

# Headline

大事件后台 API 项目，API 接口文档请参考 [https://www.showdoc.cc/escook?page\\_id=3707158761215217](https://www.showdoc.cc/escook?page_id=3707158761215217)

## 1. 初始化

### 1.1 创建项目

1. 新建 `api_server` 文件夹作为项目根目录，并在项目根目录中运行如下的命令，初始化包管理配置文件：

```
npm init -y
```

2. 运行如下的命令，安装特定版本的 `express`：

```
npm i express@4.17.1
```

3. 在项目根目录中新建 `app.js` 作为整个项目的入口文件，并初始化如下的代码：

```
// 导入 express 模块
const express = require('express')
// 创建 express 的服务器实例
const app = express()

// write your code here...

// 调用 app.listen 方法，指定端口号并启动web服务器
app.listen(3007, function () {
  console.log('api server running at http://127.0.0.1:3007')
})
```

### 1.2 配置 cors 跨域

1. 运行如下的命令，安装 `cors` 中间件：

```
npm i cors@2.8.5
```

2. 在 `app.js` 中导入并配置 `cors` 中间件：

```
// 导入 cors 中间件
const cors = require('cors')
// 将 cors 注册为全局中间件
app.use(cors())
```

### 1.3 配置解析表单数据的中间件

1. 通过如下的代码，配置解析 `application/x-www-form-urlencoded` 格式的表单数据的中间件：

```
app.use(express.urlencoded({ extended: false })))
```

## 1.4 初始化路由相关的文件夹

1. 在项目根目录中，新建 `router` 文件夹，用来存放所有的 路由 模块

路由模块中，只存放客户端的请求与处理函数之间的映射关系

2. 在项目根目录中，新建 `router_handler` 文件夹，用来存放所有的 路由处理函数模块

路由处理函数模块中，专门负责存放每个路由对应的处理函数

## 1.5 初始化用户路由模块

1. 在 `router` 文件夹中，新建 `user.js` 文件，作为用户的路由模块，并初始化代码如下：

```
const express = require('express')
// 创建路由对象
const router = express.Router()

// 注册新用户
router.post('/reguser', (req, res) => {
  res.send('reguser OK')
})

// 登录
router.post('/login', (req, res) => {
  res.send('login OK')
})

// 将路由对象共享出去
module.exports = router
```

2. 在 `app.js` 中，导入并使用 用户路由模块：

```
// 导入并注册用户路由模块
const userRouter = require('./router/user')
app.use('/api', userRouter)
```

## 1.6 抽离用户路由模块中的处理函数

目的：为了保证 路由模块 的纯粹性，所有的 路由处理函数，必须抽离到对应的 路由处理函数模块 中

1. 在 `/router_handler/user.js` 中，使用 `exports` 对象，分别向外共享如下两个 路由处理函数：

```
/**
 * 在这里定义和用户相关的路由处理函数，供 /router/user.js 模块进行调用
 */

// 注册用户的处理函数
exports.regUser = (req, res) => {
  res.send('reguser OK')
}

// 登录的处理函数
exports.login = (req, res) => {
  res.send('login OK')
}
```

2. 将 `/router/user.js` 中的代码修改为如下结构：

```
const express = require('express')
const router = express.Router()

// 导入用户路由处理函数模块
const userHandler = require('../router_handler/user')

// 注册新用户
router.post('/reguser', userHandler.regUser)
// 登录
router.post('/login', userHandler.login)

module.exports = router
```

## 2. 登录注册

### 2.1 新建 ev\_users 表

1. 在 `my_db_01` 数据库中，新建 `ev_users` 表如下：

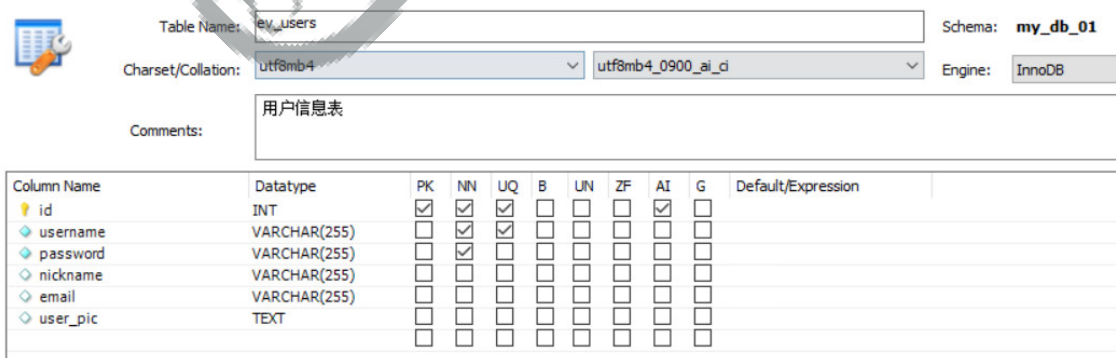


Table Name: `ev_users` Schema: `my_db_01`

Charset/Collation: `utf8mb4` `utf8mb4_0900_ai_ci` Engine: `InnoDB`

Comments: 用户信息表

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
username	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
password	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
nickname	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
email	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
user_pic	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

### 2.2 安装并配置 mysql 模块

在 API 接口项目中，需要安装并配置 `mysql` 这个第三方模块，来连接和操作 MySQL 数据库

1. 运行如下命令，安装 `mysql` 模块：

```
npm i mysql@2.18.1
```

2. 在项目根目录中新建 `/db/index.js` 文件，在此自定义模块中创建数据库的连接对象：

```
// 导入 mysql 模块
const mysql = require('mysql')

// 创建数据库连接对象
const db = mysql.createPool({
  host: '127.0.0.1',
  user: 'root',
  password: 'admin123',
  database: 'my_db_01',
})

// 向外共享 db 数据库连接对象
module.exports = db
```

## 2.3 注册

### 2.3.0 实现步骤

1. 检测表单数据是否合法
2. 检测用户名是否被占用
3. 对密码进行加密处理
4. 插入新用户

#### 2.3.1 检测表单数据是否合法

1. 判断用户名和密码是否为空

```
// 接收表单数据
const userinfo = req.body
// 判断数据是否合法
if (!userinfo.username || !userinfo.password) {
  return res.send({ status: 1, message: '用户名或密码不能为空!' })
}
```

#### 2.3.2 检测用户名是否被占用

1. 导入数据库操作模块:

```
const db = require('../db/index')
```

2. 定义 SQL 语句:

```
const sql = `select * from ev_users where username=?`
```

3. 执行 SQL 语句并根据结果判断用户名是否被占用:

```
db.query(sql, [userinfo.username], function (err, results) {
  // 执行 SQL 语句失败
  if (err) {
    return res.send({ status: 1, message: err.message })
  }
  // 用户名被占用
  if (results.length > 0) {
    return res.send({ status: 1, message: '用户名被占用, 请更换其他用户名!' })
  }
  // TODO: 用户名可用, 继续后续流程...
})
```

### 2.3.3 对密码进行加密处理

为了保证密码的安全性，不建议在数据库以 **明文** 的形式保存用户密码，推荐对密码进行 **加密存储**

在当前项目中，使用 **bcryptjs** 对用户密码进行加密，优点：

- 加密之后的密码，**无法被逆向破解**
- 同一明文密码多次加密，得到的**加密结果各不相同**，保证了安全性

1. 运行如下命令，安装指定版本的 **bcryptjs**：

```
npm i bcryptjs@2.4.3
```

2. 在 **/router\_handler/user.js** 中，导入 **bcryptjs**：

```
const bcrypt = require('bcryptjs')
```

3. 在注册用户的处理函数中，确认用户名可用之后，调用 **bcrypt.hashSync(明文密码, 随机盐的长度)** 方法，对用户的密码进行加密处理：

```
// 对用户的密码, 进行 bcrypt 加密, 返回值是加密之后的密码字符串
userinfo.password = bcrypt.hashSync(userinfo.password, 10)
```

### 2.3.4 插入新用户

1. 定义插入用户的 SQL 语句：

```
const sql = 'insert into ev_users set ?'
```

2. 调用 **db.query()** 执行 SQL 语句，插入新用户：

```
db.query(sql, { username: userinfo.username, password: userinfo.password }, function
(err, results) {
  // 执行 SQL 语句失败
  if (err) return res.send({ status: 1, message: err.message })
  // SQL 语句执行成功, 但影响行数不为 1
  if (results.affectedRows !== 1) {
    return res.send({ status: 1, message: '注册用户失败, 请稍后再试!' })
  }
  // 注册成功
  res.send({ status: 0, message: '注册成功!' })
})
```

## 2.4 优化 res.send() 代码

在处理函数中，需要多次调用 **res.send()** 向客户端响应 **处理失败** 的结果，为了简化代码，可以手动封装一个 **res.cc()** 函数

1. 在 **app.js** 中，所有路由之前，声明一个全局中间件，为 **res** 对象挂载一个 **res.cc()** 函数：

```
// 响应数据的中间件
app.use(function (req, res, next) {
  // status = 0 为成功; status = 1 为失败; 默认将 status 的值设置为 1, 方便处理失败的情况
  res.cc = function (err, status = 1) {
    res.send({
      // 状态
      status,
      // 状态描述, 判断 err 是 错误对象 还是 字符串
      message: err instanceof Error ? err.message : err,
    })
  }
  next()
})
```

## 2.5 优化表单数据验证

表单验证的原则：前端验证为辅，后端验证为主，后端**永远不要相信**前端提交过来的**任何内容**

在实际开发中，前后端都需要对表单的数据进行合法性的验证，而且，**后端做为数据合法性验证的最后一个关口**，在拦截非法数据方面，起到了至关重要的作用。

单纯的使用 `if...else...` 的形式对数据合法性进行验证，效率低下、出错率高、维护性差。因此，推荐使用**第三方数据验证模块**，来降低出错率、提高验证的效率与可维护性，**让后端程序员把更多的精力放在核心业务逻辑的处理上**。

1. 安装 `@escook/express-joi` 中间件，来实现自动对表单数据进行验证的功能：

```
npm i @escook/express-joi
```

2. 安装 `@hapi/joi` 包，为表单中携带的每个数据项，定义验证规则：

```
npm install @hapi/joi@17.1.0
```

3. 新建 `/schema/user.js` 用户信息验证规则模块，并初始化代码如下：

```
const joi = require('@hapi/joi')

/**
 * string() 值必须是字符串
 * alphanum() 值只能是包含 a-zA-Z0-9 的字符串
 * min(length) 最小长度
 * max(length) 最大长度
 * required() 值是必填项，不能为 undefined
 * pattern(正则表达式) 值必须符合正则表达式的规则
 */

// 用户名的验证规则
const username = joi.string().alphanum().min(1).max(10).required()
// 密码的验证规则
const password = joi.string().pattern(/^[\S]{6,12}$/).required()

// 注册和登录表单的验证规则对象
exports.reg_login_schema = {
  // 表示需要对 req.body 中的数据进行验证
  body: {
    username,
    password,
```

```
},  
}
```

4. 修改 `/router/user.js` 中的代码如下:

```
const express = require('express')  
const router = express.Router()  
  
// 导入用户路由处理函数模块  
const userHandler = require('../router_handler/user')  
  
// 1. 导入验证表单数据的中间件  
const expressJoi = require('@escook/express-joi')  
// 2. 导入需要的验证规则对象  
const { reg_login_schema } = require('../schema/user')  
  
// 注册新用户  
// 3. 在注册新用户的路由中, 声明局部中间件, 对当前请求中携带的数据进行验证  
// 3.1 数据验证通过后, 会把这次请求流转给后面的路由处理函数  
// 3.2 数据验证失败后, 终止后续代码的执行, 并抛出一个全局的 Error 错误, 进入全局错误级别中间件中进行处理  
router.post('/reguser', expressJoi(reg_login_schema), userHandler.regUser)  
// 登录  
router.post('/login', userHandler.login)  
  
module.exports = router
```

5. 在 `app.js` 的全局错误级别中间件中, 捕获验证失败的错误, 并把验证失败的结果响应给客户端:

```
const joi = require('@hapi/joi')  
  
// 错误中间件  
app.use(function (err, req, res, next) {  
  // 数据验证失败  
  if (err instanceof joi.ValidationError) return res.cc(err)  
  // 未知错误  
  res.cc(err)  
})
```