基于 JWT 的简单鉴权流程

对 jwt 不熟悉的推荐阅读:

- Server 端的认证神器——JWT(一)
- JSON Web Token 入门教程

node 实现 jwt 认证

技术实现方案: node + koa2 + mongodb

目录结构

开发前准备

- node
- mongodb

记得启动 node 服务之前,先本地启动 mongodb

user.js

```
const mongoose = require("mongoose");
const { Schema } = mongoose;

const userSchema = new Schema({
  name: String,
  password: String,
  salt: String,
  isAdmin: Boolean,
  age: Number
});

module.exports = mongoose.model("User", userSchema);
复制代码
```

config.js

```
module.exports = {
        'secret': 'ilovescotchyscotch', // 密钥
        'db': 'mongodb://localhost:27017/test'
}
```

复制代码

package.json

```
{
  "name": "token",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "dependencies": {
    "crypto-js": "^3.1.9-1",
    "jsonwebtoken": "^8.5.1",
    "koa": "^2.8.2",
    "koa-bodyparser": "^4.2.1",
    "koa-router": "^7.4.0",
    "mongoose": "^5.7.3"
  },
  "devDependencies": {
    "nodemon": "^1.19.3"
  },
  "scripts": {
    "start": "nodemon ./app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

复制代码

app.js

```
const Koa = require("koa");
const Router = require("koa-router");
const bodyParser = require("koa-bodyparser");
```

```
const md5 = require("crypto-js/md5");
const jwt = require("jsonwebtoken");
const mongoose = require("mongoose");
const User = require("./models/user.js");
const config = require("./config.js");

const app = new Koa();
const router = new Router();

mongoose.connect(config.db, { useUnifiedTopology: true });

app.use(bodyParser());

/**
 * @description 创建用户
 */
router.post("/user", async (ctx, next) => {
  const { username = "", password = "", age, isAdmin } =
ctx.request.body || {};
  if (username === "" || password === "") {
    ctx.status = 401;
    return (ctx.body = {
      success: false,
      code: 10000,
      msg: "用户名或者密码不能为空"
    });
  }
  // 先对密码 md5
  const md5PassWord = md5(String(password)).toString();
  // 生成随机 salt
  const salt = String(Math.random()).substring(2, 10);
  // 加盐再 md5
  const saltMD5PassWord = md5(`${md5PassWord}:${salt}`).toString();
  try {
    // 类似用户查找,保存的操作一般我们都会封装到一个实体里面,本 demo
只是演示为主, 生产环境不要这么写
    const searchUser = await User.findOne({ name: username });
    if (!searchUser) {
      const user = new User({
        name: username,
        password: saltMD5PassWord,
        salt,
        isAdmin,
        age
```

```
      });
      const result = await user.save();
      ctx.body = {
        success: true,
        msg: "创建成功"
      };
    } else {
      ctx.body = {
        success: false,
        msg: "已存在同名用户"
      };
    }
  } catch (error) {
    // 一般这样的我们在生成环境处理异常都是直接抛出 异常类, 再有全局错误处理去处理
    ctx.body = {
      success: false,
      msg: "serve is mistakes"
    };
  }
});

/**
 * @description 用户登陆
 */
router.post("/login", async (ctx, next) => {
  const { username = "", password = "" } = ctx.request.body || {};
  if (username === "" || password === "") {
    ctx.status = 401;
    return (ctx.body = {
      success: false,
      code: 10000,
      msg: "用户名或者密码不能为空"
    });
  }
  // 一般客户端对密码需要 md5 加密传输过来, 这里我就自己加密处理,假设客户端不加密。
  // 类似用户查找,保存的操作一般我们都会封装到一个实体里面,本 demo 只是演示为主, 生产环境不要这么写
  try {
    // username 在注册时候就不会允许重复
    const searchUser = await User.findOne({ name: username });
    if (!searchUser) {
      ctx.body = {
```

```
        success: false,
        msg: "用户不存在"
      };
    } else {
      // 需要去数据库验证用户密码
      const md5PassWord = md5(String(password)).toString();
      const saltMD5PassWord = md5(
        `${md5PassWord}:${searchUser.salt}`
      ).toString();
      if (saltMD5PassWord === searchUser.password) {
        // Payload: 负载, 不建议存储一些敏感信息
        const payload = {
          id: searchUser._id
        };
        const token = jwt.sign(payload, config.secret, {
          expiresIn: "2h"
        });
        ctx.body = {
          success: true,
          data: {
            token
          }
        };
      } else {
        ctx.body = {
          success: false,
          msg: "密码错误"
        };
      }
    }
  } catch (error) {
    ctx.body = {
      success: false,
      msg: "serve is mistakes"
    };
  }
});

/**
 * @description 获取用户信息
 */
router.get(
  "/user",
  async (ctx, next) => {
```

```
    // 这里应该抽成一个 auth 中间件
    const token = ctx.request.query.token ||
ctx.request.headers["token"];
    if (token) {
      jwt.verify(token, config.secret, async function(err, decoded) {
        if (err) {
          return (ctx.body = {
            success: false,
            msg: "Failed to authenticate token."
          });
        } else {
          ctx.decoded = decoded;
          await next();
        }
      });
    } else {
      ctx.status = 401;
      ctx.body = {
        success: false,
        msg: "need token"
      };
    }
  },
  async (ctx, next) => {
    try {
      const { id } = ctx.decoded;
      const { name, age, isAdmin } = await User.findOne({ _id: id });
      ctx.body = {
        success: true,
        data: { name, age, isAdmin }
      };
    } catch (error) {
      ctx.body = {
        success: false,
        msg: "server is mistakes"
      };
    }

  }
);

app.use(router.routes()).use(router.allowedMethods());
app.on("error", (err, ctx) => {
  console.error("server error", err, ctx);
```

```
});
app.listen(3000, () => {
  console.log("Server listening on port 3000");
});
```

复制代码

使用 postman 测试所有接口。

- 验证创建用户接口

- 去数据库验证

- 验证业务 api

基于 JWT 鉴权方案解决了哪些问题

- 服务端不再需要存储与用户鉴权相关的信息,鉴权信息会被加密到 token 中,服务器只需要读取 token 中包含的用户信息即可。
- 避免了共享 Session 不易扩展的问题
- 不依赖于 Cookie, 有效避免 Cookie 带来的 CORS 攻击问题
- 通过 CORS 有效解决跨域问题

关于 JWT 与 Token 的认识

通过这篇关于 jwt 与 token 讨论我纠正了自己的一些错误的观点,下一篇像记录关于 token 的学习。