认证和授权,身份验证（证明用户是其所声称的身份）和授权（确保经过身份验证的用户是否应该能够执行某些操作）的概念非常复杂，有几本好书（而且很厚）专门讨论此问题。在本章中，我们将从实践的角度和我们的FARM堆栈视角探讨身份验证和授权的主题。我们将从基于JWT的简单而强大且可扩展的FastAPI后端设置开始，JWT可以说是过去几年最流行的身份验证方法。然后，我们将看到如何将基于JWT的身份验证方法集成到React中，利用React的一些最酷和最新的功能，即Hooks、Context和React Router。本章将涵盖以下主题：
用户模型及其与其他资源的关系,JWT身份验证机制的简要概述 - 整体情况,FastAPI提供的身份验证和授权工具概述,如何保护我们的路由、路由器或整个应用程序,用React进行身份验证的各种解决方案,通过本章，您应该能够熟练掌握FastAPI和React在后端和前端提供的身份验证方法，并能够对用户进行身份验证，并在应用程序中精确控制他们可以做什么和不能做什么.

技术要求 为了在本章中运行示例应用程序，您的本地计算机上应安装Node.js和Python。要求与上一章中的要求相同，并且将指出我们将安装的新包。您应该具有较新的Python安装（版本3.6或更高
和Node.js安装（版本14或更高）。您的计算机应能够同时打开几个Visual Studio Code实例，以及几个终端和可能十几个浏览器选项卡。 理解JSON Web Token - 我们的安全钥匙,HTTP是一种无状态协议，仅此事实就意味着有几个重要的后果。其中之一是，如果我们想在请求之间保持某种状态，我们必须使用一种能够记住已登录用户、以前浏览器会话期间选择的汽车或网站首选项的机制。广义上说，我们在执行身份验证时可以采用许多策略。基于凭证的身份验证要求用户输入一些个人凭据，通常是用户名或电子邮件和密码。在过去几年中，一种新方法得到了一些关注，即无密码登录的概念 - 一旦用户创建了一个帐户，他们会收到一封包含魔术链接的电子邮件，该链接用于验证会话，无需输入（和记住！）密码。生物密码使用用户的某种生物特征，如指纹，而社交身份验证使用用户在社交网络（Google、Facebook或LinkedIn）上的帐户将用户与其帐户关联起来。在本章中，我们将考虑一种经典的个人凭证方法 - 当用户注册时，他们可以提供电子邮件并选择密码，以及可选的用户名。虽然有不同的方式可以在应用程序的不同部分维护用户的身份，但JSON Web Token（JWT）可以说是连接前端应用程序（React、Vue.js和Angular）或移动应用程序与API（在我们的情况下，是REST API）最常见和流行的方法之一。JWT只是一种标准 - 一种将看似随机的字符和数字组合成大字符串的方式。JWT包含三个部分 - 头部、有效载荷和签名。头部存储有关令牌本身的元数据：用于签署令牌的算法和令牌的类型。有效载荷是最有趣的部分。它包含数据（声明）：用户的ID（或用户名）和发行时间（iat）字段，即发行令牌的日期和时间，到期时间（令牌失效的时间），以及可选的其他字段。有效载荷可由所有人解码和读取。有一个非常有用的网站
jwt.io，我们可以在该网站上玩弄令牌并查看其外观。

FastAPI后端与用户和关系, 最后，令牌中可能最重要的部分是签名 - 令牌中保证声明的部分，可以这样说。签名被复制（计算）并与原始签名进行比较，从而防止对声明的修改。简单来说，如果一个JWT令牌可以轻松“读取”，声称用户名是John，我们可以篡改它并将用户名修改为Rita，但这样做会改变签名，导致令牌失效。如果您仔细思考，这实际上是一种简单而巧妙的机制。因此，令牌能够完全替代身份验证数据 - 用户或电子邮件和密码的组合，不需要在网络上传输多次。在本节中，我们了解了JWT是什么，其背后的逻辑是什么，以及为什么您可能希望在身份验证和授权系统中采用它。在接下来的章节中，我们将介绍如何在应用程序中实现基于JSON Web Token的身份验证流程。具有用户和关系的FastAPI后端,如果Web应用程序（或移动应用程序）不安全，则它们并不是非常有用 - 我们经常听到身份验证实现中的微小错误导致数十万甚至数百万个受损帐户，其中可能包括敏感和有价值的信息.FastAPI基于OpenAPI（以前称为Swagger） - 一种用于创建API的开放规范。因此，OpenAPI使我们能够定义与各种协议兼容的各种安全方案（apiKey、http、oauth2、openIdConnect等）。虽然FastAPI文档网站提供了一个出色而详细的教程来创建身份验证流程，但它是基于oauth2协议的，该协议使用表单数据来发送凭据（用户名和密码）。您可以有几十种方式来为API实现某种类型的安全性，但我在本章中真正想要做的只是给您提供可行选项的概念，并创建一个基于JWT和JSON作为传输机制的简单身份验证系统，这种工作流程易于扩展以适应您未来的需求，并且提供了足够的可移动部件，以便能够看到机制本身。在接下来的章节中，我们将设计一个简单的用户模型，使我们能够具有身份验证流程。然后，我们将学习如何将用户数据编码为JWT令牌，并学习如何要求令牌以访问受保护的路由。创建用户模型和MongoDB关系,为了能够讨论用户身份验证的概念，我们必须向我们的应用程序引入用户实体 - 到目前为止，我们只看到如何在单个实体（汽车）上执行CRUD操作。真实的应用程序可能至少有几个模型，而用户模型肯定是必需的。虽然您可以在用户模型中存储各种数据，但这实际上取决于您的需求；对于小型应用程序，几个字段就足够了 - 一个电子邮件和/或用户名，一个密码，可能是一些角色（常规用户、管理员或编辑者），等等。对于出版平台，您可能还想添加简短的个人简介、头像等等。

认证和授权 使用MongoDB模型数据与使用关系型数据库建模的过程有本质不同，如第2章“使用MongoDB设置文档存储”中所讨论的那样，重点在于从最开始考虑查询并建模关系，考虑到应用程序最常使用的查询。 首先，我们有哪些要求？我们的利益相关者非常满意前面的CRUD应用程序，并且最终他们希望将其转变为公共网站——汽车应该展示给潜在客户，而插入新汽车、更新或删除现有汽车的页面应该受到保护。目前我们设想的有两种类型的用户：销售员——可以插入新车辆，并编辑和删除“他们”自己的汽车（也就是他们负责的公司汽车），以及管理员——几名经理将监督整个过程并且应该能够执行所有操作，无论是什么实体。为了简化事务，我将使用一个简单的基于引用的模型；汽车将简单地具有一个附加字段（如外键），其中包含用户的ID，非常类似于关系型数据库模型。我们可以将所有用户的汽车列表嵌入到用户模型中，但在这个应用程序中，这已经足够了。 让我们从应用程序的模型开始。我们应该像为路由器一样应用相同的结构——创建一个/models目录和两个文件（users.py和cars.py）——但是为了使项目尽可能简单，我将把它们放在一个单独的models.py文件中。在你有超过两个模型的情况下应该避免这样做！
让我们从入口点main.py开始，它与上一章中使用的非常相似： from decouple import config from fastapi import FastAPI from fastapi.middleware.cors import CORSMiddleware from motor.motor_asyncio import AsyncIOMotorClient from routers.cars import router as cars_router from routers.users import router as users_router DB_URL = config('DB_URL', cast=str) DB_NAME = config('DB_NAME', cast=str)
我刚刚添加了一个新的路由器——我们将现在创建它的路由器： origins = ["*"] app = FastAPI() app.add_middleware(CORSMiddleware,

使用FastAPI构建具有用户和关系的后端 我们将创建一个基于FastAPI的后端，该后端将为用户和汽车之间建立联系。我们将实现以下功能: - 注册和登录用户 - 创建、查看、更新、删除汽车 - 将汽车指派给特定用户 - 查看每个用户已分配的汽车列表 - 查看每辆汽车所分配的用户
我们将遵循模块结构来组织我们的代码。 首先，我们需要在FastAPI应用程序中设置MongoDB客户端，以便在对数据库进行任何操作时使用它。我们定义以下变量： - DB_URL 用于指定MongoDB实例的URL - DB_NAME 用于指定要使用的数据库名称
接下来，在main.py文件中，我们设置FastAPI应用程序，并为CORS提供支持。
我们挂载了“ /cars”路由器，并添加了“ /users”路由器，以在FastAPI应用程序中处理用户路由。 在app.on_event(“startup”)中，我们连接了MongoDB客户端，并在app.on_event(“shutdown”)中断开连接。
现在，让我们创建模型，模型将用于管理我们的数据。
我们声明了PyObjectId，以将ObjectId展平为字符串类型。
我们还使用Pydantic库定义了以下模型： - User - Car - Relation
User和Car模型能够自动使用这些功能： - 创建、更新、删除 - 获取所有 - 获取特定的模型 Relation模型用于为车辆和用户之间建立关系。
我们需要使用以下方法： - create_relation - get_relation - delete_relation 在cars文件夹中，我们将实现与汽车有关的处理程序，在users文件夹中，我们将实现用户和关系的处理程序。 最后，我们将实现一些代表我们处理程序逻辑的静态方法。重要的是，不使用从数据库获取的汽车和用户，我们只使用模型和它们的id，在代表我们的逻辑时，我们将使用这些id。

身份验证和授权 164 @classmethod def validate(cls, v): if not ObjectId.is_valid(v): raise ValueError("Invalid objectid") return ObjectId(v) @classmethod def __modify_schema__(cls, field_schema): field_schema.update(type="string") class MongoBaseModel(BaseModel): id: PyObjectId = Field(default_factory=PyObjectId, alias="_id") class Config: json_encoders = {ObjectId: str}
请注意，我们导入了电子邮件验证程序包，因为它不是Pydantic的一部分 - 您应该使用以下命令安装它： pip install email-validator 这是一个必需的Pydantic软件包，用于验证电子邮件地址，因为当用户注册时我们想要要求一个有效的电子邮件地址。虽然我不会在本章实现客户端注册流程，但我会创建用户创建路由，并且它需要一个有效的电子邮件地址。谁知道，也许公司的所有者以后会决定引入客户账户？
继续使用同一个文件models.py和实际用户模型。我们定义了两个角色-销售员和管理员-以及一个非常基本的用户模型，仅包含用户名、电子邮件、密码和角色字段。由于电子邮件字段是唯一一个不能直接从Pydantic进行验证的字段，我们通过使用电子邮件验证程序包添加了一个简单的验证方法。这真的很简单 - 如果提供的值不是有效的电子邮件，则会返回错误： class Role(str, Enum): SALESPERSON = "SALESPERSON" ADMIN = "ADMIN" class UserBase(MongoBaseModel): username: str = Field(..., min_length=3, max_length=15) email: str = Field(...)

使用FastAPI后端进行用户和关系管理 在这个例子中，我们将使用FastAPI和MongoDB来建立一个小型的后端应用程序，来管理一个具有许多用户和用户之间的关系的数据库。由于我们需要使用Pydantic来处理输入和输出数据的验证，所以我们将会尝试尽可能多地使用这个库。 首先，我们需要定义一些助手模型来帮助我们在用户之间建立关系。这些模型将描述我们的应用程序中有哪些角色，以及每个用户应该拥有哪个角色。这是我们的代码： class Role(str, Enum): salesperson = "salesperson" admin = "admin" class RelationshipType(str, Enum): colleague = "colleague" manager = "manager" subordinate = "subordinate" class RelationshipBase(BaseModel): from_user_id: ObjectIdStr = Field(...) to_user_id: ObjectIdStr = Field(...) relationship_type: RelationshipType = Field(...) 在这里，我们定义了三个助手模型：Role、RelationshipType和RelationshipBase。Role将是我们的用户角色，每个用户都必须有一个角色。我们还定义了一个关系类型的助手模型，以便我们能够为不同的关系命名。最后，我们定义了RelationshipBase，它将是所有关系的基本模型，我们需要添加一些从哪个用户到哪个用户的属性，以及它们之间的关系类型。
在这之后，我们需要定义一个表示用户的模型。它将包含有关每个用户的基本信息：class UserBase(BaseModel): username: str = Field(..., min_length=3, max_length=15) email: str = EmailStr(...) password: str = Field(...) role: Role @validator("email") def valid_email(cls, v): try: email = validate_email(v).email return email except EmailNotValidError as e: raise EmailNotValidError 在这里，我们扩展了Pydantic的BaseModel，定义了用户的基本信息。其中包括用户名、电子邮件地址、密码和角色。我们还使用了Pydantic的@validator来确保电子邮件地址的有效性。 接下来是一个用于登录的辅助模型： class LoginBase(BaseModel): email: str = EmailStr(...) password: str = Field(...) 和一个表示当前用户的模型： class CurrentUser(BaseModel): email: str = EmailStr(...) username: str = Field(...) role: str = Field(...) 对于CurrentUser，我们将在我们的程序中使用它来表示已登录的用户的信息，以便我们能够验证他们是否具有执行许多操作所需的必要访问权限。 我们已经定义了一些模型和一些助手模型，接下来定义用户模型。这样我们便可以通过这个模型来管理我们的用户列表：class User(UserBase): id: ObjectIdStr = Field(...) password_hash: str = Field(...) relationships: Set[RelationshipBase] = set() 在这里，我们扩展了我们之前定义的UserBase类，并添加了一个id字段、一个password_hash字段，以及一个relationships属性。我们可以使用这个模型来获取和管理所有用户的信息（例如，我们可能需要从数据库中获取当前所有用户的列表，以便在我们的应用程序中显示它们）。最后，我们需要确保我们可以为每个用户创建一些关系。像这样： class UserCreate(UserBase): password: str class UserInDB(MongoBaseModel): username: str = Field(..., min_length=3, max_length=15) email: str = EmailStr(...) password_hash: str = Field(...) role: Role relationships: Set[RelationshipBase] = set() 当前，我们最终定义了一个名为UserCreate的类，它将包含有关我们要向数据库中添加的用户的所有信息。与此同时，我们还定义了UserInDB，一个扩展了UserCreate类的助手模型，它将包含有关已存在于数据库中的用户的所有信息。 在这些使用Pydantic的助手模型之后，我们现在可以使用FastAPI来管理我们的后端逻辑和路由了。希望这个例子能够帮助你开始构建自己的FastAPI应用程序！

身份验证和授权 166 UserModel相当简单：我们要求用户名长度在3到15个字符之间、有效的电子邮件地址、密码和角色。我添加了两个附加模型：LoginBase用于登录路由，CurrentUser包含我们将从模型中提取的数据，以检查当前是谁在发出请求。接下来，我决定将Cars模型放在同一个models.py文件中，几乎没有什么变化： class CarBase（MongoBaseModel）： 品牌：str = Field（...，min_length = 3） 制作：str = Field（...，min_length = 1） 年份：int =字段（...，gt = 1975，lt = 2023） 价格：int = Field（...） 千米：int =字段（...） cm3：int =字段（...，gt = 600，lt = 8000） class CarDB（CarBase）： 所有者：str = Field（...） class CarUpdate（MongoBaseModel）： 价格：Optional [int] =无 基础模型完整无缺，具有我们先前拥有的所有功能（里程、生产年份等等）。我只是添加了一个名为CarDB的新模型，它扩展了CarBase模型并添加了一个所有者字段，即分配给车辆的用户的ID，由于我们将所有的MongoDB ObjectIds转换为字符串，它也是一个字符串。CarUpdate模型仅包含可选的价格更新。重要的是要指出，这个模型是真实系统的很大简化。我们可能希望在User模型中添加一个汽车ID列表作为字段，我们会有一堆DateTime字段来表示车辆上架、售出、预留等时刻。然而，在本章中，我只想实现一个相当简单的基于JWT的身份验证系统，并保留最低限度的功能，以便具有可用的机制。 进入身份验证文件，恰当地称为authentication.py。让我们快速回顾一下我们身份验证机制的要求： 用户提交注册表单并向我们发送密码后，密码应进行哈希处理，然后才插入到数据库中 我们应该准备好一个函数，用于比较存储的密码哈希与登录阶段提交的后续密码，以验证它们是否匹配 我们应该能够创建/编码JWT令牌，并使用自定义过期时间和包含用户ID的有效负载对其进行解码

FastAPI后端与用户关系的身份验证 最后，我们应该有一个函数通过依赖注入接受请求并返回制作请求的用户的ID或诸如无效令牌或令牌过期之类的消息 以下机制是从YouTube视频（https://www.youtube.com/watch?v=xZnOoO3ImSY）中受到启发和调整的，该视频提供了一个比FastAPI文档中提出的更简单和简单的方法。 让我们开始构建我们的authentication.py文件。首先，我们需要安装一些用于JWT身份验证机制的库。 Pyjwt是用于编码和解码JWT的库，而passlib是用于散列字符串的库。停止您的FastAPI服务器，然后在活动环境中插入以下命令： pip install pyjwt passlib['bcrypt'] 现在我们已经准备好在authentication.py文件中声明我们的导入了: import jwt from fastapi import HTTPException, Security from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer from passlib.context import CryptContext from datetime import datetime, timedelta 正如我们之前所说，jwt在这里使我们能够编码和解码JWT，而FastAPI为我们提供了大部分所需的功能。 HTTPException将处理令牌无效的情况-有效地将代码异常转换为有效的HTTP响应-而Security用于授权并突出显示需要用户在自动文档中进行身份验证的路线。 HTTPBearer是FastAPI类，确保HTTP请求具有适当的身份验证头，而HTTPAuthorizationCredentials是从依赖注入返回的对象类型。 CryptContext用于创建散列密码的上下文，它位于passlib.context下。最后，我们导入了一些日期时间实用程序，用于签署令牌并为其赋予期望的到期日期。 在声明了我们的导入之后，现在是创建一个我将称之为Authorization的类的时候了，它将公开负责所有所需身份验证步骤的方法: class Authorization（）： 安全= HTTPBearer（） pwd_context = CryptContext（schemes=["bcrypt"], deprecated ="auto"） 密钥='FARMSTACKsecretString'

身份验证和授权 我们正在实例化FastAPI的最简单身份验证——HTTPBearer，并使用bcrypt算法创建一个密码上下文。我们还需要一个可以自动生成以增加安全性的秘密字符串。接下来，我们将处理密码的哈希： def get_password_hash(self, password): return self.pwd_context.hash(password) def verify_password(self, plain_password, hashed_password): return self.pwd_context.verify(plain_password, hashed_password) 这些相当简单的函数确保用户的密码被哈希处理，并且可以通过与明文版本进行比较来进行验证。第二个函数返回一个简单的true或false值。现在我们到了类的核心——创建JWT： def encode_token(self, user_id): payload = { 'exp': datetime.utcnow() + timedelta(days=0, minutes=35), 'iat': datetime.utcnow(), 'sub': user_id } return jwt.encode(payload, self.secret, algorithm='HS256') 上述函数完成了大部分工作——它将user_id作为唯一参数，并将其放入payload的sub部分。请记住，我们可以在JWT中编码更多信息——例如用户的角色或用户名。在这种情况下，sub部分将具有字典结构，并且JWT将会更长。过期时间设置为35分钟，而发布时间设置为JWT创建的时刻。最后，该函数使用jwt.encode方法来对令牌进行编码。我们提供算法（HS256）和一个密钥作为参数。
该类的解码部分非常类似；我们只需颠倒该过程并在必要时提供异常： def decode_token(self, token): try: payload = jwt.decode(token, self.secret, algorithm='HS256') return payload['sub'] except jwt.ExpiredSignatureError: raise ApiException(status_code=401, detail='Token已过期') except jwt.InvalidTokenError: raise ApiException(status_code=401, detail='无效的Token') 这里，我们使用jwt.decode方法完成解码。该方法需要处理异常，例如令牌过期或无效令牌。该函数返回子部分中的用户ID，该值用于身份验证和授权。

FastAPI后端用户及关联关系 接下来，在Authorization.py中创建身份验证文件。以下是我们需要的一些导入和类实例化： import jwt from fastapi import HTTPException, Security from fastapi.security import HTTPAuthorizationCredentials, SecurityScopes AUTHORIZATION_SCOPE = 'users:read' class AuthHandler: def __init__(self): self.secret_key = 'your_secret_key' def encode_token(self, user_id: int) -> str: payload = { 'sub': user_id } return jwt.encode(payload, self.secret_key, algorithm='HS256') def decode_token(self, token: str) -> int: try: payload = jwt.decode(token, self.secret_key, algorithms=['HS256']) return payload['sub'] except jwt.ExpiredSignatureError: raise HTTPException(status_code=401, detail='Signature has expired') except jwt.InvalidTokenError as e: raise HTTPException(status_code=401, detail='Invalid token') def auth_wrapper(self, auth: HTTPAuthorizationCredentials = Security(security)): return self.decode_token(auth.credentials) authorization.py文件不到40行，但有很大作用-它通过利用FastAPI优秀的依赖注入机制使我们可以保护路由。 接下来，让我们进入用户路由器并测试我们的身份验证逻辑。在路由器文件夹中，创建一个名为users.py的文件，并开始导入及实例化： from fastapi import APIRouter, Request, Body, status, HTTPException, Depends from fastapi.encoders import jsonable_encoder from fastapi.responses import JSONResponse from models import UserBase, LoginBase, CurrentUser from authentication import AuthHandler router = APIRouter() auth_handler = AuthHandler()
在标准FastAPI导入之后，包括jsonable_encoder和JSONResponse，我们从user models和authorization.py中导入AuthHandler类。然后，我们继续创建该路由器将负责所有用户路由的路由器并实例化AuthHandler。让我们从注册路由开始，以便我们可以创建一些用户并使用REST客户端进行测试： @router.post("/register", response_description="Register user") async def register(request: Request, newUser: UserBase = 请翻译此文，不要增加或删除任何其它内容

身份验证和授权 170 Body(...)) -> UserBase： newUser.password = auth_handler.get_password_hash(newUser. password) newUser = jsonable_encoder(newUser) 注册路由将在/用户/注册URL处可用，通过请求和由Pydantic的UserBase类建模的newUser实例通过请求的正文。我们首先将密码替换为散列密码，并将Pydantic模型转换为可JSON编码的编码器实例。 现在，我们执行标准的注册检查-电子邮件和用户名应该可用;否则，我们抛出异常，通知用户用户名或密码已被使用： if (existing_email := await request.app.mongodb["users"].find_one({"email": newUser["email"]}) is not None): raise HTTPException(status_code=409, detail=f"User with email {newUser['email']} already exists") if (existing_username := await request.app.mongodb["users"].find_one({"username": newUser["username"]}) is not None): raise HTTPException(status_code=409, detail=f"User with username {newUser['username']} already exists",) 前面的函数可以和应该重构，以允许进一步的检查，但我希望它们尽可能明确。函数的最后一部分很简单；我们只需要将用户插入MongoDB！您可以在以下代码中看到： user = await request.app.mongodb["users"].insert_ one(newUser) created_user = await request.app.mongodb["users"].find_one({"_id": user.inserted_id})

使用FastAPI构建具有用户和关系的后端 我们已经实现了一个基本的用户注册路由，让我们继续添加一个登录路由。我们将使用JWT(JSON Web Tokens)为我们的用户生成身份验证令牌，这样我们就可以在后续的请求中使用该令牌来验证用户身份。 首先，我们需要安装一个PyJWT来处理JWT认证。在我们的项目虚拟环境中，使用以下命令：
(venv) $ pip install pyjwt 接下来，我们需要为我们的应用定义一个令牌处理器（TokenHandler）。这个处理器将负责编码和解码JWT令牌。我们将定义一个名为auth.py的新模块，并在其中实现所需的方法。在这个模块中，我们将定义以下步骤： 1. 生成一个新的JWT令牌。 2. 解码并验证一个给定的JWT令牌。 3. 从JWT令牌中获取用户ID。 4. 从数据库中查找用户对象使用用户ID。
为了生成JWT令牌，我们将采用以下步骤： 1. 创建一个有效载荷（payload），用于存储用户ID和过期时间。 2. 使用PyJWT库中的encode方法将该有效载荷编码为JWT令牌。 3. 返回JWT令牌。
代码如下： import jwt from datetime import datetime, timedelta JWT_SECRET = "myjwtsecret" JWT_ALGORITHM = "HS256" JWT_EXP_DELTA_SECONDS = 3600 def encode_token(user_id: str) -> str: payload = { "exp": datetime.utcnow() + timedelta(seconds=JWT_EXP_DELTA_SECONDS), "sub": user_id, } return jwt.encode(payload, JWT_SECRET, JWT_ALGORITHM)
要解码和验证JWT令牌，我们将定义以下步骤： 1. 使用PyJWT库中的decode方法将JWT令牌解码为有效载荷。 2. 验证有效载荷是否有效，即是否在过期时间内。 3. 如果验证成功，返回有效载荷中的用户ID。 代码如下： def decode_token(token: str) -> str: try: payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM]) user_id = payload["sub"] return user_id except jwt.ExpiredSignatureError: raise HTTPException(status_code=401, detail="Token has expired") except jwt.InvalidTokenError: raise HTTPException(status_code=401, detail="Invalid token") 从JWT令牌中获取用户ID非常简单，只需要解码JWT令牌并返回有效载荷中的用户ID即可： def get_user_id_from_token(request: Request) -> str: token = request.headers.get("authorization", "").split("Bearer ")[1] user_id = decode_token(token) return user_id 我们现在已经具备了处理JWT令牌所需的所有工具，让我们继续开发路由。首先，让我们回到我们的users路由文件并定义一个新的路由： @router.post("/login", response_description="Login user") async def login(request: Request, loginUser: LoginBase = Body(...)) -> str: user = await request.app.mongodb["users"].find_one({"email": loginUser.email}) if (user is None) or (not auth_handler.verify_password(loginUser.password, user["password"])): raise HTTPException(status_code=401, detail="Invalid email and/or password") token = auth_handler.encode_token(user["_id"]) response = JSONResponse(content={"token": token}) return response 我们接收电子邮件和密码（我们可以选择用户名），首先，我们尝试按电子邮件查找用户。之后，我们比较密码和我们的哈希函数。如果发生错误，我们将引发HTTPException并返回401 Unauthorized响应。如果我们成功验证了电子邮件和密码，我们将生成新的JWT令牌并返回带有令牌的JSON响应。 我们返回标准的201 CREATED状态码，现在我们准备使用HTTPie进行一些基本的测试，HTTPie是我们的命令行REST客户端。让我们尝试创建一个用户，如下所示： (venv) λ http POST 127.0.0.1:8000/users/register username="bill" password="bill" role="ADMIN" email="koko@gmail.com" 我们得到一个新的用户，其中包括一个哈希密码、一个角色和_id。当然，我们不希望将密码发送回给用户，即使它是哈希的，但是您已经掌握了创建一个新的Pydantic模型，该模型返回除密码之外的所有字段的知识。让我们继续进行登录路由 - 它与您可能已经使用Flask或Express.js相似。我们接收电子邮件和密码（我们可以选择用户名），首先，我们尝试按电子邮件查找用户。之后，我们比较密码和我们的哈希函数：
@router.post("/login", response_description="Login user") async def login(request: Request, loginUser: LoginBase = Body(...)) -> str: user = await request.app.mongodb["users"].find_one({"email": loginUser.email}) if (user is None) or (not auth_handler.verify_password(loginUser.password, user["password"])): raise HTTPException(status_code=401, detail="Invalid email and/or password") token = auth_handler.encode_token(user["_id"]) response = JSONResponse(content={"token": token}) return response

身份验证和授权 172 如果用户存在并且密码通过哈希验证，我们将创建一个令牌并将其作为JSON响应返回。这个宝贵的令牌将负责所有应用程序的身份验证，它将是每个请求向服务器发送的唯一数据。我们也可以通过使用相应的凭据访问/ users / login路径来测试登录路由： λ http POST http://127.0.0.1:8000/users/login email="tanja@ gmail.com" password="tanja" HTTP / 1.1 200 OK content-length: 184 content-type: application / json date: Wed, 01 Jun 2022 20:13:32 GMT {server：uvicorn “token”：“eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9。eyJleHAiOjE2NTQxMTY1MTMsImlhdCI6MTY1NDExNDQxMywic3ViIjoiNjI4OTQyODU3YTBjYmZlNGE2MzQwNDdkIn0.v1FTBMOwIlDKUw_VVCJlsSItM58sDzDnwGbzyDKs_pc” } 我们得到了令牌！如果您愿意，您可以尝试使用错误的用户名/密码组合调用相同的路由，并检查响应。
我们将需要用户路由中的一个最终路由：/ me路由。此路由不应直接调用并生成页面，只应作为辅助功能使用-验证当前登录用户。/ me路由不应接受除身份验证依赖项之外的任何参数-测试我们的身份验证包装器的绝佳机会： @ router.get（“/ me”，response_description=“已登录用户数据”） async def me（request：Request，userId=Depends（auth_handler. auth_wrapper））：currentUser = await request.app.mongodb [“users”] .find_one（{“_id”：userId}） result = CurrentUser（** currentUser）.dict（）result [“id”] = userId return JSONResponse（status_code = status.HTTP_200_OK，content = result） 这条路线非常简单：如果提供的令牌有效且未过期，auth_wrapper将返回userId-发出请求的用户的ID。否则，它将返回HTTP异常。在这条路线中，我添加了一个数据库调用，以便根据CurrentUser模型检索有关用户的所需数据。

使用FastAPI后端进行用户和关系管理 我们本可以将所有这些数据编码到令牌中，避免访问数据库，但我希望将JWT尽可能保持轻量级。
现在，我们可以测试/me路由。首先，让我们使用先前注册的用户进行登录： (venv) λ http POST 127.0.0.1:8000/users/login password="bill" email="koko@gmail.com" {HTTP/1.1 200 OK "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE2NTM4Mzk1NTksImlhdCI6MTY1MzgzNzQ1OSwic3ViIjoiNjI5MzMzZDdlMzM4NDJkOTQ5OWU2YWM3In0. ajpoftEFBWcfn2XCIJqPDNcJMaS6OujZpaU8bCvOBNE" }
复制此令牌并将其提供给/me路由： (venv) λ http GET 127.0.0.1:8000/users/me "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE2NTM4Mzk1NTksImlhdCI6MTY1MzgzNzQ1OSwic3ViIjoiNjI5MzMzZDdlMzM4NDJkOTQ5OWU2YWM3In0. ajpoftEFBWcfn2XCIJqPDNcJMaS6OujZpaU8bCvOBNE" {HTTP/1.1 200 OK "email": "koko@gmail.com", "id": "629333d7e33842d9499e6ac7", "role": "ADMIN", "username": "bill" } 如果尝试在没有Bearer令牌的情况下测试路由，则会收到“未经身份验证”错误，并回到原点。 最后，我将展示如何将身份验证依赖项插入/cars路由器(或者实际上，您可能需要创建的任何其他路由器)。由于文件会相当长，我不会解释所有内容——我将着重介绍用于在一些路由器上执行身份验证和授权的逻辑，而整个文件可在本书的GitHub存储库中找到。让我们看一下/cars路由器的导入：
```
from typing import List, Optional from fastapi import APIRouter, Request, Body, status,
```

身份验证和授权 174 HTTPException，取决于 from fastapi.encoders import jsonable_encoder from fastapi.responses import JSONResponse from models import CarBase，CarDB，CarUpdate from authentication import AuthHandler router = APIRouter（） auth_handler = AuthHandler（）
这部分与用户路由几乎相同 -
我们导入了我们的Pydantic模型并实例化路由和身份验证处理程序类。 我们的Pydantic汽车模型已更改;现在它有一个被称为所有者的东西，基本上只是将要出售汽车的用户的ID。我们将通过我们的身份验证依赖项向数据库提供这些信息。当用户尝试插入实体的新实例（汽车）时，他们必须经过身份验证才能继续。如果他们经过了身份验证，我们将只取他们的ID并将其设置为所有者字段的值： @ router.post（“/”，response_description =“Add new car”） async def create_car（ request：Request， car：CarBase = Body（...），
userId=Depends（auth_handler.auth_wrapper），）： car = jsonable_encoder（car） car [“owner”] = userId new_car = await request.app.mongodb [“cars2”]。insert_one（car） created_car = await request.app.mongodb [“cars2”]。find_one（
{“_id”：new_car.inserted_id} ） return JSONResponse（status_code = status.HTTP_201_CREATED， content = created_car） 最简单的情况是与GET / cars相对应的路径 - 这条路线将列出所有可用的汽车，并通过查询字符串实现一些分页。假设我们只想让已登录的用户（因此是销售人员或管理员）访问此路线。

FastAPI后端带有用户和关系 我们只需要将身份验证包装器注入到依赖项中，FastAPI只有两个选择：令牌有效且新鲜，我们获得用户；或者抛出HTTPException异常。就是这么简单。因此，让我们创建列出汽车的路由 -
我们假设只有注册用户才能访问这条路线： @router.get("/", response_description="List all cars") async def list_all_cars(request: Request, min_price: int = 0, max_price: int = 100000, brand: Optional[str] = None, page: int = 1, userId=Depends(auth_handler.auth_wrapper),) -> List[CarDB]: RESULTS_PER_PAGE = 25 skip = (page - 1) * RESULTS_PER_PAGE query = {"price": {"$lt": max_price, "$gt": min_price}} if brand: query["brand"] = brand full_query = (request.app.mongodb["cars2"] .find(query) .sort("_id", -1) .skip(skip) .limit(RESULTS_PER_PAGE)) results = [CarDB(**raw_car) async for raw_car in full_query] return results 虽然整个函数执行了一些分页（每页25个硬编码结果）并具有一些智能选项，可用于按价格和品牌过滤，但认证逻辑的要点在粗体字行中。另外请注意，我创建了一个单独的MongoDB集合并将其命名为cars2，仅用于区分上一章中使用的集合，同时使用相同的数据库。

身份验证和授权 最后，让我们来看看编辑汽车的路由（仅限价格）。我们希望只有汽车所有者能够编辑价格，此外，任何管理员也可以介入并更新价格。对于这种情况，如果我们在JWT中编码了用户角色，那么这将是明智的选择，因为它可以避免我们向数据库发送请求。但是，我只是想让您意识到在开发API过程中，您必须做一些决策和权衡： @router.patch（“/{id}”，response_description =“Update car”） async def update_task（ id：str， request：Request， car：CarUpdate = Body（...）， userId = Depends（auth_handler.auth_wrapper）， ）： user = await request.app.mongodb [“users”] .find_one（{“_id”：userId}） findCar = await request.app.mongodb [“cars2”] .find_one（{“_id”：id}） if（findCar [“owner”]！= userId）并且user [“role”]！= “ADMIN”： raise HTTPException （ status_code = 401，detail =“只有所有者或管理员才能更新汽车” ）await request.app.mongodb [“cars2”] .update_one（ {“_id”：id}，{“$ set”：car.dict（exclude_unset = True）} ） 如果（car：= await request.app.mongodb [“cars2”] .find_one（{“_id”：id}））不为None： return CarDB（** car） 引发HTTPException （status_code = 404，detail = f“未找到带有 {id} 的汽车”） 在这个路由处理程序中，我们首先获取发出请求的用户，然后定位要编辑的汽车。最后，我们进行检查：如果汽车的所有者不是发出请求的用户，而且这个用户不是管理员，则抛出异常。否则，我们执行更新。FastAPI的依赖注入是一个简单而强大的机制，在认证和授权领域真的很出色！

在React中认证用户 在本节中，我们在FastAPI后端创建了一个简单而有效的身份验证系统，创建了一个JWT生成器，并能够验证令牌，保护了某些路由，并提供了所需的路由以创建（注册）新用户和登录。现在是时候看看前端的工作原理了！ 与安全的其他方面一样，React中的身份验证是一个巨大的主题，超出了本书的范围。在本节中，我只会给您一个非常基本的机制，使我们能够在客户端拥有简单的身份验证流程。一切都围绕JWT和我们决定如何处理它的方式进行。在本章中，我们将仅将其存储在内存中。
互联网和专业文献中充满了关于存储身份验证数据的最佳解决方案的争议 - 在我们的情况下，JWT令牌。与往常一样，每种解决方案都有其优点和缺点。
Cookie存在已经有很长时间了 - 它们可以在浏览器中以键值对的方式存储数据，并且从浏览器和服务器都可读。它们的流行与经典的服务器端渲染的网站相吻合。但是，它们可以存储非常有限的数据，而且所述数据的结构必须非常简单。
Localstorage和Session Storage是与HTML5一起引入的，以解决单页应用程序中存储复杂数据结构等问题。它们的容量约为10MB，取决于浏览器的实现，而Cookie的容量为4KB。会话存储数据在会话中持久存在，而本地存储在浏览器中保留，即使在关闭和重新打开浏览器之后也会保留，直到手动删除。两者都可以托管复杂的JSON数据结构。 在localstorage中存储JWT很好，易于使用，并且允许出色的用户体验和开发人员体验。然而，它被认为是不可取的，因为它会打开应用程序许多漏洞，因为任何在浏览器中运行的客户端JavaScript都可以访问它们。 大多数权威机构建议在HTTP中存储JWT-仅使用无法通过JavaScript访问的Cookie，并要求前端和后端在同一域上运行。这可以通过不同的方式实现，通过路由请求，使用代理等。另一种流行的策略是使用所谓的刷新令牌-我们在登录时发出一个令牌，然后该令牌会自动用于生成其他（刷新）令牌，使我们能够在安全性和用户体验之间进行缓解。 在本节中，我将构建一个非常简单且极简的React应用程序，该应用程序仅仅能够满足要求;一些路线和页面应该受到保护，除非用户登录。我不会以任何方式持久化JWT-当用户刷新应用程序时，他们将退出登录。不是最愉快的用户体验，但现在不是问题。

身份验证和授权 178 让我们一步一步地进行。我们的FastAPI后端正在运行，我们准备创建我们的简单前端： 1.进入/chapter7目录，并从终端创建一个React应用程序： npx create-react-app frontend 2.更改目录到frontend并安装Tailwind CSS： npm install -D tailwindcss postcss@latest autoprefixer
3.使用以下命令初始化Tailwind： npx tailwindcss init -p
4.现在，是时候编辑postcss.config.js文件： module.exports = { content: ["./src/**/*.{js,jsx,ts,tsx}",], theme: { extend: {}, }, plugins: [], }
5.最后，删除src/index.css文件中的所有内容，并将内容替换为以下内容： @tailwind base; @tailwind components; @tailwind utilities; 这些步骤应该已经很熟悉了，但现在我想将过程推进一步。Tailwind在过去几年中变得越来越受欢迎，不同的UI工具包和实用程序基于基本的Tailwind类。其中最受欢迎和可用的之一称为DaisyUI（https://daisyui.com），我们将用它来原型化我们的应用程序。
6.安装过程与Tailwind本身类似。在终端中输入以下内容： npm i daisyui

在React中认证用户 179 7. 完成后，我们需要将daisyui注册为tailwind.config.js中的Tailwind插件，如下所示： module.exports = { //... plugins: [require("daisyui")], } 8. 最后，删除所有不需要的文件（例如App.css和Logo.svg），并将您的App.js文件缩减为以下内容，以测试React是否已获取UI依赖项： function App() { return (It works! }); export default App; 9. 现在，我们可以测试应用程序，并查看Tailwind和DaisyUI是否正在正确运行-您应该能够看到一个带有样式按钮的漂亮空页面。我必须再次运行以下命令： npm install postcss@latest 也许到您阅读本文时，修复已不再必要。 10.为了进行认证，我们将深入了解React Router 6，并利用其一些新功能和组件。停止终端进程并勇敢地安装路由器： npm install react-router-dom@6 11.我们将在index.js文件中设置路由器，如下所示： import React from 'react'; import ReactDOM from 'react-dom/client'; import './index.css'; import {BrowserRouter, Routes, Route} from 'react-router-dom' import App from './App'; const root = ReactDOM.createRoot(document.

认证和授权 我们正在将所有内容都包裹在路由器中，以便它“覆盖”整个应用程序，路径是一个通配符（*），而需要提供给路由器的元素是根 App.js 组件。现在来到定义所有可能路由和组件的繁琐部分，但再次，我们将使用 React Router 的新功能——嵌套路由。我们将使用路由器的 Outlet 组件，而不是将每个组件都包装到包含公共网页元素（如导航栏或页脚）的 Layout 组件中。Outlet 组件只是使用 URL 模式匹配的嵌套组件内容填充组件。 12. 在 /src 下创建一个 components 文件夹，并构建通用的 Header.jsx 和 Footer.jsx 组件，利用我们的 React ES6 Visual Studio Code 扩展程序（通过输入 _rafce 来完成）： const Header = () => { return Header; }; export default Header; 13. 按照完全相同的过程，在 /src/components 文件夹中创建以下组件：Footer、HomePage、Login 和 Register，这些组件仅包含返回组件名称的 div。Layout.jsx 将利用嵌套路由： import { Outlet } from "react-router-dom"; import Header from "./Header"; import Footer from "./Footer"; const Layout = () => { return ( ); }; export default Layout;

在React中对用户进行身份验证 Layout组件很简单，但非常有用：它利用Outlet组件作为高阶组件，有效地包装了包含的路由并添加了头部和脚注。我使用了Tailwind的类使页面成为全高度，并将显示设置为flex。主要部分设置为flex-1，以占用所有剩余空间。 App.js文件现在更新如下： import {Route，Routes} from“react-router-dom” import Layout from "./components/Layout"; import Login from "./components/Login"; import Register from "./components/Register"; import HomePage from "./components/HomePage"; function App() { return (}> } /> } /> } /> } /> }); export default App;

身份验证和授权 182 敏锐的观察者立刻会注意到使用 Layout 组件作为渲染元素的 Route 元素不是自我关闭的-实际上，它包含了所有剩余的路由，引导了它们，并在过程中添加了 Header 和 Footer
组件。非常出色和优雅！您可以手动尝试更改URL；导航到 /login 或 /register 或简单地 /（React
站点的根URL），并查看中间部分是否更新。路由器设置并工作。我们将为汽车的 CRUD 操作添加更多的路由，但它们将受到保护-用户必须提供有效的 JWT
令牌来访问它们（即使用户可以访问 React
路由，但没有令牌，也无法在后端执行操作）。现在是介绍另一个 React
钩子的时候了- useContext。 上下文是解决 React 中所知的问题的一种方法，即当位于组件树下方的组件需要通过一系列组件-父组件来接受
props，而这些父组件并不需要这些 props 时。上下文是一种分享值（字符串、数字值、列表和对象）的方法，这些值与封装在上下文提供程序中的所有组件分享。useContext Hook-用于与上下文交互-是新的基于 Hook 的 React
中最酷的功能之一，也是可以处理许多常见设计问题的东西。
使用上下文有点特别，不像 useState 或 useEffect
钩子一样，因为它涉及到更多的移动部件，但我们将使用最简单的版本，与自定义 Hook 结合使用，以便更容易地访问。 第一步是创建上下文，使用 React 提供的 createContext 函数。它接受默认参数，因此您可以将其提供给字典：{username：“Marko”}。除非提供其他值，否则仅使用此参数。甚至可以将用于设置或修改上下文值的函数传递给上下文，而这正是我们要做的。我们可以设置一个 auth
值，用于存储已登录用户的数据（如果有），但也可调用 setAuth
函数来设置用户数据。我们还可以使用此函数注销用户，只需将 auth
的上下文值设置为 null 值即可。
第二步是使用上下文提供程序--一个允许其他组件消耗上下文的 React 组件。所有包含在提供程序中的消费者都会在上下文（提供程序的值）更改时重新呈现。提供程序是提供上下文值（而不是 props）给子组件的工具。 现在是钩子 useContext 的时候了，它将上下文作为参数，并使其可用于组件。我们将使用它来访问上下文。让我们继续进行示例，因为它将变得更加清晰。按照以下步骤操作： 1。 我将在 /src/context/AutProvider.js
文件中创建最简单的上下文，其中包含单个状态变量，称为 auth（使用 useState 钩子，setAuth）： import { createContext, useState } from "react"; const AuthContext = createContext({}) export const AuthProvider = ({children}) => { const [auth, setAuth] = useState({

在React中进行用户身份验证 1. 首先，我们要创建一个具有认证状态（auth）和设置认证状态（setAuth）功能的Context，这样我们就可以在所有组件中使用它。在/src/context文件夹中，创建一个文件AuthProvider.js： import { createContext, useState } from 'react' const AuthContext = createContext() const AuthProvider = ({children}) => { const [auth, setAuth] = useState(false) return {children} } export default AuthContext 2. 然后，我们可以在index.js文件中包装Router的路由，使得auth和setAuth对所有路由都可用。编辑index.js文件： import { AuthProvider } from './context/AuthProvider'; … } />
最后，我们不想在每个组件中都导入AuthContext provider和useContext，所以我们将创建一个简单的实用Hook，它将为我们导入上下文。 3.
在/src/hooks文件夹中，创建一个名为useAuth.js的文件： import { useContext } from "react"; import AuthContext from "../context/AuthProvider"; const useAuth = () => { return useContext(AuthContext) } export default useAuth;
这个设置可能看起来很复杂，但实际上并不复杂——我们只需创建一个上下文和一个Hook来简化我们的工作。好处是现在我们可以覆盖整个应用程序范围，并设置和获取auth变量的值。让我们开始使用我们的React身份验证机制，并创建将实际让我们登录的Login组件。为了处理表单，我想介绍一个第三方包：React-Form-Hook（https://react-hook-form.com/）。

认证和授权 我们已经看到在React中手动处理表单会变得非常繁琐，因此有一些经过良好测试的解决方案。在本章中，我们将使用React form hook。让我们开始安装它：
npm install react-hook-form 使用npm run start重新启动React服务器，并打开Login.jsx组件。从逻辑上说，这可能是最复杂的组件，因此让我们将其分解： import {useForm} from "react-hook-form"; import {useState} from "react"; import {useNavigate} from "react-router-dom"; import useAuth from "../hooks/useAuth"; 我们导入useForm
Hook，useState用于一些状态变量，使用路由器的useNavigate
Hook进行登录后重定向，使用我们的useAuth
Hook设置身份验证上下文。然后开始编写组件并设置Hook： const Login = () => { const [apiError, setApiError] = useState() ; const {setAuth} = useAuth() ; let navigate = useNavigate() ; const { register, handleSubmit, formState: {errors}, } = useForm() ; ApiError变量应该是不言自明的-我将使用它来存储从后端生成的潜在错误，以便稍后显示它们。导航对于在路由器内部对不同页面进行编程导航是必要的，而react-form-hook为我们提供了几个有用的工具：register用于向Hook实例注册表单输入，handleSubmit用于处理表单提交，而errors将在过程中托管错误。让我们继续编写代码： const onFormSubmit = async (data) => { const response = await fetch("http://127.0.0.1:8000/users/ login", { method: "POST", headers: { "Content-Type": "application/json",

在React中验证用户 185 }, body: JSON.stringify(data), }); if (response.ok) { const token = await response.json(); await getUserData(token["token"]); } else { let errorResponse = await response.json(); setApiError(errorResponse["detail"]); setAuth(null); } }; const onErrors = (errors) => console.error(errors); onSubmit与我们手动执行的操作非常相似：我们将表单数据编码为JSON并发送到/login端点的POST请求。如果一切正常（OK响应，它的响应代码在200-299的范围内），我们将继续获取token。然后我们将此token提供给另一个名为getUserData的函数。如果API发送任何错误，我们将获取这个错误并将其放在apiError状态变量中。请记住，FastAPI具有包含可读取的人类消息错误的细节键。错误仅会发送到控制台。
让我们看一下getUserData函数-它只是对后端上的/me路由的调用： const getUserData = async (token) => { const response = await fetch("http://127.0.0.1:8000/users/ me", { method: "GET", headers: { "Content-Type": "application/json", Authorization: `Bearer ${token}`, }, }); if (response.ok) { let userData = await response.json(); userData["token"] = token; setAuth(userData); setApiError(null); navigate("/", { replace: true }); ----------请将这段内容翻译成中文，不要添加或删除任何内容

身份验证和授权 这是一个登录页面的React组件，它允许用户输入其凭证并进行身份验证和授权。这个组件使用useState和useContext钩子，这些钩子将处理身份验证和授权的状态和功能。 为了进行身份验证和授权，我们需要使用JSON Web Token（JWT）。在onFormSubmit函数中，我们调用一个名为fetchUser的异步函数，该函数将使用用户提供的凭证在后端上检查用户的身份验证和授权。 在fetchUser函数中，我们创造一个名为data的对象和一个名为reqOptions的对象。data对象包含用户的凭证，reqOptions则包含我们将要使用的选项。然后，我们使用fetch和reqOptions异步地获取一个JSON Web Token。 随后，我们执行一个名为getUserByToken的异步函数，该函数将使用JWT来获取用户的数据。如果我们得到一个用户（一个OK响应），我们将该用户的数据用于填充authContext中的auth对象。最后，我们使用路由器将用户发送到主页。函数的剩余部分是标记和一些实用类。 以上是该函数实际使用我们的令牌的部分——我们将其添加到请求头中，如果检索到用户（一个OK响应），我们使用该用户的数据来填充auth Context中的auth对象。最后，我们使用路由器将用户发送到主页。函数的其余部分是标记和一些实用类。

在React中对用户进行身份验证 在此示例中，我们将使用React Hook Form来处理登录表单。我们还将使用useHistory hook 来实现重定向。用法如下：
```
import React from "react"; import { useForm } from "react-hook-form"; import { useHistory } from "react-router-dom"; import useAuth from "../hooks/useAuth"; const Login = () => { const { auth, login } = useAuth(); const { register, handleSubmit, formState: { errors }, } = useForm(); const history = useHistory(); const onSubmit = async (data) => { const result = await login(data.email, data.password); if (result) { history.push("/"); } }; const apiError = auth?.error; return ( {errors?.email && errors.email.message} {errors?.password && errors.password.message} Login {apiError && ( {apiError} )} ); }; export default Login;
```
需要注意的是，表单中的每个字段都有一个register属性，将其绑定到由React表单钩子控制的表单中。如果尝试使用不存在的电子邮件或密码进行登录，将显示API错误，但如果一切顺利，应将重定向到主页。为了查看身份验证数据，我们可以查看Chrome浏览器中的React扩展。在上下文提供程序下的组件选项卡下，您应该能够看到存储在auth对象中的所有数据。 缺乏适当的导航将很难继续开发，因此让我们访问DaisyUI网站，找到合适的导航栏。在四处搜寻后，我发现了以下解决方案，需要复制并进行一些针对React Router链接结构的调整：
```
import React from "react"; import { Link } from "react-router-dom"; import useAuth from "../hooks/useAuth"; const Navbar = () => { const { auth, logout } = useAuth(); return ( Home Dashboard {!auth.isLoggedIn ? ( Login ) : ( logout()} className="btn"> Logout )} ); }; export default Navbar;
```

身份验证和授权 188 const Header = () => { const { auth, setAuth } = useAuth(); return (FARM Cars {auth?.username ? `已登录：${auth?.username} - ${auth.role}` : "未登录"} {!auth?.username && (登录)} {!auth?.username && (注册)} {auth?.username && (登出 {auth?. username})}

在React中进行用户身份验证
这是一个常规的导航菜单，带有一些上下文的好处:我们导入了useAuth Hook并立即访问authContext。这使我们能够有条件地显示或隐藏登录和注册或注销链接。我在导航栏中添加了一个小的span，以通知用户是否有任何人已经登录。由于默认主题相当乏味，我将应用DaisyUI主题 - 您可以在https://daisyui.com/docs/themes/上探索它们。我喜欢秋季的主题，因此我只需找到index.html文件，并将data-theme =“autumn”添加到html打开标记中。 我们的注销按钮没有做任何有用的事情，因此我们需要在同一Header.js文件中添加一个logout处理程序： let navigate = useNavigate(); const logout = () => { setAuth({}) navigate("/login", {replace:true}) }
然后将onClick处理程序添加到Logout按钮并将其设置为{logout}。 我们已经创建了一个非常简单的身份验证系统，但我们没有可以保护的路由，特别是涉及汽车的路由：更新，添加和删除。那就是我想在这里展示的身份验证系统的最终部分。有许多方法可以防止在React中有条件地显示或显示某些组件。一种优雅的方法是再次利用React路由 - 使用outlets。 简单地说，我们将创建一个身份验证组件，它将仅检查auth数据的存在 - 如果数据存在，则您将获得outlet，包含受保护的路由和相应的组件，如果不存在，则路由将将您发送到登 录页面（或您选择的任何页面）。
让我们创建一个名为RequiredAuthentication.jsx的组件： import { useLocation, Navigate, Outlet } from "react-router-dom"; import useAuth from "../hooks/useAuth"; const RequireAuthentication = () => { const { auth } = useAuth(); const location = useLocation; …

身份验证和授权 该组件作为一个简单的开关:如果认证对象中存在用户名，则输出返回门户组件，允许客户访问包装的任何路由。否则，它将强制导航到/login路由。这与其他使用简单功能组件的方法并没有太大不同，然后条件性地呈现保留输出或登录路由。要能够在实践中看到我们的身份验证逻辑，我们需要至少有一个受保护的路由。让我们创建一个新组件并将其称为CarList.jsx。它将简单地显示数据库中的所有汽车，但是为了能够访问，用户必须登录——作为管理员或销售人员。CarList组件具有一些标准的导入和Hooks:Car组件在这里真的不重要——它只是由DaisyUI提供的一个卡片元素，类似于我们在第6章中使用的来显示汽车信息。useAuth
hook通过Context为我们提供了一种快速检查身份验证用户信息的方法。useEffect
Hook用于调用FastAPI服务器并填充汽车数组:

在React中对用户进行身份认证 为了在React应用程序中对用户进行身份验证，我们需要创建一个新的组件RequireAuthentication，该组件检查用户是否已经登录，如果没有，则将其重定向到登录页面。 import { Navigate, useLocation } from "react-router-dom"; import { useAuth } from "../../hooks/useAuth"; const RequireAuthentication = ({ children }) => { const auth = useAuth(); const location = useLocation(); const isAuthenticated = auth.isAuthenticated() || false; if (!isAuthenticated) { return ; } return children; }; export default RequireAuthentication; 依赖于useAuth钩子和useLocation Hook。
接下来，我们将在CarList组件中使用它来保护对汽车列表的访问： import { useEffect, useState } from "react"; import Card from "./Card"; import req from "../../utils/req"; const CarList = () => { const [cars, setCars] = useState(null); useEffect(() => { req.get("/api/cars/").then((res) => { setCars(res.data.cars); }).catch((err) => { console.error(err); }); }, []);
最后，返回汽车列表的JSX只是一个对汽车数组进行映射的功能： return (Cars Page {cars && cars.map((el) => { return ; })}); }; export default CarList;
为了使该组件与应用程序连接，我们需要使用路由更新App.js文件： }> } /> } /> } /> }> } />

身份验证和授权 请注意，我们将CarList组件包装在RequireAuthentication路由中：我们可以以相同的方式添加其他需要身份验证的路由，还可以更细粒度地控制哪个用户可以访问哪个路由。轻松编辑RequireAuthentication组件并对验证用户的类型执行其他检查 - 因此我们可以为管理员创建一个区域，但不适用于普通销售人员等等。
最后，让我们也更新Header.jsx组件，以便显示到新创建的/cars路由的链接：{!auth? .username && (注册)} 汽车 我让链接对所有访问者可见 - 不管是否登录 - 以展示身份验证路由的功能；如果你点击未登录的链接，你将被发送到登录页，否则，你应该看到一个显示汽车的良好卡集。
实际上，没有必要展示应该需要身份验证的汽车的其余CRUD操作 - 我们已经看到后端如何通过读取JWT令牌来检查适当的用户，所以它只是确保令牌存在并且是有效的问题。 正如我早先强调的，身份验证和授权可能是任何应用程序中最基本和最严重的主题，它们让开发人员和利益相关者面临一系列早期需要解决的挑战和问题。虽然外部解决方案（例如Auth0，AWS Cognito，Firebase，Okta等）提供了强大和工业级别的安全性和功能，但你的项目可能需要一个自定义解决方案，其中数据的所有权被充分控制。在这些情况下，重要的是你仔细权衡你的选择，谁知道 - 也许你最终需要编写你自己的认证。毕竟，并不是所有的应用程序都是为银行业务而做的！ 总结 在本章中，我们看到了身份验证机制的一个非常基本但相当典型的实现。我们已经看到了FastAPI如何使我们能够使用符合标准的身份验证方法，并且我们实现了一种最简单但有效的解决方案。

概述 本章我们学习了使用FastAPI和MongoDB定义细粒度角色和权限的优雅和灵活，通过Pdantic作为中间人来完成。本章仅关注JWT令牌作为通信手段，因为它是现代单页应用程序中主要且最受欢迎的工具，它可以在服务或微服务之间实现出色的连接。最后，我们创建了一个简单的React应用程序，实现了在内存中存储用户数据的登录机制。本意是不展示有关持久化JWT令牌的任何解决方案 - 重点只是看看React应用程序如何处理经过身份验证和未经身份验证的用户。使用本地存储和Cookie都有其优点和漏洞（更多是本地存储），但对于安全要求非常低的应用程序，它们都是可行的解决方案。再次强调，FARM堆栈可作为一个很好的原型工具，因此了解在创建认证流程时的操作方法，即使不是最理想或完全牢固的，可能也足以帮助你跨越这个MVP的难关，赛跑寻求下一个伟大的数据驱动产品！在下一章中，我们将看到如何将基于MongoDB和FastAPI的后端集成到强大的React框架 - Next.js -
中，并覆盖一些标准的Web开发任务，如图片和文件上传、带有httpOnly
cookie的身份验证、简单数据可视化、发送电子邮件以及利用堆栈的灵活性。