8 Papéis Usuários

Resumo: *Nem todos os usuários de aplicações web são criados iguais. Na maioria das aplicações, uma pequena porcentagem de usuários recebe poderes extras para ajudar a manter a aplicação funcionando sem problemas. Administradores são o melhor exemplo, mas em muitos casos, usuários avançados de nível intermediário, como moderadores de conteúdo, também existem. Para implementar isso, todos os usuários recebem uma função. Existem várias maneiras de implementar funções em uma aplicação. O método apropriado depende muito de quantas funções precisam ser suportadas e de quão complexas elas são. Por exemplo, uma aplicação simples pode precisar de apenas duas funções, uma para usuários comuns e uma para administradores. Nesse caso, ter um campo booleano* `is_administrator` *no modelo* `User` *pode ser suficiente. Uma aplicação mais complexa pode precisar de funções adicionais com níveis variados de poder entre usuários comuns e administradores.Em algumas aplicações, pode nem fazer sentido falar sobre funções discretas e, em vez disso, conceder aos usuários um conjunto de permissões individuais pode ser a abordagem correta. A implementação de funções de usuário apresentada neste sessão é um híbrido entre funções discretas e permissões. Os usuários recebem uma função discreta, mas cada função define quais ações ela permite que seus usuários executem por meio de uma lista de permissões.*

Representação de funções em banco de dados

Apresentamos a seguir um modelo de papéis com um campo padrão. Este campo deve ser definido como Verdadeiro para apenas uma função e Falso para todas as outras. A função marcada como padrão será aquela atribuída a novos usuários no registro. Como o aplicativo irá pesquisar a tabela de funções para encontrar a função padrão, esta coluna está configurada para ter um índice, o que tornará as pesquisas muito mais rápidas.

```
#   app/models.py

class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')
```

```
    def __init__(self, **kwargs):
        super(Role, self).__init__(**kwargs)
        if self.permissions is None:
            self.permissions = 0
```

Outra adição ao modelo é o campo de permissões, que é um valor inteiro que define a lista de permissões para a função de forma compacta. Como o `SQLAlchemy` definirá este campo como `None` por padrão, um construtor de classe é adicionado para defini-lo como 0 se um valor inicial não for fornecido nos argumentos do construtor.

A lista de tarefas para as quais as permissões são necessárias é obviamente específica do aplicativo. Para o `Flask`, a lista é mostrada na Tabela a seguir.

Nomes das tarefas	Nomes das permissões	Valores das permissões
Usuários Seguidores	FOLLOW	1
Comentam em Posts	COMMENT	2
Escrevem Artigos	WRITE	4
Moderadores de comentários	MODERATE	8
Acesso Administradores	ADMIN	16

A vantagem de usar potências de dois para valores de permissão é que isso permite que as permissões sejam combinadas, dando a cada combinação possível de permissões um valor exclusivo para armazenar no campo de permissões da função. Por exemplo, para uma função de usuário que concede aos usuários permissão para seguir outros usuários e comentar em postagens, o valor da permissão é `FOLLOW + COMMENT` = 3. Essa é uma maneira muito eficiente de armazenar a lista de permissões atribuídas a cada função.

A representação do código da Tabela é mostrada a seguir.

```
#   app/models.py

class Permission:
    FOLLOW = 1
    COMMENT = 2
    WRITE = 4
    MODERATE = 8
    ADMIN = 16
```

Com as constantes de permissão implementadas, alguns novos métodos podem ser adicionados ao modelo de função para gerenciar permissões.

```
#   app/models.py

class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')
```

```
    def __init__(self, **kwargs):
        super(Role, self).__init__(**kwargs)
        if self.permissions is None:
            self.permissions = 0

    @staticmethod
    def insert_roles():
        roles = {
            'User': [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE],
            'Moderator': [Permission.FOLLOW, Permission.COMMENT,
                          Permission.WRITE, Permission.MODERATE],
            'Administrator': [Permission.FOLLOW, Permission.COMMENT,
                              Permission.WRITE, Permission.MODERATE,
                              Permission.ADMIN],
        }
        default_role = 'User'
        for r in roles:
            role = Role.query.filter_by(name=r).first()
            if role is None:
                role = Role(name=r)
            role.reset_permissions()
            for perm in roles[r]:
                role.add_permission(perm)
            role.default = (role.name == default_role)
            db.session.add(role)
        db.session.commit()

    def add_permission(self, perm):
        if not self.has_permission(perm):
            self.permissions += perm

    def remove_permission(self, perm):
        if self.has_permission(perm):
            self.permissions -= perm

    def reset_permissions(self):
        self.permissions = 0

    def has_permission(self, perm):
        return self.permissions & perm == perm

    def __repr__(self):
        return '<Role %r>' % self.name
```

Os métodos `add_permission()`, `remove_permission()` e `reset_permission()` usam operações aritméticas básicas para atualizar a lista de permissões. O método `has_permission()` é o mais complexo do conjunto, pois se baseia no operador **&** para verificar *bit* a *bit* se um valor de permissão combinado inclui a permissão básica fornecida. Você pode usar estes métodos em um shell Python:

```
(venv) $ flask shell
>>> r = Role(name='User')
>>> r.add_permission(Permission.FOLLOW)
>>> r.add_permission(Permission.WRITE)
>>> r.has_permission(Permission.FOLLOW)
True
>>> r.has_permission(Permission.ADMIN)
False
>>> r.reset_permissions()
>>> r.has_permission(Permission.FOLLOW)
False
```

A próxima Tabela mostra a lista de funções de usuário que serão suportadas neste aplicativo, juntamente com as combinações de permissões que definem cada uma delas.

Papei Usuários	Permissões	Descrições
`None`	Nenhuma	Acesso somente leitura ao aplicativo. Isso se aplica a usuários desconhecidos que não estejam logados.
`User`	FOLLOW, COMMENT, WRITE	Permissões básicas para escrever artigos e comentários e seguir outros usuários. Esta é a permissão padrão para novos usuários.
`Moderator`	FOLLOW, COMMENT, WRITE, MODERATE	Adiciona permissão para moderar comentários feitos por outros usuários.
`Administrator`	FOLLOW, COMMENT, WRITE, MODERATE, ADMIN	Acesso total, que inclui permissão para alterar as funções de outros usuários.

Adicionar as funções ao banco de dados manualmente é demorado e propenso a erros, portanto, em vez disso, um método de classe pode ser adicionado à classe Role para essa finalidade. Isso facilitará a recriação das funções e permissões corretas durante os testes unitários e, mais importante, no servidor de produção após a implantação do aplicativo.

A função `insert_roles()` não cria diretamente novos objetos de função. Em vez disso, ela tenta encontrar funções existentes por nome e atualizá-las. Um novo objeto de função é criado apenas para funções que ainda não estão no banco de dados. Isso é feito para que a lista de funções possa ser atualizada no futuro, quando alterações precisarem ser feitas.

Para adicionar uma nova função ou alterar as atribuições de permissão para uma função, altere o dicionário de funções no topo da função e execute a função novamente. Observe que a função *"Anonymous"* não precisa ser representada no banco de dados, pois é a função que representa usuários que não são conhecidos e, portanto, não estão no banco de dados.

Observe também que `insert_roles()` é um método estático, um tipo especial de método que não requer a criação de um objeto, pois pode ser invocado diretamente na classe, por exemplo, como `Role.insert_roles()`. Métodos estáticos não recebem um argumento próprio como métodos de instância.