamps: false
    });
};
```

Teacher.js

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define('Subject', {
    subject: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      primaryKey: true
    },
    subject_name: {
      type: DataTypes.STRING(50),
      allowNull: false
    },
    pulpit: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      references: {
        model: 'Pulpit',
        key: 'pulpit'
      }
    },
    {
      sequelize,
      tableName: 'Subject',
      timestamps: false
    });
};
```

Subject.js

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define('Auditorium_type', {
    auditorium_type: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      primaryKey: true
    },
    auditorium_typename: {
      type: DataTypes.STRING(30),
      allowNull: false
    }
  }, {
    sequelize,
    tableName: 'Auditorium_type',
    timestamps: false
  });
};
```

Auditorium_type.js

```
module.exports = function (sequelize, DataTypes) {
  return sequelize.define('Auditorium', {
    auditorium: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      primaryKey: true
    },
    auditorium_name: {
      type: DataTypes.STRING(200),
      allowNull: true
    },
    auditorium_capacity: {
      type: DataTypes.INTEGER,
      allowNull: true
    },
    auditorium_type: {
      type: DataTypes.CHAR(10),
      allowNull: false,
      references: {
        model: 'Auditorium_type',
        key: 'auditorium_type'
      }
    },
    sequelize,
    tableName: 'Auditorium',
    timestamps: false
  });
};
```

Auditorium.js

КОНТЕКСТ

```
const DataTypes = require("sequelize").DataTypes;
const _Auditorium = require("./Auditorium");
const _Auditorium_type = require("./Auditorium_type");
const _Faculty = require("./Faculty");
const _Pulpit = require("./Pulpit");
const _Subject = require("./Subject");
const _Teacher = require("./Teacher");

function initModels(sequelize) {
  let Auditorium = _Auditorium(sequelize, DataTypes);
  let Auditorium_type = _Auditorium_type(sequelize, DataTypes);
  let Faculty = _Faculty(sequelize, DataTypes);
  let Pulpit = _Pulpit(sequelize, DataTypes);
  let Subject = _Subject(sequelize, DataTypes);
  let Teacher = _Teacher(sequelize, DataTypes);

  Auditorium.belongsTo(Auditorium_type, { as: "auditorium_type_Auditorium type", foreignKey: "auditorium_type" });
  Auditorium_typehasMany(Auditorium, { as: "Auditoria", foreignKey: "auditorium_type" });
  Pulpit.belongsTo(Faculty, { as: "faculty_Faculty", foreignKey: "faculty" });
  Faculty.hasMany(Pulpit, { as: "Pulpits", foreignKey: "faculty" });
  Subject.belongsTo(Pulpit, { as: "pulpit_Pulpit", foreignKey: "pulpit" });
  Pulpit.hasMany(Subject, { as: "Subjects", foreignKey: "pulpit" });
  Teacher.belongsTo(Pulpit, { as: "pulpit_Pulpit", foreignKey: "pulpit" });
  Pulpit.hasMany(Teacher, { as: "Teachers", foreignKey: "pulpit" });

  return [
    Auditorium,
    Auditorium_type,
    Faculty,
    Pulpit,
    Subject,
    Teacher,
  ];
}

module.exports = initModels;
module.exports.ORM = initModels;
```

init-models.js

Выборка данных (select)

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('SeqTest', 'student', 'fitfit', { host: '172.16.193.223', dialect: 'mssql' });
const { Faculty, Pulpit, Teacher, Subject, Auditorium_type, Auditorium } = require('./models/init-models').ORM(sequelize, DataTypes);

const print = (result) => {
  let counter = 0;
  console.log('-----');
  result.forEach(elem => { console.log(++counter, elem.dataValues); });
}

sequelize.authenticate() // проверка соединения
  .then(() => { console.log('Соединение с базой данных установлено'); })
  .then(() => {
    Faculty.findAll().then(faculties => print(faculties));
    Pulpit.findAll().then(pulpits => print(pulpits));
    Teacher.findAll().then(teachers => print(teachers));
    Subject.findAll().then(subjects => print(subjects));
    Auditorium_type.findAll().then(auditorium_types => print(auditorium_types));
    Auditorium.findAll().then(auditoriums => print(auditoriums));
  })
  .catch(err => { console.log('Ошибка при соединении с базой данных:', err.message); });
console.log('-----');
```

Метод **findAll(options)** используется для выборки из таблицы, связанной с моделью.

Псевдонимы для столбцов

```
Faculty.findAll({attributes:['faculty', ['faculty_name', 'Наименование факультета']]})  
.then(faculties => print(faculties));
```

```
1 { faculty: 'ИдиП      ',  
  'Наименование факультета': 'Издательское дело и полиграфия' }  
2 { faculty: 'ИЭФ      ',  
  'Наименование факультета': 'Инженерно-экономический факультет' }  
3 { faculty: 'ЛХФ      ',  
  'Наименование факультета': 'Лесохозяйственный факультет' }  
4 { faculty: 'ТОВ      ',  
  'Наименование факультета': 'Технология органических веществ' }  
5 { faculty: 'ТТЛП      ',  
  'Наименование факультета': 'Технология и техника лесной промышленности' }  
6 { faculty: 'ХТиТ      ',  
  'Наименование факультета': 'Химическая технология и техника' }
```

options.attributes – список столбцов, которые нужно выбрать.

Чтобы **переименовать атрибут**, нужно передать массив с двумя элементами: первый – это имя столбца в таблице БД (или какое-то выражение), а второй – псевдоним.

Вызов агрегирующих функций

```
Teacher.findAll({  
    attributes:[  
        [sequelize.fn('COUNT', sequelize.col('pulpit')), 'count_pulpit'],  
    ]  
})  
.then(p => print(p));
```

1 { count_pulpit: 29 }

`sequelize.fn(fn, args)` используется для вызова функции БД (агрегирующих функций),

Если же нужно ссылаться на столбцы в функции, необходимо использовать `sequelize.col(col)`, чтобы столбцы правильно интерпретировались (как столбцы в БД, а не strings).

```
Teacher.findAll({  
    where:{pulpit:'ИСиТ'},  
    attributes:[  
        ['teacher_name', 'Преподаватель'],  
        ['pulpit', 'Кафедра']  
    ]  
})
```

options.where – условия выборки (по умолчанию предполагает сравнение на равенство Op.eq).

```
Executing (default): SELECT [teacher_name] AS [Преподаватель], [pulpit] AS [Кафедра] FROM [Teacher] AS [Teacher] WHERE [Teacher].[pulpit] = N'ИСиТ';  
-----  
1 { 'Преподаватель': 'Неизвестный', 'Кафедра': 'ИСиТ' }  
2 { 'Преподаватель': 'Акунович Станислав Иванович',  
  'Кафедра': 'ИСиТ' }  
3 { 'Преподаватель': 'Бракович Андрей Игорьевич',  
  'Кафедра': 'ИСиТ' }  
4 { 'Преподаватель': 'Герман Олег Витольдович',  
  'Кафедра': 'ИСиТ' }  
5 { 'Преподаватель': 'Гурин Николай Иванович',  
  'Кафедра': 'ИСиТ' }  
6 { 'Преподаватель': 'Дедко Александр Аркадьевич',  
  'Кафедра': 'ИСиТ' }  
7 { 'Преподаватель': 'Жиляк Надежда Александровна',  
  'Кафедра': 'ИСиТ' }
```

```
Teacher.findAll({  
    where:{pulpit:'ИСиТ', teacher:'БРКВЧ'},  
    attributes:[  
        ['teacher_name', 'Преподаватель'],  
        ['pulpit', 'Кафедра']  
    ]  
})
```

При передаче нескольких условий они по умолчанию объединяются операцией AND.

```
Executing (default): SELECT [teacher_name] AS [Преподаватель], [pulpit] AS [Кафедра] FROM [Teacher] AS [Teacher] WHERE [Teacher].[pulpit] = N'ИСиТ' AND [Teacher].[teacher] = N'БРКВЧ';  
-----  
1 { 'Преподаватель': 'Бракович Андрей Игорьевич',  
  'Кафедра': 'ИСиТ' }
```

```
Teacher.findAll({
  where: {
    [Sequelize.Op.or]: [{pulpit: 'ИСиТ'}, {pulpit: 'ЛВ'}]
  },
  attributes: [
    ['teacher_name', 'Преподаватель'],
    ['pulpit', 'Кафедра']
  ]
})
.then(p => print(p));
```

```
Teacher.findAll({
  where: {
    pulpits: {
      [Sequelize.Op.or]: ['ИСИТ', 'ЛВ']
    }
  },
  attributes: [
    ['teacher_name', 'Преподаватель'],
    ['pulpit', 'Кафедра']
  ]
})
```

```
Executing (default): SELECT [teacher_name] AS [Преподаватель], [pulpit] AS [Кафедра] FROM [Teacher] AS [Teacher] WHERE ([Teacher].[pulpit] = N'ИСиТ' OR [Teacher].[pulpit] = N'ЛВ');
-----
1 { 'Преподаватель': 'Неизвестный', 'Кафедра': 'ИСиТ' }
2 { 'Преподаватель': 'Акунович Станислав Иванович',
  'Кафедра': 'ИСиТ' }
3 { 'Преподаватель': 'Бракович Андрей Игорьевич',
  'Кафедра': 'ИСиТ' }
4 { 'Преподаватель': 'Герман Олег Витольдович',
  'Кафедра': 'ИСиТ' }
5 { 'Преподаватель': 'Гурин Николай Иванович',
  'Кафедра': 'ИСиТ' }
6 { 'Преподаватель': 'Дедко Александр Аркадьевич',
  'Кафедра': 'ИСиТ' }
7 { 'Преподаватель': 'Жиляк Надежда Александровна',
  'Кафедра': 'ИСиТ' }
8 { 'Преподаватель': 'Кабайло Александр Серафимович',
  'Кафедра': 'ИСиТ' }
9 { 'Преподаватель': 'Колесников Леонид Валерьевич',
  'Кафедра': 'ИСиТ' }
10 { 'Преподаватель': 'Лабоха Константин Валентинович',
   'Кафедра': 'ЛВ' }
```

Операторы **Op.and**, **Op.or**, **Op.not** и **др.** могут использоваться для создания произвольно сложных вложенных логических сравнений.

Sequelize.Op

- [Op.and]
- [Op.or]
- [Op.eq]
- [Op.ne]
- [Op.is]
- [Op.not]
- [Op.gt]
- [Op.gte]
- [Op.lt]
- [Op.lte]
- [Op.between]
- [Op.notBetween]
- [Op.all]
- [Op.in]
- [Op.notIn]
- [Op.like]
- [Op.notLike]
- [Op.startsWith]
- [Op.endsWith]
- [Op.substring]
- [Op.iLike]
- [Op.notILike]
- [Op.regexp] // MySQL/PG only
- [Op.notRegexp] // MySQL/PG only
- [Op.iRegexp] // PG only
- [Op.notIRegexp] // PG only
- [Op.any]

[Подробнее](#)

```
Auditorium.findAll({  
  where:{  
    auditorium_capacity: {[Sequelize.Op.gt]:80}  
  }  
})
```

Executing (default): `SELECT [auditorium], [auditorium_name], [auditorium_capacity], [auditorium_type] FROM [Auditorium] AS [Auditoriumt] WHERE [Auditoriumt].[auditorium_capacity] > 80;`

```
1 { auditorium: '?',  
  auditorium_name: '??',  
  auditorium_capacity: 90,  
  auditorium_type: 'ЛК' }  
2 { auditorium: '02Б-4',  
  auditorium_name: '02Б-4',  
  auditorium_capacity: 90,  
  auditorium_type: 'ЛК' }  
3 { auditorium: '103-4',  
  auditorium_name: '103-4',  
  auditorium_capacity: 90,  
  auditorium_type: 'ЛК' }
```

```
Auditorium.findAll({  
  where:{  
    auditorium_capacity: {  
      [Sequelize.Op.and]:  
      {  
        [Sequelize.Op.gt]:30,  
        [Sequelize.Op.lt]:90}  
      }  
    }  
})
```

Executing (default): `SELECT [auditorium], [auditorium_name], [auditorium_capacity], [auditorium_type] FROM [Auditorium] AS [Auditoriumt] WHERE ([Auditoriumt].[auditorium_capacity] > 30 AND [Auditoriumt].[auditorium_capacity] < 90);`

```
1 { auditorium: '236-1',  
  auditorium_name: '236-1',  
  auditorium_capacity: 60,  
  auditorium_type: 'ЛК' }  
2 { auditorium: '313-1',  
  auditorium_name: '313-1',  
  auditorium_capacity: 60,  
  auditorium_type: 'ЛК' }
```

Сортировка

```
Auditorium.findAll({
    order:[
        [ 'auditorium_capacity', 'DESC' ]
    ]
})
```

Executing (default): `SELECT [auditorium], [auditorium_name], [auditorium_capacity], [auditorium_type] FROM [Auditorium] AS [Auditoriumt] ORDER BY [Auditoriumt].[auditorium_capacity] DESC;`

```
1 { auditorium: '?',          ,
  auditorium_name: '??',
  auditorium_capacity: 90,
  auditorium_type: 'ЛК      ' }
2 { auditorium: '02Б-4',       ,
  auditorium_name: '02Б-4',
  auditorium_capacity: 90,
  auditorium_type: 'ЛК      ' }
```



```
Auditorium.findAll({
    order:[
        [ 'auditorium_type', 'ASC' ], [ 'auditorium_capacity', 'DESC' ]
    ]
})
```

Executing (default): `SELECT [auditorium], [auditorium_name], [auditorium_capacity], [auditorium_type] FROM [Auditorium] AS [Auditoriumt] ORDER BY [Auditoriumt].[auditorium_type] ASC, [Auditoriumt].[auditorium_capacity] DESC;`

```
1 { auditorium: '304-4',      ,
  auditorium_name: '304-4',
  auditorium_capacity: 90,
  auditorium_type: 'ЛБ-К      ' }
2 { auditorium: '423-1',      ,
  auditorium_name: '423-1',
  auditorium_capacity: 90,
  auditorium_type: 'ЛБ-К      ' }
```

`options.order` – определяет **порядок сортировки**. Используя массив, можно упорядочить несколько столбцов. Каждый элемент может быть дополнительном заключен в двухэлементный массив. Первый элемент – это столбец для сортировки, второй – направление.

Группировка

```
Auditorium.findAll({
  attributes:['auditorium_type', [sequelize.fn('sum', sequelize.col('auditorium_capacity')), 'sum_capacity']],
  group:['auditorium_type']
})
.then(auditoriums => print(auditoriums));
```

```
Executing (default): SELECT [auditorium_type], sum([auditorium_capacity]) AS [sum capacity] FROM [Auditorium] AS [Auditoriumt]
GROUP BY [auditorium type];
-----
1 { auditorium_type: 'ЛБ-К' , sum_capacity: 225 }
2 { auditorium_type: 'ЛК' , sum_capacity: 1220 }
3 { auditorium_type: 'ЛК-К' , sum_capacity: 90 }

D:\PSCA\Lec16_sequelize>
```

options.group – определяет поле или поля, по которым будут группироваться
выходные строки.

Соединение таблиц (активная загрузка)

Для того, чтобы выполнить соединение таблиц (активную загрузку), необходимо сперва настроить ассоциации между моделями

```
Auditorium.belongsTo(Auditorium_type, { as: "auditorium_type_Auditorium_type", foreignKey: "auditorium_type"});  
Auditorium_typehasMany(Auditorium, { as: "Auditoria", foreignKey: "auditorium_type"});  
Pulpit.belongsTo(Faculty, { as: "faculty_Faculty", foreignKey: "faculty"});  
Faculty.hasMany(Pulpit, { as: "Pulpits", foreignKey: "faculty"})  
Subject.belongsTo(Pulpit, { as: "pulpit_Pulpit", foreignKey: "pulpit"});  
Pulpit.hasMany(Subject, { as: "Subjects", foreignKey: "pulpit"});  
Teacher.belongsTo(Pulpit, { as: "pulpit_Pulpit", foreignKey: "pulpit"});  
Pulpit.hasMany(Teacher, { as: "Teachers", foreignKey: "pulpit"})
```

Соединение таблиц (активная загрузка)

```
Faculty.findAll({
  include: [
    { model: Pulpit, as: 'Pulpits', required: true }
  ]
})
.then(pulpits_faculties => {
  pulpits_faculties.forEach(facultyElem => {
    console.log(facultyElem.dataValues.faculty, facultyElem.dataValues.faculty_name);
    facultyElem.dataValues.Pulpits.forEach(pulpitElem => {
      console.log(' - ', pulpitElem.dataValues.pulpit, pulpitElem.dataValues.pulpit_name);
    });
  });
})
```

Executing (default): SELECT [Faculty].[faculty], [Faculty].[faculty_name], [Pulpits].[pulpit] AS [Pulpits.pulpit], [Pulpits].[pulpit_name] AS [Pulpits.pulpit_name], [Pulpits].[faculty] AS [Pulpits.faculty] FROM [Faculty] AS [Faculty] INNER JOIN [Pulpit] AS [Pulpits] ON [Faculty].[faculty] = [Pulpits].[faculty];
ИДИП Издательское дело и полиграфия
- - ИСИТ Информационные системы и технологии
- - ПОИСОИ Полиграфического оборудования и систем обработки информации
ЛХФ Лесохозяйственный факультет
- - ЛВ Лесоводства
- - ЛЭЗДВ Лесозащиты и древесиноведения
- - ЛПиСПС Ландшафтного проектирования и садово-паркового строительства
- - ЛУ Лесоустройства
- - ОВ Охотоведения
ТГЛП Технология и техника лесной промышленности
- - ЛМиЛМ Лесных машин и технологии лесозаготовок

required: true – inner join

options.include – список ассоциаций для загрузки.
Поддерживается либо {include: [Model1, Model2, ...]}, либо {include: [{model: Model1, as: 'Alias'}]} или {include: ['Alias']}.

model - модель, которую необходимо загрузить (то есть с которой нужно соединить).

as – псевдоним отношения на случай, если модель, которую необходимо загрузить, является псевдонимом. Для hasOne / belongsTo это должно быть имя в единственном числе, а дляhasMany - во множественном числе.

Соединение таблиц (активная загрузка)

```
Faculty.findAll({
  include: [
    { model: Pulpit, as: 'Pulpits', required: false }      required: false – left outer join
  ]
})
.then(pulpits_faculties => {
  pulpits_faculties.forEach(facultyElem => {
    console.log(facultyElem.dataValues.faculty, facultyElem.dataValues.faculty_name);
    facultyElem.dataValues.Pulpits.forEach(pulpitElem => {
      console.log('--', pulpitElem.dataValues.pulpit, pulpitElem.dataValues.pulpit_name);
    });
  });
})
```

Executing (default): SELECT [Faculty].[faculty], [Faculty].[faculty_name], [Pulpits].[pulpit] AS [Pulpits.pulpit], [Pulpits].[pulpit_name] AS [Pulpits.pulpit_name], [Pulpits].[faculty] AS [Pulpits.faculty] FROM [Faculty] AS [Faculty] LEFT OUTER JOIN [Pulpit] AS [Pulpits] ON [Faculty].[faculty] = [Pulpits].[faculty];

ИДИП Издательское дело и полиграфия
-- ИСИТ Информационный систем и технологий
-- ПОИСОИ Полиграфического оборудования и систем обработки информации
ИЭФ Инженерно-экономический факультет
-- МиЭП Менеджмента и экономики природопользования
-- ЭТИМ экономической теории и маркетинга
ЛХФ Лесохозяйственный факультет
-- ЛВ Лесоводства

== Faculty.findAll({ include: 'Pulpits' })

== Faculty.findAll({ include: { association: 'Pulpits' } })

Соединение таблиц (активная загрузка)

```
Faculty.findAll({
  include: [
    { model: Pulpit, as: 'Pulpits', right: true }      right: true - right join
  ]
})
.then(pulpits_faculties => {
  pulpits_faculties.forEach(facultyElem => {
    console.log(facultyElem.dataValues.faculty, facultyElem.dataValues.faculty_name);
    facultyElem.dataValues.Pulpits.forEach(pulpitElem => {
      console.log('--', pulpitElem.dataValues.pulpit, pulpitElem.dataValues.pulpit_name);
    })
  })
});
```

Executing (default): SELECT [Faculty].[faculty], [Faculty].[faculty_name], [Pulpits].[pulpit] AS [Pulpits.pulpit], [Pulpits].[pulpit_name] AS Pulpits.pulpit_name], [Pulpits].[faculty] AS [Pulpits.faculty] FROM [Faculty] AS [Faculty] RIGHT OUTER JOIN [Pulpit] AS [Pulpits] ON [Faculty].[faculty] = [Pulpits].[faculty];
ИДП Издательское дело и полиграфия
-- ИСИТ Информационный систем и технологий
-- ПОИСОИ Полиграфического оборудования и систем обработки информации
ЛХФ Лесохозяйственный факультет
-- ЛВ Лесоводства
-- ЛЗИДВ Лесозащиты и древесиноведения
-- ЛПИСПС Ландшафтного проектирования и садово-паркового строительства
-- ЛУ Лесоустройства

Выборка одной записи

```
Faculty.findByPk('ХТИТ')
    .then((faculty) => console.log(faculty.dataValues))
```

```
Executing (default): SELECT [faculty], [faculty_name] FROM [Faculty] AS [Faculty] WHERE [Faculty].[faculty] = N'ХТИТ';
{ faculty: 'ХТИТ', faculty_name: 'Химическая технология и техника' }
```

Метод **.findByPk(prKey)** получает объект **по первичному ключу**.

```
Teacher.findOne({ where: { pulpit: 'ИСИТ' } })
    .then((teacher) => console.log(teacher.dataValues))
```

```
Executing (default): SELECT [teacher], [teacher_name], [pulpit] FROM [Teacher] AS [Teacher] WHERE [Teacher].[pulpit] = N'ИСИТ' ORDER BY [Teacher].[teacher] OFFSET 0 ROWS FETCH NEXT 1 ROWS ONLY;
{ teacher: '?', teacher_name: 'Неизвестный', pulpit: 'ИСИТ' }
```

Метод **.findOne(options)** получает **один объект** по определенному критерию.

Добавление строк

```
Pulpit.create({pulpit:'MC',pulpit_name:'Мобильных систем', faculty:'ИТ'})  
.then(task=>console.log(task))  
.catch(err=>console.log('Error: ', err.message ));  
Executing (default): INSERT INTO [Pulpit] ([pulpit],[pulpit_name],[faculty]) OUTPUT INSERTED.* VALUES (@0,@1,@2)  
;  
Error: The INSERT statement conflicted with the FOREIGN KEY constraint "FK_PULPIT_FACULTY". The conflict occurred in database "SeqTest", table "dbo.FACULTY", column 'FACULTY'.
```

```
Faculty.create({faculty: 'ИТ', faculty_name: 'Информационных технологий'})  
.then(task=>console.log(task.dataValues))  
.catch(err=>console.log('Error: ', err.message ));  
Executing (default): INSERT INTO [Faculty] ([faculty],[faculty_name]) OUTPUT INSERTED.* VALUES (@0,@1);  
{ faculty: 'ИТ', faculty_name: 'Информационных технологий' }
```

```
Pulpit.create({pulpit:'MC',pulpit_name:'Мобильных систем', faculty:'ИТ'})  
.then(task=>console.log(task.dataValues))  
.catch(err=>console.log('Error: ', err.message ));  
Executing (default): INSERT INTO [Pulpit] ([pulpit],[pulpit_name],[faculty]) OUTPUT INSERTED.* VALUES (@0,@1,@2)  
;  
{ pulpit: 'MC', pulpit_name: 'Мобильных систем', faculty: 'ИТ' }
```

Вставка новых **данных** в таблицу осуществляется с помощью метода `.create(values, options)`.

Обновление строк

```
Pulpit.update(  
    {pulpit_name:'Программное обеспечение мобильных систем'},  
    {where:{pulpit:'XX'}}  
)  
.then(task=>console.log('Result:', task))  
.catch(err=>console.log('Error: ', err.message ));  
  
Executing (default): UPDATE [Pulpit] SET [pulpit_name]=@0 OUTPUT INSERTED.* WHERE [pulpit] = @1  
Result: [ 0 ]
```

При попытке изменения данных, которых не существует, ошибка возникать не будет.

```
Pulpit.update(  
    {pulpit_name:'Программное обеспечение мобильных систем'},  
    {where:{pulpit:'MC'}}  
)  
.then(task=>console.log('Result:', task))  
.catch(err=>console.log('Error: ', err.message ));  
  
Executing (default): UPDATE [Pulpit] SET [pulpit_name]=@0 OUTPUT INSERTED.* WHERE [pulpit] = @1  
Result: [ 1 ]
```

Для **обновления** применяется метод **.update(values, options)**, в который передается объект с новыми значениями и объект с критерием выборки обновляемых объектов (секция options.where).

В результате получаем **количество строк, затронутых операцией обновления**.

Удаление строк

```
Faculty.destroy({where:{faculty:'XX'}})
  .then(task=>console.log('Result:', task))
  .catch(err=>console.log('Error: ', err.message));
```

При попытке удалить несуществующие данные ошибки не возникнет.

```
Executing (default): DELETE FROM [Faculty] WHERE [faculty] = N'XX'; SELECT @@ROWCOUNT AS AFFECTEDROWS;
Result: 0
```

```
Faculty.destroy({where:{faculty:'ИТ'}})
  .then(task=>console.log('Result:', task))
  .catch(err=>console.log('Error: ', err.message));
```

При попытке удалить связанную строку будет возникать ошибка внешнего ключа.

```
Executing (default): DELETE FROM [Faculty] WHERE [faculty] = N'ИТ'; SELECT @@ROWCOUNT AS AFFECTEDROWS;
Error: The DELETE statement conflicted with the REFERENCE constraint "FK_PULPIT_FACULTY". The conflict occurred
in database "SeqTest", table "dbo.PULPIT", column 'FACULTY'.
```

```
Pulpit.destroy({where:{faculty:'ИТ'}})
  .then(task=>console.log('Result:', task))
  .catch(err=>console.log('Error: ', err.message));
```

```
Соединение с базой данных установлено
Executing (default): DELETE FROM [Pulpit] WHERE [faculty] = N'ИТ'; SELECT @@ROWCOUNT AS AFFECTEDROWS;
Result: 1
```

Для **удаления** используется метод **.destroy(options)**, в который передается объект-критерий выборки удаляемых объектов.

В результате получаем количество удаленных строк.

Raw query

Поскольку часто бывают случаи, когда проще выполнять "сырые" (самописные) SQL-запросы, можно использовать метод `sequelize.query(query, options)`.

```
sequelize.query('select pulpit, faculty from Pulpit')
  .then((pulpits)=>{console.log(pulpits)})
```

```
Executing (default): select pulpit, faculty from Pulpit
[ { pulpit: 'ИСиТ', faculty: 'ИДиП' },
  { pulpit: 'ЛВ', faculty: 'ЛХФ' },
  { pulpit: 'ЛЗиДВ', faculty: 'ЛХФ' },
  { pulpit: 'ЛМиЛЗ', faculty: 'ТТЛП' },
  { pulpit: 'ЛПиСПС', faculty: 'ЛХФ' },
```

```
sequelize.query('select Pulpit.pulpit, Pulpit.faculty, Subject.subject_name  from Pulpit join Subject '
  + ' on Pulpit.pulpit = Subject.pulpit order by Pulpit.faculty')
  .then((result)=>{console.log(result)})
```

```
Executing (default): select Pulpit.pulpit, Pulpit.faculty, Subject.subject_name  from Pulpit join Subject  on Pu
lpit.pulpit = Subject.pulpit order by Pulpit.faculty
[ { pulpit: 'ИСиТ',
  faculty: 'ИДиП',
  subject_name: 'Базы данных' },
  { pulpit: 'ИСиТ',
  faculty: 'ИДиП',
  subject_name: 'Дискретная математика' },
```

По умолчанию функция **возвращает массив результатов и объект, содержащий метаданные** (например, кол-во затронутых строк и т. д., зависит от диалекта).

Raw query (замены)

```
sequelize.query('select Pulpit.pulpit, Pulpit.faculty, Subject.subject_name from Pulpit join Subject'  
    +' on Pulpit.pulpit = Subject.pulpit'  
    +' where Pulpit.faculty = :faculty',  
    { replacements: { faculty: 'ИдиП' } }  
)  
.then((result)=>{console.log(result)})
```

Executing (default): select Pulpit.pulpit, Pulpit.faculty, Subject.subject_name from Pulpit join Subject on Pulpit.pulpit = Subject.pulpit where Pulpit.faculty = N'ИдиП'
[{ pulpit: 'ИСиТ',
 faculty: 'ИдиП',
 subject_name: 'Базы данных' },
{ pulpit: 'ИСиТ',
 faculty: 'ИдиП',
 subject_name: 'Дискретная математика' },

Замены (replacements) в запросе могут выполняться двумя разными способами:

- с использованием **именованных** параметров, начинающихся со знака :
- с использованием **безымянных**, представленных знаком ?

Замены передаются в объекте options в свойстве replacements.

Если передан массив, ? будут заменены в том порядке, в котором они появляются в массиве

Если передан объект, :key будут заменены ключами этого объекта.

Если объект содержит ключи, не найденные в запросе, или наоборот, будет выброшено исключение.

Scopes

```
1 const Sequelize = require('sequelize');
const sequelize = new Sequelize('SeqTest', 'student', 'fitfit', {
  host: '172.16.193.223', dialect: 'mssql',
})

const Faculty = sequelize.define('Faculty', { ... });
const Pulpit = sequelize.define('Pulpit', { ... });
const Teacher = sequelize.define('Teacher', {
  teacher: { type: Sequelize.STRING, allowNull: false, primaryKey: true },
  teacher_name: { type: Sequelize.STRING, allowNull: false },
  pulpit: { type: Sequelize.STRING, allowNull: false, references: { model: Pulpit, key: 'pulpit' } },
}, {
  scopes: {
    teachersByFac(f) {
      return {
        include: [
          { model: Pulpit, where: { faculty: f } }
        ]
      }
    },
    teachersFromISiT: {
      where: { pulpit: 'ИСИТ' }
    }
  },
  sequelize,
  tableName: 'Teacher',
  timestamps: false
});

Pulpit.belongsTo(Faculty, { foreignKey: 'faculty', targetKey: 'faculty' });
Teacher.belongsTo(Pulpit, { foreignKey: 'pulpit', targetKey: 'pulpit' });
```

2

```
quelize.authenticate()
.then(async () => {
  console.log('Соединение с базой данных установлено');

  let teachers = await Teacher.scope({ method: ['teachersByFac', 'ИЭФ'] }).findAll();
  teachers.forEach(t => {
    console.log(t.teacher_name)
  })

  teachers = await Teacher.scope('teachersFromISiT').findAll();
  teachers.forEach(t => {
    console.log(t.teacher_name)
  })
})  
.catch(err => { console.log('Ошибка при соединении с базой данных', err) })
```

```
PS D:\Материалы\cwp_16> node .\16-03.js
Executing (default): SELECT 1+1 AS result
Соединение с базой данных установлено
Executing (default): SELECT [Teacher].[teacher], [Teacher].[teacher_name], [Teacher].[pulpit], [Pulpit].[pulpit] AS [Pulpit.pulpit].[name], [Pulpit].[faculty] AS [Pulpit.faculty] FROM [Teacher] AS [Teacher] INNER JOIN [Pulpit] AS [Pulpit] ON [Teacher].[pulpit].[faculty] = N'ИЭФ';
Барановский Станислав Иванович
Неверов Александр Васильевич
Неизвестный
Акунович Станислав Иванович
```

Области видимости, скоупы используются, чтобы помочь повторно использовать код. Можно определить часто используемые запросы, указав такие параметры, как where, include, limit и т.д. Их можно применить к findAll, count, update, destroy.... Скоупы бывают **обычными** и **по умолчанию**. Скоупы определяются **при создании модели** (`options.scopes`, `options.defaultScope`) или с помощью `Model.addScope()`. Могут быть **объектами** поиска или **функциями**, **возвращающими объекты** поиска, за исключением скоупа по умолчанию (`options.defaultScope`), который может быть только объектом.

Hooks

1

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('SeqTest', 'student', 'fitfit', {
  host: '172.16.193.223', dialect: 'mssql',
  define: {
    hooks: { // глобальный хук
      beforeCreate: (instance, options) => { console.log('----- global beforeCreate #1 -----'); }
    }
  }
});
sequelize.addHook('beforeCreate', () => { // глобальный хук
  console.log('----- global beforeCreate #2 -----');
});

const Faculty = sequelize.define('Faculty', {
  faculty: { type: Sequelize.STRING, allowNull: false, primaryKey: true },
  faculty_name: { type: Sequelize.STRING, allowNull: false }
}, {
  hooks: { // локальные хуки
    beforeCreate: (instance, options) => { console.log('----- local faculty beforeCreate -----'); },
    beforeBulkUpdate: (instance, options) => { console.log('----- beforeUpdate -----'); },
    afterBulkUpdate: (instance, options) => { console.log('----- afterUpdate -----'); }
  },
  sequelize,
  tableName: 'Faculty',
  timestamps: false
});
const Pulpit = sequelize.define('Pulpit', {
  pulpit: { type: Sequelize.STRING, allowNull: false, primaryKey: true },
  pulpit_name: { type: Sequelize.STRING, allowNull: false },
  faculty: { type: Sequelize.STRING, allowNull: false, references: { model: Faculty, key: "faculty" } },
}, {
  sequelize,
  tableName: 'Pulpit',
  timestamps: false
});
```

2

```
quelize.authenticate()
  .then(async () => {
    console.log('Соединение с базой данных установлено');
    await Faculty.create({ faculty: 'ИТ', faculty_name: 'Информационных технологий' });
    await Pulpit.create({ pulpit: 'ПИ', pulpit_name: 'Программной инженерии', faculty: 'ИТ' });
    await Faculty.update({ faculty_name: 'Информационных технологий v2' }, { where: { faculty: 'ИТ' } });
  })
  .catch(err => { console.log('Ошибка при соединении с базой данных', err) })
```

```
PS D:\Материалы\cwp_16> node .\16-02.js
Executing (default): SELECT 1+1 AS result
Соединение с базой данных установлено
----- local faculty beforeCreate -----
----- global beforeCreate #2 -----
Executing (default): INSERT INTO [Faculty] ([faculty],[faculty_name]) OUTPUT INSERTED.
----- global beforeCreate #1 -----
----- global beforeCreate #2 -----
Executing (default): INSERT INTO [Pulpit] ([pulpit],[pulpit_name],[faculty]) OUTPUT IN
----- beforeUpdate -----
Executing (default): UPDATE [Faculty] SET [faculty_name]=@0 WHERE [faculty] = @1
----- afterUpdate -----
```

Хуки, или события жизненного цикла представляют собой функции, которые выполняются до/после/во время действий с данными.

Хуки бывают **глобальные постоянные** (для всех моделей), **глобальные по умолчанию** (используется для модели если она не определяет свой собственный хук),

локальные хуки инстанса и **локальные хуки модели**.

Локальные хуки всегда запускаются перед глобальными хуками.

Определение хуков модели осуществляется **при ее создании** (`options.hooks`) либо с помощью `Model.addHook()` (удаление через `Model.removeHook()`).

[Подробнее про хуки](#)

Транзакции (неуправляемые)

```
sequelize.authenticate() // проверка соединения
.then(()=>{console.log('Соединение с базой данных установлено');})
.then(()=>{
  return sequelize.transaction({isolationLevel: Sequelize.Transaction.ISOLATION_LEVELS.READ_COMMITTED})
  .then (t=>{
    return Pulpit.create({pulpit:'MC',pulpit_name:'Мобильных систем',faculty:'ИТ'}, {transaction:t})
    .then((r)=>{
      return Subject.update({ pulpit:'XXX'},
        {where:{subject:'БД'}},
        {transaction:t}
      )
    })
    .then((r)=>{console.log('--commit',r); return t.commit();})
    .catch((e)=>{console.error("--rollback", e.name); return t.rollback();});
  })
})
.catch(err => {console.log('Ошибка при соединении с базой данных:', err.message)});
```

Во время вызовов необходимо передавать транзакцию в качестве свойства options.transaction

Подтверждение и откат транзакции
должны выполняться **вручную**
(путем вызова .commit(), .rollback()).

```
Executing (default): SELECT 1+1 AS result
Соединение с базой данных установлено
Executing (8c18971780bc966daf85): BEGIN TRANSACTION;
Executing (8c18971780bc966daf85): INSERT INTO [Pulpit] ([pulpit],[pulpit_name],[faculty]) OUTPUT INSERTED.* VALUES (@0,@1,@2);
Executing (default): UPDATE [Subject] SET [pulpit]=@0 OUTPUT INSERTED.* WHERE [subject] = @1
--rollback SequelizeForeignKeyConstraintError
Executing (8c18971780bc966daf85): ROLLBACK TRANSACTION;

D:\PSCA\Lec16_sequelize>
```

Транзакции (неуправляемые)

```
sequelize.authenticate() // проверка соединения
.then(()=>{console.log('Соединение с базой данных установлено')})
.then(()=>{
    return sequelize.transaction({isolationLevel: Sequelize.Transaction.ISOLATION_LEVELS.READ_COMMITTED})
    .then (t=>{
        return Faculty.create({faculty:'ИТ',faculty_name:'Информационных технологий'}, {transaction:t})
        .then((r)=>{
            return Pulpit.create({pulpit:'ПИ', pulpit_name:'Программной инженерии',faculty:'ИТ'},
                {transaction:t}
            )
        })
        .then((r)=>{console.log('--commit'); return t.commit();})
        .catch((e)=>{console.error("--rollback", e.name); return t.rollback();});
    })
})
.catch(err => {console.log('Ошибка при соединении с базой данных:', err.message)});
```

В свойстве **options.isolationLevel** можно задать уровень изолированности.

```
D:\PSCA\Lec16_sequelize>node 16-03
-----
Executing (default): SELECT 1+1 AS result
Соединение с базой данных установлено
Executing (9b3f431c3f5d4b377acd): BEGIN TRANSACTION;
Executing (9b3f431c3f5d4b377acd): INSERT INTO [Faculty] ([faculty],[faculty_name]) OUTPUT INSERTED.* VALUES (@0,@1);
Executing (9b3f431c3f5d4b377acd): INSERT INTO [Pulpit] ([pulpit],[pulpit_name],[faculty]) OUTPUT INSERTED.* VALUES (@0,@1,@2);
--commit
Executing (9b3f431c3f5d4b377acd): COMMIT TRANSACTION;

D:\PSCA\Lec16_sequelize>
```

Транзакции (управляемые)

Управляемая транзакция создается, передавая обратный вызов в `sequelize.transaction(options, callback)`.

```
sequelize.authenticate()
  .then(() => {
    console.log('Соединение с базой данных установлено');
  })
  .then(() => {
    return sequelize.transaction(async (t) => {
      await Pulpit.create({ pulpit: 'МС', pulpit_name: 'Мобильных систем', faculty: 'ИТ' }, { transaction: t })
      await Teacher.update({ pulpit: 'МС' }, { where: { teacher: 'БРКВЧ' } }, { transaction: t })
    })
  })
  .catch(err => { console.log('Ошибка при соединении с базой данных', err) })
```

Во время вызовов необходимо передавать транзакцию в качестве свойства `options.transaction`

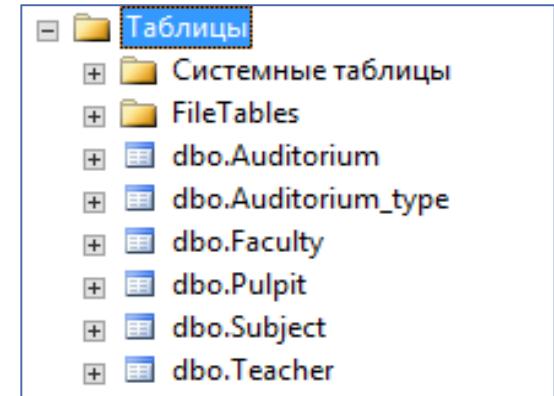
Sequelize **автоматически откатит** транзакцию, если возникнет какая-либо ошибка, **или зафиксирует** транзакцию в противном случае.

```
ps D:\NodeJS\samples\cwp_16> node 16-06
Executing (default): SELECT 1+1 AS result
Соединение с базой данных установлено
Executing (4c55ab6bbaade4bac097): BEGIN TRANSACTION;
Executing (4c55ab6bbaade4bac097): INSERT INTO [Pulpit] ([pulpit],[pulpit_name],[faculty]) OUTPUT INSERTED.[pulpit],INSERTED.[pulpit_name],INSERTED.[faculty] VALUES(?, ?, ?)
Executing (4c55ab6bbaade4bac097): UPDATE [Teacher] SET [pulpit]=@0 WHERE [teacher] = @1
Executing (4c55ab6bbaade4bac097): COMMIT TRANSACTION;
```

Подход code first

```
function internalORM(sequelize){  
    Faculty.init(  
        { ...  
        },  
        {sequelize, modelName:'Faculty', tableName:'Faculty', timestamps:false}  
    );  
    Pulpit.init (  
        { ...  
        },  
        {sequelize, modelName:'Pulpit', tableName:'Pulpit', timestamps:false}  
    );  
  
    Teacher.init(  
        { ...  
        },  
        {sequelize, modelName:'Teacher', tableName:'Teacher', timestamps:false}  
    );  
    Subject.init(  
        { ...  
        },  
        {sequelize, modelName:'Subject', tableName:'Subject', timestamps:false}  
    );  
    Auditorium_type.init(  
        { ...  
        },  
        {sequelize, modelName:'Auditorium_type', tableName:'Auditorium_type', timestamps:false}  
    );  
    Auditorium.init(  
        { ...  
        },  
        {sequelize, modelName:'Auditorium', tableName:'Auditorium', timestamps:false}  
    )  
    sequelize.sync({force:true});  
};
```

force: true – создает таблицу, сначала удаляя ее, если она уже существует.
alter: true – проверяет текущее состояние таблицы в БД, а затем выполняет необходимые изменения для соответствия модели.



Модель можно синхронизировать с БД, вызвав **model.sync(options)**. С помощью этого вызова Sequelize автоматически выполнит SQL-запрос к базе данных. Обратите внимание, что в итоге изменится только таблица в БД, а не модель на стороне JavaScript.

Можно использовать **sequelize.sync()** для автоматической синхронизации всех моделей.

Подход database first

Модели можно автоматически генерировать через командную строку с помощью пакета **sequelize-auto**. Кроме него еще потребуется пакет для используемого **диалекта** (tedious, mysql, pg,...) и сам **sequelize**.

```
PS D:\NodeJS\samples\cwp_16> npm install --save-dev sequelize-auto
added 19 packages, and audited 149 packages in 5s

50 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\NodeJS\samples\cwp_16> npm install tedious
up to date, audited 149 packages in 2s

50 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

```
{ config.json > {} dialectOptions > {} options > encrypt
  1 {
  2   "dialectOptions": {
  3     "options": {
  4       "encrypt": false
  5     }
  6   }
  7 }
```

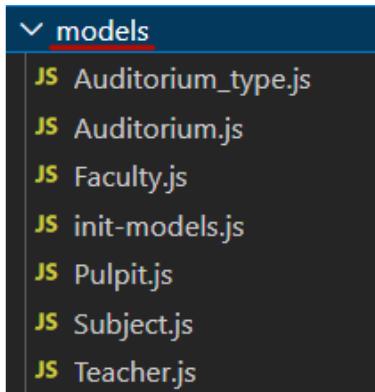
В файле config.js можно указать специфические для диалекта настройки.

```
PS D:\NodeJS\samples\cwp_16> npx sequelize-auto -o "./models" -d Univer -h localhost -u sa -p 1433 -x 123 -e mssql -c config.json
Warning: using a password on the command line interface can be insecure.
{
  dialectOptions: { options: { encrypt: false } },
  directory: './models',
  additional: {},
  dialect: 'mssql',
  port: 1433,
  host: 'localhost',
  database: 'Univer',
```

```
Executing (default): SELECT COUNT(0) AS trigger_count
  FROM sys.objects tr, sys.objects tb
 WHERE tr.type = 'TR'
   AND tr.parent_object_id = tb.object_id
   AND tb.object_id = OBJECT_ID('dbo.Auditorium')
```

Done!

Подход database first



В результате sequelize-auto создает **файл инициализации** ./models/init-models.js, который содержит код для загрузки определения каждой модели в Sequelize, а также создание ассоциаций между ними.

Кроме файла инициализации создаются **файлы с моделями для всех таблиц** в указанной БД.

```
models > JS init-models.js > ...
1 var DataTypes = require("sequelize").DataTypes;
2 var _Auditorium = require("./Auditorium");
3 var _Auditorium_type = require("./Auditorium_type");
4 var _Faculty = require("./Faculty");
5 var _Pulpit = require("./Pulpit");
6 var _Subject = require("./Subject");
7 var _Teacher = require("./Teacher");
8
9 function initModels(sequelize) {
10    var Auditorium = _Auditorium(sequelize, DataTypes);
11    var Auditorium_type = _Auditorium_type(sequelize, DataTypes);
12    var Faculty = _Faculty(sequelize, DataTypes);
13    var Pulpit = _Pulpit(sequelize, DataTypes);
14    var Subject = _Subject(sequelize, DataTypes);
15    var Teacher = _Teacher(sequelize, DataTypes);
16
17    Auditorium.belongsTo(Auditorium_type, { as: "auditorium_type_Auditorium_type", foreignKey: "auditorium_type_id" });
18    Auditorium_type.hasMany(Auditorium, { as: "Auditoria", foreignKey: "auditorium_type_id" });
19    Pulpit.belongsTo(Faculty, { as: "faculty_Faculty", foreignKey: "faculty_id" });
20    Faculty.hasMany(Pulpit, { as: "Pulpits", foreignKey: "faculty_id" });
21    Subject.belongsTo(Pulpit, { as: "pulpit_Pulpit", foreignKey: "pulpit_id" });
22    Pulpit.hasMany(Subject, { as: "Subjects", foreignKey: "pulpit_id" });
23    Teacher.belongsTo(Pulpit, { as: "pulpit_Teacher", foreignKey: "pulpit_id" });
24    Pulpit.hasMany(Teacher, { as: "Teachers", foreignKey: "pulpit_id" });
25
26    return {
27      Auditorium,
28      Auditorium_type,
29      Faculty,
30      Pulpit,
31      Subject,
32      Teacher,
33    };
34  }
35  module.exports = initModels;
36  module.exports.initModels = initModels;
37  module.exports.default = initModels;
```

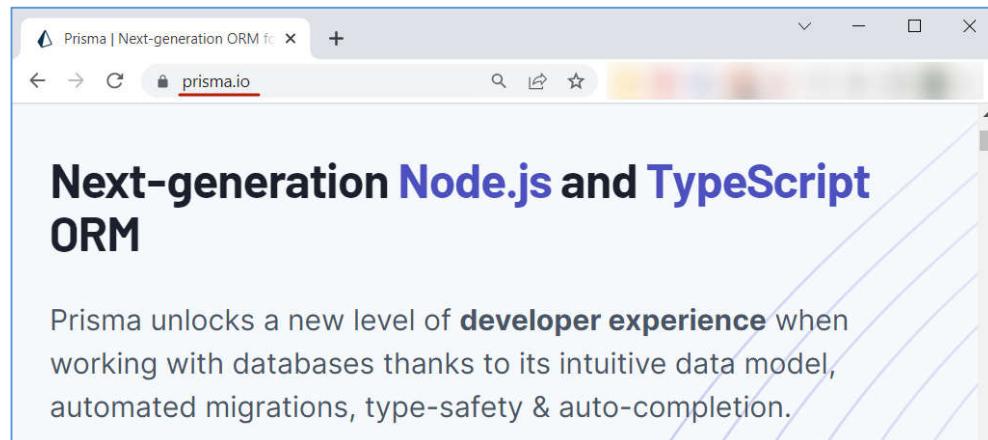
```
models > JS Faculty.js > ...
1 const Sequelize = require('sequelize');
2 module.exports = function(sequelize, DataTypes) {
3   return sequelize.define('Faculty', {
4     faculty: {
5       type: DataTypes.CHAR(10),
6       allowNull: false,
7       primaryKey: true
8     },
9     faculty_name: {
10       type: DataTypes.STRING(50),
11       allowNull: true
12     }
13   }, {
14     sequelize,
15     tableName: 'Faculty',
16     schema: 'dbo',
17     timestamps: false,
18     indexes: [
19       {
20         name: "PK_FACULTY",
21         unique: true,
22         fields: [
23           { name: "faculty" },
24         ]
25       }
26     ],
27   });
28 };
```

Prisma

=

это **ORM-библиотека**, нового поколения с открытым исходным кодом для Node.js и TypeScript. Она состоит из следующих инструментов:

- Prisma Client: автогенерируемый и типобезопасный клиент базы данных;
- Prisma Migrate: система миграций;
- Prisma Studio: пользовательский интерфейс для просмотра и редактирования данных.



[документация](#)

Основные характеристики

- написан на Rust;
- реализует паттерн Data Mapper;
- поддерживаемые языки: TypeScript, JavaScript;
- поддерживаются MySQL, PostgreSQL, MSSQL, SQLite, MongoDB, CockroachDB (поддержка PlanetScale в предварительной версии);
- предоставляет типобезопасный API (TypeScript);
- имеет набор инструментов для работы с базами данных (отправка запросов, моделирование, миграции, прототипирование, data seeding, студия для просмотра и изменения);
- способен генерировать определение схемы базы данных и клиентский код на основе структуры базы данных;
- поддерживает различные параметры запросов (фильтрация, сортировка, группировка, пагинация и др.);
- [Prisma Client API](#)

Установка

Прежде всего необходимо скачать [пакет prisma](#) в качестве dev-dependency.

```
PS D:\NodeJS\samples\cwp_32> npm install prisma --save-dev
                                         ^^^^^^
up to date, audited 5 packages in 2s
found 0 vulnerabilities
```

Создание моделей

Существует два способа определения модели данных:

- 1) Генерация модели данных путем интроспекции БД: если есть существующая БД, можно создать модель данных путем интроспекции БД.
- 2) Написание моделей данных вручную + Prisma Migrate: можно написать свою модель данных вручную и сопоставить ее с БД с помощью Prisma Migrate.

Создание моделей вручную

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlserver"
  url      = env("DATABASE_URL")
}

model Auditorium {
  auditorium   String      @id(map: "PK_Auditorium_9ADEC5635B13F344") @db.NVarChar(10)
  name          String?
  capacity      Int?
  type          String      @map("auditorium_type") @db.NVarChar(10)
  createdAt     DateTime   @default(now())
  updatedAt     DateTime   @updatedAt
  Auditorium_type Auditorium_type @relation(fields: [type], references: [type], onUpdate: NoAction, map: "FK_Auditorium_type")
  @@map("auditoriums")
}

model Auditorium_type {
  type          String      @id(map: "AUDITORIUM_TYPE_PK") @map("auditorium_type") @db.NVarChar(10)
  typename      String      @map("auditorium_typename") @db.NVarChar(30)
  createdAt     DateTime   @default(now())
  updatedAt     DateTime   @updatedAt
  Auditorium Auditorium[]
  @@map("auditorium_types")
}
```

Файл схемы Prisma (schema file, Prisma schema или schema) – это основной файл конфигурации для настройки Prisma. Обычно он называется `schema.prisma`. Файл схемы написан на языке **Prisma Schema Language** (PSL).

В схеме обычно настраиваются три вещи:

- **Data source** (источник данных): указывает сведения об источниках данных, к которым Prisma должна подключиться (например, подключение к БД).
- **Generator** (генератор): указывает, какие клиенты должны быть созданы на основе модели данных (например, клиент Prisma).
- **Data model** (модель данных): определяет модели и их взаимосвязь.

Модель

Каждая из моделей **описывает таблицу/коллекцию в БД**. Модель выполняет две основные функции:

- 1) **представлять** (сущность) **таблицу/коллекцию** в реляционных БД или коллекцию в MongoDB;
- 2) **обеспечивать основу для запросов** в клиентском API Prisma.

```
model Auditorium {  
    auditorium String      @id(map: "PK_Auditori_9ADEC5635B13F344") @db.NVarChar(10)  
    name        String?  
    capacity    Int?  
    type        String      @map("auditorium_type") @db.NVarChar(10)  
    createdAt   DateTime   @default(now())  
    updatedAt   DateTime   @updatedAt  
    Auditorium_type Auditorium_type @relation(fields: [type], references: [type], onUpdate: NoAction, map: "FK_Auditoriu_audit_1CF15040")  
    @@map("auditoriums")  
}
```

Соглашения об именовании моделей Prisma (форма ед. числа, PascalCase) не всегда соответствуют именам таблиц/коллекций в БД (форма множ. числа, snake_case). Однако можно придерживаться соглашения об именах, не переименовывая базовую таблицу/коллекцию в БД, используя атрибут `@@map`.

Поля

Свойства модели называются **полями**. Поле состоит из:

- **имя** поля;
- **тип** поля;
- необязательные **модификаторы типа**;
- необязательные **атрибуты**, включая собственные атрибуты типа базы данных.

```
model Auditorium {  
    auditorium      String          @id(map: "PK_Auditori_9ADEC5635B13F344") @db.NVarChar(10)  
    name             String?  
    capacity         Int?  
    type             String          @map("auditorium_capacity")  
    type             String          @map("auditorium_type") @db.NVarChar(10)  
    createdAt        DateTime        @default(now())  
    updatedAt        DateTime        @updatedAt  
    Auditorium_type Auditorium_type @relation(fields: [type], references: [type], onUpdate: NoAction, map: "FK_Auditoriu_audit_1CF15040")  
    @map("auditoriums")  
}
```

Типы

- **скалярные** (String, Boolean, Int, BigInt, Float, Decimal, DateTime, Json, Bytes),
- **поля отношений** (relation field),
- **атрибут собственных типов БД** (@db).
- также можно определять **enum**'ы (считается скалярным) и **составные типы**, если они поддерживаются СУБД.

```
model Auditorium {  
    auditorium      String          @id(map: "PK_Auditori_9ADEC5635B13F344") @db.NVarChar(10)  
    name            String?  
    capacity        Int?  
    type            String          @map("auditorium_type") @db.NVarChar(10)  
    createdAt       DateTime        @default(now())  
    updatedAt       DateTime        @updatedAt  
    Auditorium_type Auditorium_type @relation(fields: [type], references: [type], onUpdate: NoAction, map: "FK_Auditoriu_audit_1CF15040")  
    @@map("auditoriums")  
}
```

Модификаторы типа

- знак ? (необязательное),
- знак [] (список)

```
model Pulpit {  
    pulpit      String    @id(map: "PK_PULPIT") @db.NVarChar(10)  
    pulpit_name String?  @db.NVarChar(100)  
    faculty    String?  @db.NVarChar(10)  
    Faculty    Faculty? @relation(fields: [faculty], references: [faculty], onDelete: Cascade,  
    Subject    Subject[]  
    Teacher    Teacher[]  
  
    @@map("pulpits")  
}
```

Атрибуты

Атрибуты **изменяют поведение полей** (префикс @) **или** **модели** (префикс @@). Описание некоторых из них:

- @id – поле является РК таблицы.
- @@id – составной РК таблицы.
- @unique – уникальное поле в пределах таблицы.
- @@unique – составное ограничение уникальности для указанных полей.
- @relation – отношение между таблицами (fields – поля текущей модели, references – поля другой модели, тар – имя FK в БД, ...)
- @map – привязка поля модели к указанному столбцу таблицы.
- @@map – привязка модели к указанной таблице.
- @updateAt – обновляет поле текущей датой и временем при модификации записи.
- @ignore – невалидное поле.
- @@ignore – невалидная модель.

Attributes	@@index
@id	
@@id	
@default	
@unique	
@@unique	
@@index	
@@relation	
@@map	
@@@map	
@@updatedAt	
@ignore	
@@@ignore	

Функции атрибутов

Функции атрибутов можно использовать для указания значений по умолчанию для полей модели.

- **auto** – представляет дефолтные значения, генерируемые БД, используется для генерации ObjectId (только для MongoDB);
- **autoincrement** – генерирует последовательные целые числа (SERIAL в PostgreSQL, не поддерживается MongoDB);
- **cuid** – генерирует глобальный уникальный идентификатор на основе спецификации cuid;
- **uuid** – генерирует глобальный уникальный идентификатор на основе спецификации UUID;
- **now** – возвращает текущую отметку времени (timestamp) (CURRENT_TIMESTAMP в PostgreSQL);
- **sequence** – создает последовательность целых чисел в базовой базе данных и назначает увеличенные значения значениям созданных записей на основе этой последовательности (только для CockroachDB).
- **dbgenerated** – представляет дефолтные значения, которые не могут быть выражены в схеме (например, random()).

Attribute functions

auto()

autoincrement()

sequence()

cuid()

uuid()

now()

dbgenerated()

Миграция Бд =

это **контролируемый набор изменений**, который влияет на структуру базы данных. Миграции помогают **перевести схему базы данных из одного состояния в другое**. Например, в рамках миграции можно создавать или удалять таблицы и столбцы, разделять поля в таблице или добавлять типы и ограничения в свою базу данных.

Prisma Migrate =

это **инструмент миграции базы данных**,
который поддерживает шаблон
миграции `model/entity-first` для
управления схемами базы данных в
вашей локальной среде и в рабочей
среде.

Миграция

```
PS D:\NodeJS\samples\cwp_32> npx prisma migrate dev --name init
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma
Datasource "db": SQL Server database

dev - выполняет миграцию для разработки
deploy - выполняет производственную миграцию

Applying migration `20230309221538_init`

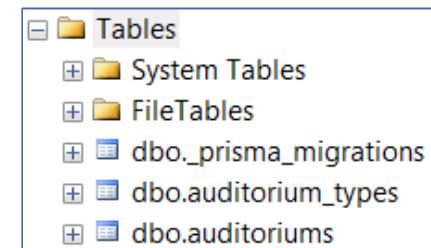
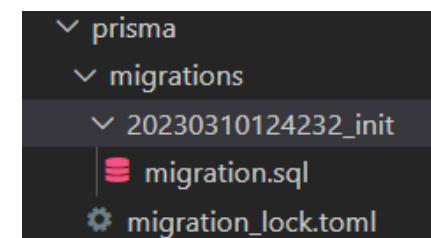
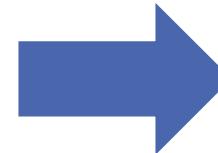
The following migration(s) have been created and applied from new schema changes:

migrations/
└── 20230309221538_init/
    └── migration.sql

Your database is now in sync with your schema.

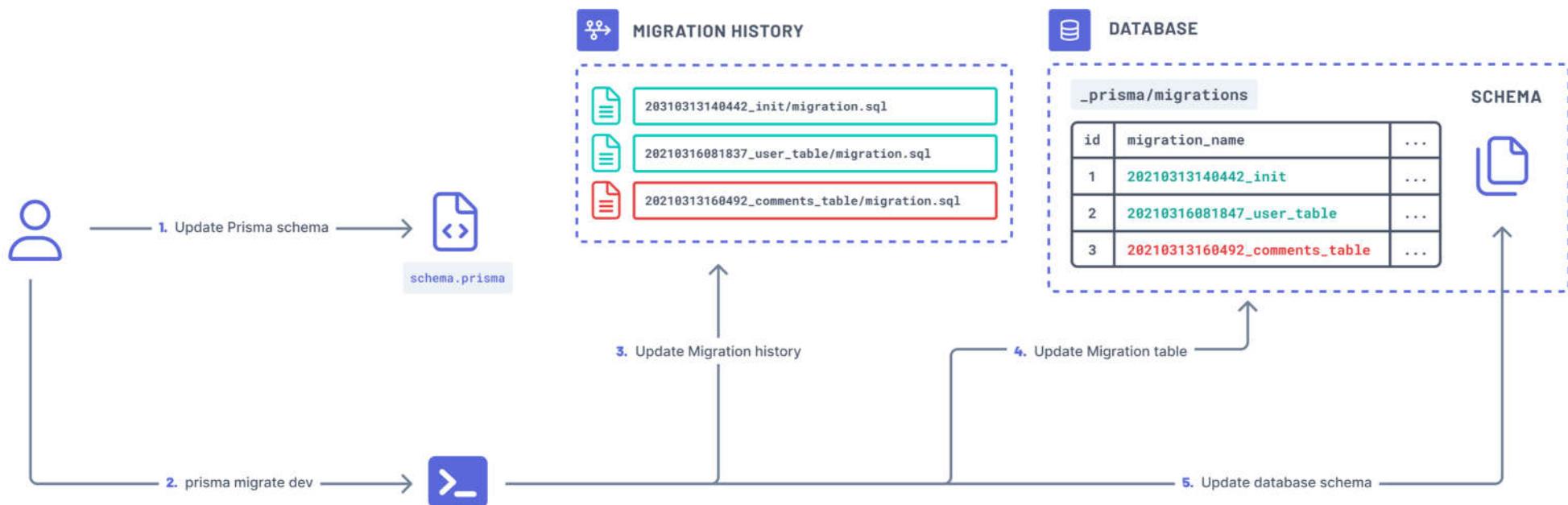
✓ Generated Prisma Client (4.10.1 | library) to .\node_modules\@prisma\client in 106ms
```

dev – выполняет миграцию для разработки
deploy – выполняет производственную миграцию



Команда **migrate** приводит к созданию БД при ее отсутствии, генерации файла `prisma/migrations/migration_name.sql`, выполнению инструкции из этого файла (синхронизации БД со схемой) и генерации (регенерации) клиента.

Создание/обновление моделей вручную + Prisma Migrate



Создание моделей путем интроспекции

1. Инициализация проекта

```
PS D:\NodeJS\samples\cwp_32> npx prisma
Prisma is a modern DB toolkit to query, migrate and model your database (https://prisma.io)
Usage
$ prisma [command]
Commands
  init      Set up Prisma for your app
  generate   Generate artifacts (e.g. Prisma Client)
  db         Manage your database schema and lifecycle
  migrate    Migrate your database
  studio     Browse your data with Prisma Studio
  validate   Validate your Prisma schema
  format     Format your Prisma schema
Flags
  --preview-feature Run Preview Prisma commands
```

Нужно настроить проект Prisma с помощью **команды prisma init**.

Эта команда делает две вещи: создает **новый каталог** с именем `prisma`, который содержит **файл с именем schema.prisma**, и создает **файл .env** в корневом каталоге проекта, который используется для определения переменных среды (строки подключения к БД).

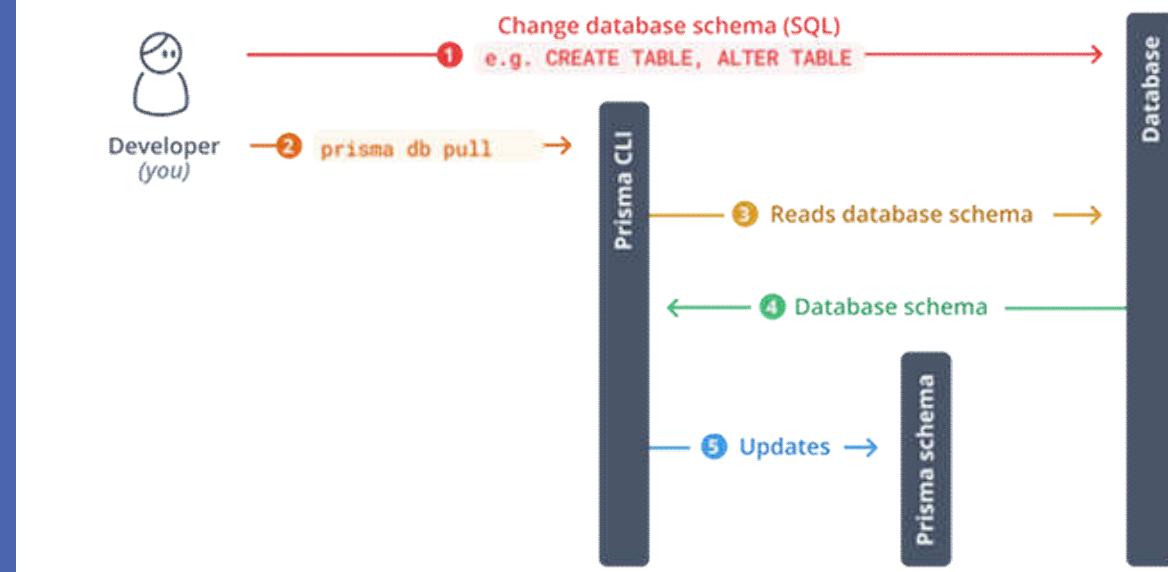
```
PS D:\NodeJS\samples\cwp_32> npx prisma init --datasource-provider sqlserver
✓ Your Prisma schema was created at prisma/schema.prisma
You can now open it in your favorite editor.

Next steps:
1. Set the DATABASE_URL in the .env file to point to your existing database. If your database has no tables yet, read https://pris.ly/d/getting-started
2. Run prisma db pull to turn your database schema into a Prisma schema.
3. Run prisma generate to generate the Prisma Client. You can then start querying your database.

More information in our documentation:
https://pris.ly/d/getting-started
```

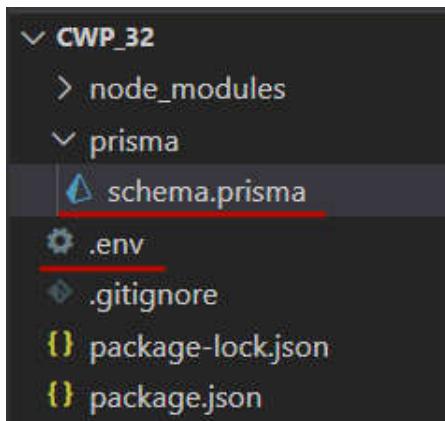
Интроспекция = (самоанализ)

это процесс **сбора информации о БД**.
Самоанализ часто используется для создания начальной версии модели данных при добавлении Prisma в существующий проект.
У самоанализа в Prisma есть одна основная функция: заполнить схему Prisma моделью данных, которая отражают текущую схему БД.



Настройка проекта

2. Настройка связи с БД



```
prisma > schema.prisma > ...
1 // This is your Prisma schema file,
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 generator client {
5   provider = "prisma-client-js"
6 }
7
8 datasource db {
9   provider = "sqlserver"
10  url      = env("DATABASE_URL")
11 }
```

Необходимо проверить, что в schema.prisma указан правильный провайдер, а затем указать строку подключения в .env.

```
.env
1 # Environment variables declared in this file are automatically made available to Prisma.
2 # See the documentation for more detail: https://pris.ly/d/prisma-schema#accessing-environment-variables-from-the-schema
3
4 # Prisma supports the native connection string format for PostgreSQL, MySQL, SQLite, SQL Server, MongoDB and CockroachDB.
5 # See the documentation for all the connection string options: https://pris.ly/d/connection-strings
6
7 DATABASE_URL="sqlserver://localhost:1433;database=Univer;user=sa;password=123;trustServerCertificate=true;connection_limit=5;connectTimeout=10"
```

Настройка проекта

3. Генерация моделей:

```
PS D:\NodeJS\samples\cwp_32> npx prisma db pull
Prisma schema loaded from prisma\schema.prisma
Environment variables loaded from .env
Datasource "db": SQL Server database

✓ Introspected 6 models and wrote them into prisma\schema.prisma
Run prisma generate to generate Prisma Client.

PS D:\NodeJS\samples\cwp_32>
```

Вот общий обзор шагов, которые **команда db pull** выполняет внутри:

- читает URL-адрес подключения из конфигурации;
- открывает соединение с БД;
- интроспектирует (или собирает информацию) схему БД;
- преобразует схему БД в модель данных Prisma;
- записывает модель данных в схему Prisma или обновляет существующие.

Интроспекция БД с помощью команды **prisma db pull**.

```
prisma > schema.prisma > ...
1 generator client {
2   provider = "prisma-client-js"
3 }
4
5 datasource db {
6   provider = "sqlserver"
7   url      = env("DATABASE_URL")
8 }
9
10 model Auditorium {
11   auditorium        String      @id(map: "PK_Auditorium_9ADEC5635B13F344") @db.NVarChar(10)
12   auditorium_name   String?
13   auditorium_capacity Int?
14   auditorium_type   String      @db.NVarChar(10)
15   Auditorium_type   Auditorium_type @relation(fields: [auditorium_type], references: [auditorium_type],
16 )
17
18 model Auditorium_type {
19   auditorium_type   String      @id(map: "AUDITORIUM_TYPE_PK") @db.NVarChar(10)
20   auditorium_typename String    @db.NVarChar(30)
21   Auditorium        Auditorium[]
22 }
23
```

Настройка проекта

4. Генерация клиента

Первым шагом при использовании Prisma Client является установка [пакета @prisma/client](#).

```
PS D:\NodeJS\samples\cwp_32> npm install @prisma/client
added 2 packages, and audited 5 packages in 10s

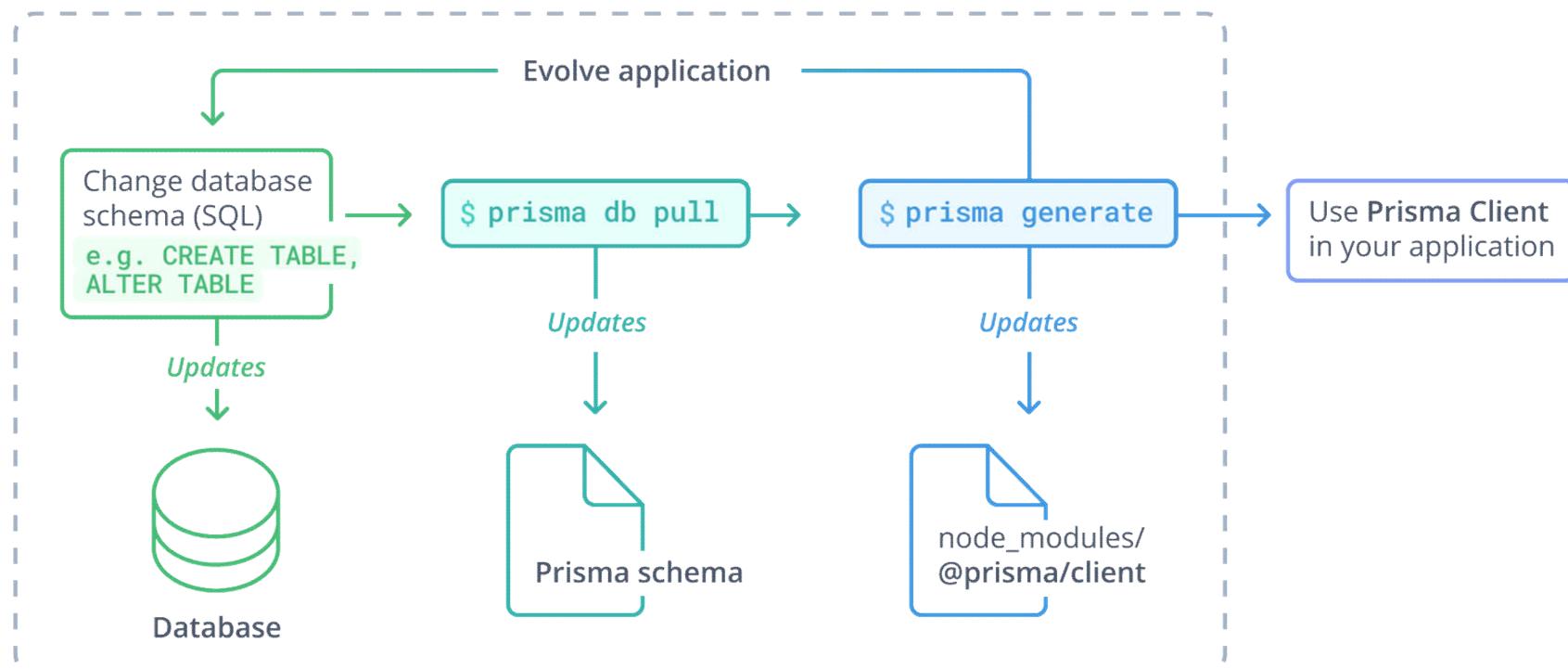
found 0 vulnerabilities
PS D:\NodeJS\samples\cwp_32> npx prisma generate
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma

✓ Generated Prisma Client (4.10.1 | library) to .\node_modules\@prisma\client in 700ms
You can now start using Prisma Client in your code. Reference: https://pris.ly/d/client
```
import { PrismaClient } from '@prisma/client'
const prisma = new PrismaClient()
```

Затем необходимо вызвать [команду prisma generate](#), которая считывает схему Prisma и генерирует код клиента Prisma. По умолчанию код генерируется в папку node\_modules/.prisma/client.

В случае, если модель данных изменяется, нужно вручную повторно сгенерировать клиент Prisma.

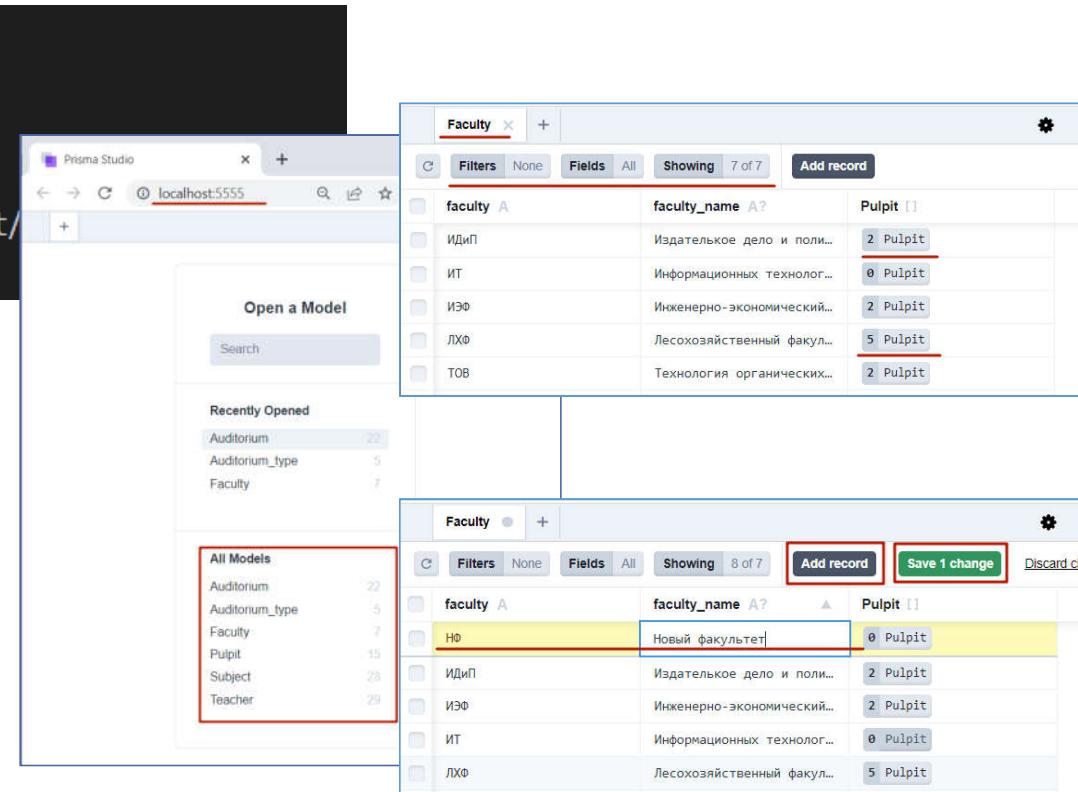
# Генерация моделей путем интроспекции



# Prisma Studio – это визуальный редактор данных в БД.

```
PS D:\NodeJS\samples\cwp_32> npx prisma studio
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma
Prisma Studio is up on http://localhost:5555
imports from "@prisma/client/runtime" are deprecated.
Use "@prisma/client/runtime/library", "@prisma/client/
r "@prisma/client/runtime/binary"
```

Запустить визуальный редактор  
МОЖНО С ПОМОЩЬЮ **команды**  
**prisma studio**.



The screenshot shows the Prisma Studio interface with two main windows. The left window is a sidebar titled 'Open a Model' with a list of models: Auditorium, Auditorium\_type, Faculty, and others. The right window is a detailed view of the 'Faculty' model. It has tabs for 'Filters', 'Fields', and 'Showing 7 of 7'. A table lists faculty entries with columns for 'faculty' and 'faculty\_name'. The first entry, 'faculty\_A', is selected. The second column shows 'faculty\_name' and 'Pulpit'. The bottom part of the window shows a form for adding a new record, with 'faculty\_name' set to 'НФ' and 'Pulpit' set to '0'. Buttons for 'Add record', 'Save 1 change', and 'Discard changes' are at the bottom.

| faculty   | faculty_name                       | Pulpit   |
|-----------|------------------------------------|----------|
| faculty_A | Издательское дело и полиграфия     | 2 Pulpit |
| ИДиП      | Информационных технологий          | 0 Pulpit |
| ИТ        | Инженерно-экономический факультет  | 2 Pulpit |
| ИЭФ       | Лесохозяйственный факультет        | 5 Pulpit |
| ЛХФ       | Технология органических материалов | 2 Pulpit |
| ТОВ       |                                    |          |

# Выборка всех данных

```
const { PrismaClient } = require('@prisma/client');

const prisma = new PrismaClient()

async function main() {
 const allFaculties = await prisma.faculty.findMany();
 console.log(allFaculties);
}

main()
 .then(async () => {
 await prisma.$disconnect();
 })
 .catch(async (e) => {
 console.error(e);
 await prisma.$disconnect();
 })

```

Экземпляр PrismaClient  
лениво подключается при  
первом запросе к API  
(\$connect() вызывается  
автоматически).

Метод **findMany**  
возвращает все записи,  
соответствующие  
заданному критерию.

```
PS D:\NodeJS\samples\cwp_32> node 32-01
[
 {
 faculty: 'ИДИП',
 faculty_name: 'Издательское и полиграфия'
 },
 {
 faculty: 'ИЭФ',
 faculty_name: 'Инженерно-экономический факультет'
 },
 {
 faculty: 'ЛХФ',
 faculty_name: 'Лесохозяйственный факультет'
 },
 {
 faculty: 'ТОВ',
 faculty_name: 'Технология органических веществ'
 },
 {
 faculty: 'ТТИП',
 faculty_name: 'Технология и техника лесной промышленности'
 },
 {
 faculty: 'ХТИТ',
 faculty_name: 'Химическая технология и техника'
 }
]
```

PrismaClient подключается и  
отключается от источника данных  
двумя следующими способами:  
**\$connect()**, **\$disconnect()**. PrismaClient  
автоматически отключается, когда  
процесс Node.js завершается.

# Выборка данных по критерию поиска

```
async function main() {
 const auditoriums = await prisma.auditorium.findMany({
 where: {
 auditorium_type: 'ЛК'
 },
 })
 console.log(auditoriums);
}
```

Свойство **where** используется для фильтрации всех записей.

```
PS D:\NodeJS\samples\cwp_32> node 32-02
[
 {
 auditorium: '?',
 auditorium_name: '???',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК'
 },
 {
 auditorium: '02Б-4',
 auditorium_name: '02Б-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК'
 },
 {
 auditorium: '103-4',
 auditorium_name: '103-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК'
 }
]
```

# Выборка первой записи из набора

```
async function main() {
 const pulpit = await prisma.pulpit.findFirst({
 where: { faculty: 'ИЭФ' },
 })
 console.log(pulpit);
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-19
{
 pulpit: 'МиЭП',
 pulpit_name: 'Менеджмента и экономики природопользования',
 faculty: 'ИЭФ'
}
```

Метод **findFirst** возвращает первую запись, соответствующую заданному критерию.

Метод **findFirstOrThrow** делает то же, что и `findFirst`. Однако, если запись не найдена, то выбрасывается `NotFoundError`.

```
findMany / findFirst / findFirstOrThrow ({
 where?: condition,
 select?: fields,
 include?: relations,
 rejectOnNotFound?: boolean,
 distinct?: field,
 orderBy?: order,
 cursor?: position,
 skip?: number,
 take?: number
})
```

# Выборка уникальной записи

```
async function main() {
 const auditorium = await prisma.auditorium.findUnique({
 where: {
 auditorium: '313-1',
 },
 select: {
 auditorium_name: true,
 auditorium_capacity: true
 }
 })
 console.log(auditorium);
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-03
{ auditorium_name: '313-1', auditorium_capacity: 60 }
PS D:\NodeJS\samples\cwp_32>
```

Метод `findUnique` позволяет извлекать единичные записи по идентификатору или уникальному полю. Метод `findUniqueOrThrow` делает то же, что и `findUnique`. Однако, если запись не найдена, то выбрасывается `NotFoundError`.

Свойство `select` позволяет выбрать набор полей модели, которые необходимо вернуть (по умолчанию возвращаются только скалярные поля).

```
findUnique({
 where: condition,
 select?: fields,
 include?: relations,
 rejectOnNotFound?: boolean,
})
```

# Выборка связанных записей

```
prisma > schema.prisma > Pulpit
24 model Faculty {
25 faculty String @id(map: "PK_FACULTY") @db.NVarChar(10)
26 faculty_name String? @db.NVarChar(50)
27 Pulpit Pulpit[]
28 }
```

Для выборки связанных записей в модели должно быть указано, что значением поля является список.

```
async function main() {
 const faculties = await prisma.faculty.findMany({
 include: {
 Pulpit: true
 }
 })
 console.dir(faculties, { depth: null });
}
```

Свойство `include` позволяет включить в результирующий набор связанные записи (по сути сделать `join`).



```
PS D:\NodeJS\samples\cwp_32> node 32-04
[{
 faculty: 'ИДИП',
 faculty_name: 'Издательское дело и полиграфия',
 Pulpit: [
 {
 pulpits: 'ИСИТ',
 pulpits_name: 'Информационный систем и технологий',
 faculty: 'ИДИП'
 },
 {
 pulpits: 'ПОИСОИ',
 pulpits_name: 'Полиграфического оборудования и систем обработки информации',
 faculty: 'ИДИП'
 }
],
 faculty: 'ИЭФ',
 faculty_name: 'Инженерно-экономический факультет',
 Pulpit: [
 {
 pulpits: 'МиЭП',
 pulpits_name: 'Менеджмента и экономики природопользования',
 faculty: 'ИЭФ'
 }
]
}]
```

# Выборка связанных записей

```
async function main() {
 const faculties = await prisma.faculty.findUnique({
 where: {
 faculty: 'ИЭФ'
 },
 include: {
 Pulpit: {
 select: {
 pulpit_name: true,
 }
 }
 }
 })
 console.dir(faculties, { depth: null });
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-06
{
 faculty: 'ИЭФ',
 faculty_name: 'Инженерно-экономический факультет',
 Pulpit: [
 { pulpit_name: 'Менеджмента и экономики природопользования' },
 { pulpit_name: 'экономической теории и маркетинга' }
]
}
```

**include => select** используется для выборки полей только из связанной модели.  
Из исходной модели будут выбраны все скалярные поля.

# Выборка связанных записей

```
async function main() {
 const faculties = await prisma.faculty.findUnique({
 where: {
 faculty: 'ИЭФ'
 },
 select: {
 faculty_name: true,
 Pulpit: {
 select: {
 pulpit_name: true,
 },
 orderBy: {
 pulpit_name: 'desc',
 },
 },
 },
 })
 console.dir(faculties, { depth: null });
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-07
{
 faculty_name: 'Инженерно-экономический факультет',
 Pulpit: [
 { pulpit_name: 'экономической теории и маркетинга' },
 { pulpit_name: 'Менеджмента и экономики природопользования' }
]
}
```

Для того, чтобы выбрать поля исходной модели и поля связанной модели, используется **вложенный select** (`select => select`, в таком случае `include` не нужен).

**Свойство orderBy** позволяет сортировать записи по возрастанию или убыванию.

# Выборка данных (filter conditions)

```
async function main() {
 const faculties = await prisma.faculty.findMany({
 where: {
 faculty_name: {
 contains: 'технолог',
 // mode: 'insensitive'
 }
 }
 })
 console.dir(faculties, { depth: null });
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-08
[
 { faculty: 'ТОВ', faculty_name: 'Технология органических веществ' },
 { faculty: 'ТЛП', faculty_name: 'Технология и техника лесной промышленности' },
 { faculty: 'ХТИТ', faculty_name: 'Химическая технология и техника' }
]
```

Условие **contains** используется для выборки по части содержимого поля (полнотекстовый поиск).

`== LIKE '%some_text%'`

# Выборка данных (filter conditions)

```
async function main() {
 const faculties = await prisma.faculty.findMany({
 where: {
 faculty: { in: ['ИЭФ', 'ТОВ'] }
 },
 include: {
 Pulpit: {
 select: {
 pulpit: true
 }
 }
 }
 })
 console.dir(faculties, { depth: null });
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-05
[{
 faculty: 'ИЭФ',
 faculty_name: 'Инженерно-экономический факультет',
 Pulpit: [{ pulpit: 'МиЭП' }, { pulpit: 'ЭТИМ' }]
},
{
 faculty: 'ТОВ',
 faculty_name: 'Технология органических веществ',
 Pulpit: [{ pulpit: 'ОХ' }, { pulpit: 'ТНХСИПМ' }]
}]
```

Условие `in` позволяет задавать несколько значений поля для выборки.

# Выборка данных (filter operators)

```
async function main() {
 const auditoriums = await prisma.auditorium.findMany({
 where: {
 AND: [
 {
 auditorium_name: {
 endsWith: '-4',
 },
 auditorium_type: 'ЛК'
 }
],
 }
 })
 console.log(auditoriums)
}
```

Операторы **AND** (все условия должны возвращать значение true, используется по умолчанию ), **OR** (одно или несколько условий должны возвращать значение true), **NOT** (все условия должны возвращать false).

```
PS D:\NodeJS\samples\cwp_32> node 32-13
[
 {
 auditorium: '02Б-4',
 auditorium_name: '02Б-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК'
 },
 {
 auditorium: '103-4',
 auditorium_name: '103-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК'
 },
 {
 auditorium: '105-4',
 auditorium_name: '105-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК'
 }
]
```

# Filter conditions & operations

- equals
- not
- in
- notIn
- lt
- lte
- gt
- gte
- contains
- search
- mode
- startsWith
- endsWith
- AND
- OR
- NOT

[Подробнее](#)

# Выборка данных (relation fields) фильтр на связь «-КО-МНОГИМ»

|       |               |                        |   |                   |   |
|-------|---------------|------------------------|---|-------------------|---|
| where | auditorium... | gt                     | v | 100               | x |
|       | auditorium A  | auditorium_capacity #? |   | auditorium_type A |   |
|       | 429-4         | 110                    |   | ЛК                |   |

ожидаемый результат

|                   |
|-------------------|
| auditorium_type A |
| ЛБ-Х              |
| ЛБ-К              |
| ЛБ-СК             |
| ЛК                |
| ЛК-К              |

```
//Filter on "-to-many" relations
const auditoriumTypes = await prisma.auditorium_type.findMany({
 where: {
 Auditorium: {
 none: {
 auditorium_capacity: {
 gt: 100,
 },
 }
 }
 }
})
```

```
PS D:\NodeJS\samples\cwp_32> node 32-09
[
 {
 auditorium_type: 'ЛБ-Х',
 auditorium_typename: 'Химическая лаборатория'
 },
 {
 auditorium_type: 'ЛБ-К',
 auditorium_typename: 'Компьютерный класс'
 },
 {
 auditorium_type: 'ЛБ-СК',
 auditorium_typename: 'Спец. компьютерный класс'
 },
 {
 auditorium_type: 'ЛК-К',
 auditorium_typename: 'Лекционная с уст. компьютерами'
 }
]
```

Фильтр **none** возвращает все записи, в которых ни одна связанная запись не соответствует критериям фильтрации.

# Выборка данных (relation fields)

## фильтр на связь «-КО-МНОГИМ»

```
const faculties = await prisma.faculty.findMany({
 where: {
 Pulpit: {
 none: {}, // вернет факультеты без кафедр
 },
 }
})
```

```
PS D:\NodeJS\samples\cwp_32> node 32-11
[{ faculty: 'ИТ', faculty_name: 'Информационных технологий' }]
PS D:\NodeJS\samples\cwp_32>
```

Если передать в фильтр **пустой объект**, то будут искааться записи исходной модели, у которых нет указанных связанных записей.

# Выборка данных (relation fields)

## фильтр на связь «-ко-многим»

| auditorium_capacity | auditorium_type |
|---------------------|-----------------|
| 15                  | ЛБ-К            |
| 90                  | ЛБ-К            |
| 15                  | ЛБ-К            |
| 15                  | ЛБ-К            |
| 90                  | ЛБ-К            |
| 110                 | ЛК              |
| 90                  | ЛК              |
| 60                  | ЛК              |
| 60                  | ЛК              |
| 90                  | ЛК              |
| 90                  | ЛК              |
| 50                  | ЛК              |
| 90                  | ЛК              |
| 90                  | ЛК              |
| 90                  | ЛК              |
| 30                  | ЛК              |
| 30                  | ЛК              |
| 90                  | ЛК              |
| 90                  | ЛК              |
| 90                  | ЛК-К            |

ожидаемый результат

```
//Filter on "-to-many" relations
const auditoriumTypes = await prisma.auditorium_type.findMany({
 where: {
 Auditorium: {
 every: {
 auditorium_capacity: {
 lte: 40,
 }
 }
 }
 }
})
```

```
PS D:\NodeJS\samples\cwp_32> node 32-09
[
 {
 auditorium_type: 'ЛБ-Х',
 auditorium_typename: 'Химическая лаборатория'
 },
 {
 auditorium_type: 'ЛБ-СК',
 auditorium_typename: 'Спец. компьютерный класс'
 }
]
```

Фильтр **every** возвращает все записи, в которых все связанные записи соответствует критериям фильтрации.

# Выборка данных (relation fields)

## фильтр на связь «-ко-многим»

| ожидаемый результат |                      |
|---------------------|----------------------|
| where               | auditorium... gt 100 |
| auditorium_capacity | #?                   |

110      ЛК

```
const auditoriumTypes = await prisma.auditorium_type.findMany({
 where: {
 Auditorium: {
 some: {
 auditorium_capacity: {
 gt: 100
 }
 }
 }
 }
})
```

```
PS D:\NodeJS\samples\cwp_32> node 32-09
[{ auditorium_type: 'ЛК', auditorium_typename: 'Лекционная' }]
PS D:\NodeJS\samples\cwp_32>
```

Фильтр **some** возвращает все записи, в которых хотя бы одна связанная запись соответствует критериям фильтрации.

# Выборка данных (relation fields)

## фильтр на связь «-к-одному»

```
//Filter on "-to-one" relations
const auditorium = await prisma.auditorium.findMany({
 where: {
 Auditorium_type: {
 is: {
 auditorium_type: 'ЛБ-К'
 }
 }
 }
})
```

Фильтр `is` возвращает все записи, в которых связанная запись соответствует критериям фильтрации.

```
PS D:\NodeJS\samples\cwp_32> node 32-10
[
 {
 auditorium: '206-1',
 auditorium_name: '206-1',
 auditorium_capacity: 15,
 auditorium_type: 'ЛБ-К'
 },
 {
 auditorium: '301-1',
 auditorium_name: '301-1',
 auditorium_capacity: 15,
 auditorium_type: 'ЛБ-К'
 },
 {
 auditorium: '304-4',
 auditorium_name: '304-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛБ-К'
 },
 {
 auditorium: '413-1',
 auditorium_name: '413-1',
 auditorium_capacity: 15,
 auditorium_type: 'ЛБ-К'
 },
 {
 auditorium: '423-1',
 auditorium_name: '423-1',
 auditorium_capacity: 90,
 auditorium_type: 'ЛБ-К'
 }
]
```

# Выборка данных (relation fields)

## фильтр на связь «к-одному»

```
//Filter on "-to-one" relations
const auditorium = await prisma.auditorium.findMany({
 where: {
 Auditorium_type: {
 isNot: {
 auditorium_type: { in: ['ЛК', 'ЛБ-К'] }
 }
 }
 }
})
```

```
PS D:\NodeJS\samples\cwp_32> node 32-10
[
 {
 auditorium: '114-4',
 auditorium_name: '114-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК-К'
 }
]
```

Фильтр **isNot** возвращает все записи, в которых связанная запись не соответствует критериям фильтрации.

# Fluent API

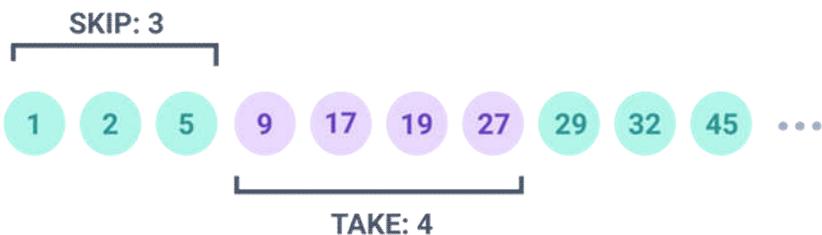
```
async function main() {
 // Fluent API
 const faculty = await prisma.faculty
 .findUnique({ where: { faculty: 'ИЭФ' } })
 .Pulpit()
 console.log(faculty);
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-12
[
 {
 pulpit: 'МиЭП',
 pulpit_name: 'Менеджмента и экономики природопользования',
 faculty: 'ИЭФ'
 },
 {
 pulpit: 'ЭТИМ',
 pulpit_name: 'экономической теории и маркетинга',
 faculty: 'ИЭФ'
 }
]
```

Fluent API – потоковый интерфейс (цепочка методов). **Последний вызов** функции **определяет возвращаемый тип** всего запроса (в примере pulpit). Единственное требование для цепочки состоит в том, что **предыдущий вызов** функции **должен возвращать только один объект**.

# Выборка данных (пагинация)

Для постраничного вывода (или пагинации) используются свойства `skip` и `take`.



```
async function main() {
 const pulpits = await prisma.pulpit.findMany({
 skip: 5,
 take: 4,
 })
 console.log(pulpits)
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-15
[
 { pulpit: 'ЛУ', pulpit_name: 'лесоустройства', faculty: 'ЛХФ' },
 { pulpit: 'МиЭП',
 pulpit_name: 'Менеджмента и экономики природопользования',
 faculty: 'ИЭФ' },
 { pulpit: 'ОВ', pulpit_name: 'охотоведения', faculty: 'ЛХФ' },
 { pulpit: 'ОХ', pulpit_name: 'Органической химии', faculty: 'ТОВ' }
]
```

Свойство `skip: n` позволяет пропустить  $n$  элементов.

Свойство `take: m` позволяет «взять»  $m$  элементов.

# Выборка данных (подсчет записей)

Свойство `_count` используется для подсчета кол-ва записей. Может использоваться внутри `include` или `select` верхнего уровня. Кроме того, может использоваться в сортировке по кол-ву связанных записей.

```
async function main() {
 const faculties = await prisma.faculty.findMany({
 take: 5,
 orderBy: {
 Pulpit: {
 _count: 'desc',
 },
 },
 include: {
 _count: {
 select: { Pulpit: true }
 }
 }
 })
 console.dir(faculties, { depth: null });
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-14
[
 {
 faculty: 'ЛХФ',
 faculty_name: 'Лесохозяйственный факультет',
 _count: { Pulpit: 5 }
 },
 {
 faculty: 'ТОВ',
 faculty_name: 'Технология органических веществ',
 _count: { Pulpit: 2 }
 },
 {
 faculty: 'ТЛП',
 faculty_name: 'Технология и техника лесной промышленности',
 _count: { Pulpit: 2 }
 },
 {
 faculty: 'ХТИ',
 faculty_name: 'Химическая технология и техника',
 _count: { Pulpit: 2 }
 },
 {
 faculty: 'ИДИП',
 faculty_name: 'Издательское дело и полиграфия',
 _count: { Pulpit: 2 }
 }
]
```

`include => _count => select => relation` – подсчет кол-ва только указанных связанных записей

# Выборка данных (подсчет данных)

```
async function main() {
 const facultiesCount = await prisma.faculty.count()
 console.log(facultiesCount)

 const faculties = await prisma.faculty.findMany({
 select: {
 faculty: true,
 _count: true
 }
 })
 console.dir(faculties, { depth: null })
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-18
7
[
 { faculty: 'ИДиП', _count: { Pulpit: 2 } },
 { faculty: 'ИТ', _count: { Pulpit: 0 } },
 { faculty: 'ИЭФ', _count: { Pulpit: 2 } },
 { faculty: 'ЛХФ', _count: { Pulpit: 5 } },
 { faculty: 'ТОВ', _count: { Pulpit: 2 } },
 { faculty: 'ТТЛП', _count: { Pulpit: 2 } },
 { faculty: 'ХТИТ', _count: { Pulpit: 2 } }
]
```

**select => \_count: true** – подсчет кол-ва всех связанных записей

# Выборка данных (подсчет данных)

```
async function main() {
 const faculties = await prisma.faculty.findMany({
 select: {
 faculty: true,
 Pulpit: {
 select: {
 pulpIt: true,
 _count: true
 }
 }
 }
 })
 console.dir(faculties, { depth: null })
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-20
[{
 faculty: 'ИДиП',
 Pulpit: [
 { pulpIt: 'ИСИТ', _count: { Subject: 14, Teacher: 14 } },
 { pulpIt: 'ПОИСОИ', _count: { Subject: 2, Teacher: 2 } }
],
 faculty: 'ИТ',
 Pulpit: []
},
{
 faculty: 'ИЭФ',
 Pulpit: [
 { pulpIt: 'МиЭП', _count: { Subject: 1, Teacher: 1 } },
 { pulpIt: 'ЭТИМ', _count: { Subject: 1, Teacher: 1 } }
],
}
```

**select => relation => select => \_count: true** – подсчет кол-ва всех связанных с relation записей)

# Агрегатные функции

```
async function main() {
 const aggregationCount = await prisma.auditorium.aggregate({
 where: {
 auditorium_type: 'ЛК'
 },
 _count: {
 auditorium: true,
 },
 })
 console.log('Count of auditoriums: ' + aggregationCount._count.auditorium);

 const aggregationAvg = await prisma.auditorium.aggregate({
 where: {
 auditorium_type: 'ЛК'
 },
 _avg: {
 auditorium_capacity: true,
 }
 })
 console.log('Average capacity: ' + aggregationAvg._avg.auditorium_capacity);

 const aggregationSum = await prisma.auditorium.aggregate({
 where: {
 auditorium_type: 'ЛК'
 },
 _sum: {
 auditorium_capacity: true,
 }
 })
 console.log('Summary capacity: ' + aggregationSum._sum.auditorium_capacity);
```

```
PS D:\NodeJS\samples\cwp_32> node 32-16
Count of auditoriums: 16
Average capacity: 77.5
Summary capacity: 1240
PS D:\NodeJS\samples\cwp_32>
```

Свойства для агрегирования:

- **\_count**: возвращает количество записей или непустых полей;
- **\_avg**: возвращает среднее значение всех значений указанного поля;
- **\_sum**: возвращает сумму всех значений указанного поля;
- **\_min**: возвращает наименьшее значение указанного поля;
- **\_max**: возвращает наибольшее значение указанного поля.

```
aggregate({
 where?: UserWhereInput
 orderBy?: XOR<Enumerable<UserOrderByInput>, UserOrderByInput>
 cursor?: UserWhereUniqueInput
 take?: number
 skip?: number
 distinct?: Enumerable<UserDistinctFieldEnum>
 _count?: true | UserCountAggregateInputType
 _avg?: UserAvgAggregateInputType
 _sum?: UserSumAggregateInputType
 _min?: UserMinAggregateInputType
 _max?: UserMaxAggregateInputType
})
```

# Группировка данных

```
async function main() {
 const groupAuditoriums = await prisma.auditorium.groupBy({
 by: ['auditorium_type'],
 _sum: {
 auditorium_capacity: true,
 },
 having: {
 auditorium_capacity: {
 min: {
 gte: 30,
 },
 },
 },
 })
 console.dir(groupAuditoriums, { depth: null })
}
```

```
[
 { _sum: { auditorium_capacity: 225 }, auditorium_type: 'ЛБ-К' },
 { _sum: { auditorium_capacity: 1240 }, auditorium_type: 'ЛК' },
 { _sum: { auditorium_capacity: 90 }, auditorium_type: 'ЛК-К' }
]
```

```
PS D:\NodeJS\samples\cwp_32> node 32-17
[
 { _sum: { auditorium_capacity: 1240 }, auditorium_type: 'ЛК' },
 { _sum: { auditorium_capacity: 90 }, auditorium_type: 'ЛК-К' }
]
```

**by** — определяет поле или комбинацию полей для группировки записей

**having** — позволяет фильтровать группы по агрегируемому значению

**groupBy** – группировка записей по одному или нескольким полям).

```
groupBy ({
 by?: by,
 having?: having,
 where?: condition,
 orderBy?: order,
 skip?: number,
 take?: number,
 _count: count,
 _avg: avg,
 _sum: sum,
 _min: min,
 _max: max
})
```

# Выборка данных

```
async function main() {
 const auditoriums = await prisma.auditorium.findMany({
 distinct: ['auditorium_type'],
 })
 console.log(auditoriums)
}
```

**distinct** – возвращает только уникальные значения по одному или нескольким полям

```
PS D:\NodeJS\samples\cwp_32> node 32-21
[
 {
 auditorium: '?',
 auditorium_name: '????',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК'
 },
 {
 auditorium: '114-4',
 auditorium_name: '114-4',
 auditorium_capacity: 90,
 auditorium_type: 'ЛК-К'
 },
 {
 auditorium: '206-1',
 auditorium_name: '206-1',
 auditorium_capacity: 15,
 auditorium_type: 'ЛБ-К'
 }
]
```

# Создание одной записи

| faculty | faculty_name                |
|---------|-----------------------------|
| ИдиП    | Издательское дело и поли... |
| ИЭФ     | Инженерно-экономический...  |
| ЛХФ     | Лесохозяйственный факул...  |
| ТОВ     | Технология                  |
| ТТЛП    | Технология                  |
| ХТИТ    | Химическая                  |

```
async function main() {
 // one record
 const result = await prisma.faculty.create({
 data: {
 faculty: "ИТ",
 faculty_name: "Информационных технологий"
 },
 })
 console.log(result);
}
```

В **data** указываются данные создаваемой записи.

```
PS D:\NodeJS\samples\cwp_32> node 32-22
{ faculty: 'ИТ', faculty_name: 'Информационных технологий' }
```

**create** – создание одной записи

```
create ({
 data: _data,
 select?: fields,
 include?: relations
})
```

# Создание связанных записей

```
async function main() {
 // nested write
 const result = await prisma.faculty.create({
 data: {
 faculty: "НФ",
 faculty_name: "Новый факультет",
 Pulpit: {
 create: { pulpit: 'НК', pulpit_name: 'Новая кафедра' }
 },
 include: { Pulpit: true }
 }
 })
 console.log(result);
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-23
{
 faculty: 'НФ',
 faculty_name: 'Новый факультет',
 Pulpit: [{ pulpit: 'НК', pulpit_name: 'Новая кафедра', faculty: 'НФ' }]
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-23
{ faculty: 'НФ', faculty_name: 'Новый факультет' } без include
PS D:\NodeJS\samples\cwp_32> node 32-24
```

create => create/createMany – создание одной или нескольких связанных записей,  
вложенная запись

# Создание нескольких записей

```
async function main() {
 // multiple records and nested write
 await prisma.faculty.createMany({
 data: [
 {
 faculty: "ВФ",
 faculty_name: "Военный"
 },
 {
 faculty: "ММ",
 faculty_name: "Механико-математический",
 // в createMany нельзя делать вложенную запись
 // Pulpit: {
 // create: { pulpit: 'ПМ', pulpit_name: 'Прикладной математики' }
 // }
 },
],
 })
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-24
{ count: 2 }
```

**skipDuplicates** – при значении true  
создаются только уникальные записи.

**createMany** – создание нескольких записей, вложенная запись невозможна

```
createMany ({
 data: _data[],
 skipDuplicates?: boolean
})
```

# Создание записи с присоединением существующей записи

```
async function main() {
 // connect an existing record
 const result = await prisma.pulpit.create({
 data: {
 pulpit: 'ПМ',
 pulpit_name: 'Прикладной математики',
 Faculty: {
 connect: { faculty: 'ММ' },
 }
 }
 })
 console.log(result);
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-25
{ pulpit: 'ПМ', pulpit_name: 'Прикладной математики', faculty: 'ММ' }
```

**create => connect** – присоединение существующей записи

В **createMany** нельзя так сделать.

# Создание записи с присоединением к существующей или созданием новой

```
async function main() {
 // connect an existing record or create new
 const result = await prisma.pulpit.create({
 data: {
 pulpit: 'КС',
 pulpit_name: 'Кафедра связи',
 Faculty: {
 connectOrCreate: {
 where: {
 faculty: 'ВФ',
 },
 create: {
 faculty: 'ВФ',
 faculty_name: 'Военный факультет'
 }
 }
 }
 }
 })
 console.log(result);
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-26
{ pulpit: 'КС', pulpit_name: 'Кафедра связи', faculty: 'ВФ' }
```

`create => connectOrCreate` – присоединение существующей записи или создание новой, если ранее указанной не существует.

# Обновление одной записи

```
async function main() {
 const updatedFaculty = await prisma.faculty.update({
 where: { faculty: "НФ" },
 data: { faculty_name: "Новый факультет 2" },
 })
 console.log(updatedFaculty)
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-27
{ faculty: 'НФ', faculty_name: 'Новый факультет 2' }
```

**update** – изменяет одну указанную запись (первую подходящую под условие)

# Обновление нескольких записей

```
async function main() {
 const auditoriums = await prisma.auditorium.updateMany({
 data: {
 auditorium_capacity: {
 increment: 100,
 // decrement: 100
 }
 },
 })
 console.log(auditoriums)
}
```

PS D:\NodeJS\samples\cwp\_32> node 32-34  
{ count: 24 }

`updateMany` – изменяет все записи, подходящие под критерий выборки

# Обновление записи или ее создание

```
async function main() {
 const updatedFaculty = await prisma.faculty.upsert({
 where: {
 faculty: 'НФ2'
 },
 create: {
 faculty: 'НФ2',
 faculty_name: 'Новый факультет 3'
 },
 update: {
 faculty_name: 'Новый факультет 3'
 }
 })
 console.log(updatedFaculty)
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-28
{ faculty: 'НФ2', faculty_name: 'Новый факультет 3' }
```

**upsert** – изменяет указанную запись, если она уже существует, или создает новую, если указанной нет

# Обновление записей с удалением связанный (-ых) записи (-ей)

```
async function main() {
 const updatedFaculty = await prisma.faculty.update({
 where: {
 faculty: 'НФ'
 },
 data: {
 Pulpit: {
 deleteMany: {
 pulpits: 'НК',
 }
 }
 },
 select: {
 Pulpit: true,
 }
 })
 console.log(updatedFaculty)
}
```

| pulpit A | pulpit_name A?      | faculty A? |
|----------|---------------------|------------|
| НК       | Новая кафедра       | НФ         |
| CHK      | Самая новая кафедра | НФ         |

```
PS D:\NodeJS\samples\cwp_32> node 32-29
{
 Pulpit: [
 {
 pulpits: 'CHK',
 pulpits_name: 'Самая новая кафедра',
 faculty: 'НФ'
 }
]
}
```

update => deleteMany/delete – удаление одной или нескольких связанных записей

# Обновление записей с обновлением связанных записей

```
async function main() {
 const updatedAuditoriums = await prisma.auditorium_type.update({
 where: {
 auditorium_type: 'ЛК'
 },
 data: {
 Auditorium: {
 updateMany: {
 where: {
 auditorium_capacity: 90,
 },
 data: {
 auditorium_capacity: 45,
 },
 },
 },
 include: {
 Auditorium: true
 }
 }
 })
 console.log(updatedAuditoriums)
}
```

update => updateMany – обновление нескольких связанных записей

| auditorium A | auditorium_name A? | auditorium_capacity #? | auditorium_type A |
|--------------|--------------------|------------------------|-------------------|
| ?            | ???                | 45                     | ЛК                |
| 026-4        | 026-4              | 45                     | ЛК                |
| 103-4        | 103-4              | 45                     | ЛК                |
| 105-4        | 105-4              | 45                     | ЛК                |
| 107-4        | 107-4              | 45                     | ЛК                |
| 132-4        | 132-4              | 45                     | ЛК                |

| auditorium A | auditorium_name A? | auditorium_capacity #? | auditorium_type A |
|--------------|--------------------|------------------------|-------------------|
| ?            | ???                | 90                     | ЛК                |
| 026-4        | 026-4              | 90                     | ЛК                |
| 103-4        | 103-4              | 90                     | ЛК                |
| 105-4        | 105-4              | 90                     | ЛК                |
| 107-4        | 107-4              | 90                     | ЛК                |
| 132-4        | 132-4              | 90                     | ЛК                |

```
PS D:\NodeJS\samples\cwp_32> node 32-30
{
 auditorium_type: 'ЛК',
 auditorium_typename: 'Лекционная',
 Auditorium: [
 {
 auditorium: '?',
 auditorium_name: '??',
 auditorium_capacity: 45,
 auditorium_type: 'ЛК'
 },
 {
 auditorium: '026-4',
 auditorium_name: '026-4',
 auditorium_capacity: 45,
 auditorium_type: 'ЛК'
 }
]
}
```

# Обновление записей с обновлением одной связанной записи

```
async function main() {
 const updatedAuditorium = await prisma.auditorium_type.update({
 where: {
 auditorium_type: 'ЛБ-К'
 },
 data: {
 Auditorium: {
 update: {
 where: {
 auditorium: '423-1',
 },
 data: {
 auditorium_capacity: 45,
 },
 },
 include: {
 Auditorium: {
 where: {
 auditorium: '423-1'
 }
 }
 }
 }
 })
 console.log(updatedAuditorium)
}
```

|       |       |    |      |
|-------|-------|----|------|
| 423-1 | 423-1 | 90 | ЛБ-К |
|-------|-------|----|------|



```
PS D:\NodeJS\samples\cwp_32> node 32-31
{
 auditorium_type: 'ЛБ-К',
 auditorium_typename: 'Компьютерный класс',
 Auditorium: [
 {
 auditorium: '423-1',
 auditorium_name: '423-1',
 auditorium_capacity: 45,
 auditorium_type: 'ЛБ-К'
 }
]
}
```

update => update – обновление одной  
связанной записи

# Обновление записей с обновлением или созданием связанной записи

```
async function main() {
 const updatedAuditorium = await prisma.auditorium_type.update({
 where: {
 auditorium_type: 'ЛК'
 },
 data: {
 Auditorium: {
 upsert: {
 where: {
 auditorium: '466-4',
 },
 create: {
 auditorium: '466-4',
 auditorium_name: '466-4',
 auditorium_capacity: 120
 },
 update: {
 auditorium_capacity: 120
 }
 },
 },
 include: {
 Auditorium: {
 where: {
 auditorium: '466-4'
 }
 }
 }
 }
 })
 console.log(updatedAuditorium);
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-32
{
 auditorium_type: 'ЛК',
 auditorium_typename: 'Лекционная',
 Auditorium: [
 {
 auditorium: '466-4',
 auditorium_name: '466-4',
 auditorium_capacity: 120,
 auditorium_type: 'ЛК'
 }
]
}
```

update => upsert – изменяет указанную связанную запись, если она существует, иначе создает новую

# Обновление записей с созданием связанный (-ых) записи (-ей)

```
async function main() {
 const facultyWithPulpits = await prisma.faculty.update({
 where: {
 faculty: 'ИТ'
 },
 data: {
 Pulpit: {
 createMany: {
 data: [
 { pulpit: 'НК1', pulpit_name: 'Новая кафедра 1' },
 { pulpit: 'НК2', pulpit_name: 'Новая кафедра 2' }
],
 },
 },
 include: {
 Pulpit: true
 }
 }
 })
 console.log(facultyWithPulpits)
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-33
{
 faculty: 'ИТ',
 faculty_name: 'Информационных технологий',
 Pulpit: [
 { pulpit: 'НК1', pulpit_name: 'Новая кафедра 1', faculty: 'ИТ' },
 { pulpit: 'НК2', pulpit_name: 'Новая кафедра 2', faculty: 'ИТ' }
]
}
```

update => create/createMany – создает одну или несколько связанных записей

# Удаление одной записи

```
async function main() {
 const deletedPulpit = await prisma.pulpit.delete({
 where: {
 pulpit: 'НК1',
 },
 })
 console.log(deletedPulpit)
}
```

```
PS D:\NodeJS\samples\cwp_32> node 32-35
{ pulpit: 'НК1', pulpit_name: 'Новая кафедра 1', faculty: 'ИТ' }
```

**delete** – удаляет одну (первую) указанную запись

# Удаление нескольких записей

```
async function main() {
 const deletedPulpits = await prisma.pulpit.deleteMany({
 where: {
 faculty: 'ИТ',
 },
 })
 console.log(deletedPulpits);

 const allDeletedPulpits = await prisma.pulpit.deleteMany({})
 console.log(allDeletedPulpits)
}
```

```
[...]
PS D:\NodeJS\samples\cwp_32> node 32-36
{ count: 1 }
{ count: 18 }
```

**deleteMany** – удаляет все записи, подходящие под критерий выборки

# Транзакции

Prisma предоставляет следующие варианты использования транзакций:

- **Вложенные записи**: обработка нескольких операций с одной или несколькими связанными записями внутри одной транзакции.
- **Пакетные (batch) / массовые (bulk) транзакции**: массовая обработка одной или нескольких операций с помощью updateMany, deleteMany и createMany.
- API Prisma Client **\$transaction**
  - ✓ **Последовательные операции**: передается массив клиентских запросов для последовательного выполнения внутри транзакции.
  - ✓ **Интерактивные транзакции**: передается функция, которая может содержать пользовательский код, включая запросы Prisma Client, код, отличный от Prisma, и другой поток управления, который будет выполняться в транзакции.

# Транзакции (последовательные операции)

```
async function main() {
 await prisma.$transaction(
 [
 prisma.auditorium.update({
 where: { auditorium: '111-1' },
 data: { auditorium_capacity: 100 }
 }),
 prisma.faculty.create({ data: { faculty: 'МС', faculty_name: 'Машиностроения' } })
],
 {
 isolationLevel: Prisma.TransactionIsolationLevel.Serializable
 }
)
}
```

Необходимо передать **массив клиентских запросов** для последовательного выполнения внутри транзакции

| Database    | Default        |
|-------------|----------------|
| PostgreSQL  | ReadCommitted  |
| MySQL       | RepeatableRead |
| SQL Server  | ReadCommitted  |
| CockroachDB | Serializable   |
| SQLite      | Serializable   |

# Транзакции (интерактивные)

```
async function main() {
 await prisma.$transaction(
 async (tx) => {
 await tx.$executeRaw`UPDATE Auditorium SET auditorium_type = 'ЛК' WHERE auditorium = '111-1';
 // $queryRaw - для SELECT, возвращает результирующий набор

 // ... можно делать какие-то проверки

 await tx.faculty.delete({
 where: {
 faculty: 'МС',
 },
 });

 throw new Error(`Some error`);
 },
 {
 isolationLevel: Prisma.TransactionIsolationLevel.ReadUncommitted,
 maxWait: 5000, // default: 2000
 timeout: 10000, // default: 5000
 }
)
}
```

**maxWait** – максимальное кол-во времени, в течение которого будет ожидаться получение транзакции из Бд.  
**timeout** – максимальное кол-во времени, в течение которого интерактивная транзакция может выполняться до отмены и отката.

Когда приложение достигает конца функции, транзакция фиксируется в Бд. **Если возникнет ошибка, асинхронная функция выдаст исключение и автоматически откатит транзакцию.**

# Необработанные запросы

Для реляционных баз данных Prisma Client предоставляет методы, которые позволяют отправлять необработанные запросы:

- `$queryRaw` для возврата фактических записей (например, с помощью SELECT)
- `$executeRaw` для возврата количества затронутых строк (например, после UPDATE или DELETE)