Authentication and Authorization

Ensure appropriate access for appropriate people

Pascal Sun

UWA NLP-TLP Group

The University of Western Australia

December 6, 2024

Table of Contents

Concepts

Authentication vs Authorization

Authentication (AuthN)

- Verifies who you are
- Like showing your ID at airport
- Common methods:
  - Username/password
  - Biometrics
  - Security tokens

Authorization (AuthZ)

- Determines what you can do
- Like hotel key card access
- Implementation through:
  - Roles (RBAC)
  - Permissions
  - Access Control Lists

Authentication Methods

Common Authentication Methods

- **Knowledge-based:**
  - Passwords
  - Security questions
  - PIN numbers
- **Possession-based:**
  - Security tokens
  - Mobile devices
  - Smart cards
- **Inherence-based:**
  - Fingerprints
  - Face recognition
  - Voice recognition

Authorization Types

Role-Based Access Control (RBAC):

- Permissions assigned to roles
- Users assigned to roles
- Example: Admin, Editor, Viewer

Attribute-Based Access Control (ABAC):

- Based on user attributes
- Context-aware decisions
- Example: Location, time, device

Resource-Based Access Control:

- Permissions tied to resources
- Object-level security
- Example: Document ownership

Authentication Flow Example

1. User enters credentials
2. System validates credentials
3. If valid:
   - Generate session/token
   - Store authentication state
   - Return success response
4. If invalid:
   - Return error message
   - Log failed attempt
   - Implement security measures

Best Practices

Security Guidelines

- Use strong password policies
- Implement MFA where possible
- Secure credential storage
- Regular security audits
- Session management
- Rate limiting

Common Mistakes

- Plain text password storage
- Weak password requirements
- Missing access controls
- Insufficient logging

JWT

The Authentication Challenge

Modern Web Architecture:

- Frontend and Backend are separate applications
- They communicate through HTTP/WebSocket APIs
- Often running on different domains

The Problem:

1. User logs in with username/password
2. But... we can't ask for password on every request!
3. We need a way to remember authenticated users
4. And verify their identity for each request

Traditional Solutions and Their Limitations

Session-Based Authentication:

- Server stores session information
- Gives client a session ID
- Problems:
  - Needs server storage
  - Doesn't scale well
  - Hard for multiple backends

We Need Something Better:

- No server storage required
- Can verify user without database
- Works across multiple services
- Secure and tamper-proof

Enter JWT: The Solution

JWT as Digital ID Card

How it Works:

1. User logs in once with credentials
2. Server creates a special digital ID (JWT)
3. Server signs it with a secret key
4. User presents this ID for future requests
5. Server can verify it's genuine using the signature

Just Like a Physical ID Card:

- Contains user information
- Has security features (signature)
- Can be verified without calling issuer
- Has expiration date

JWT Workflow

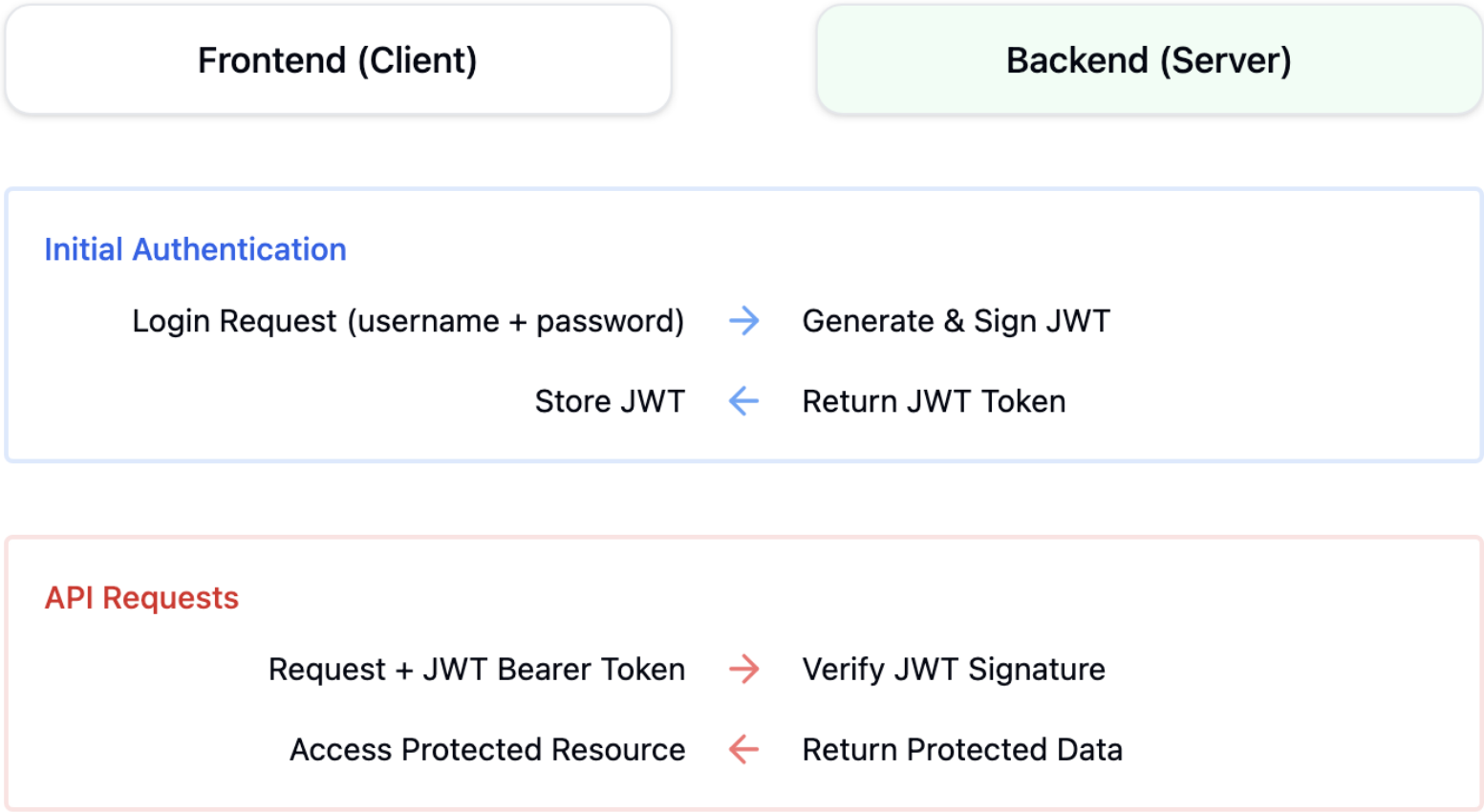


Figure: The workflow for JWT request

What is JWT?

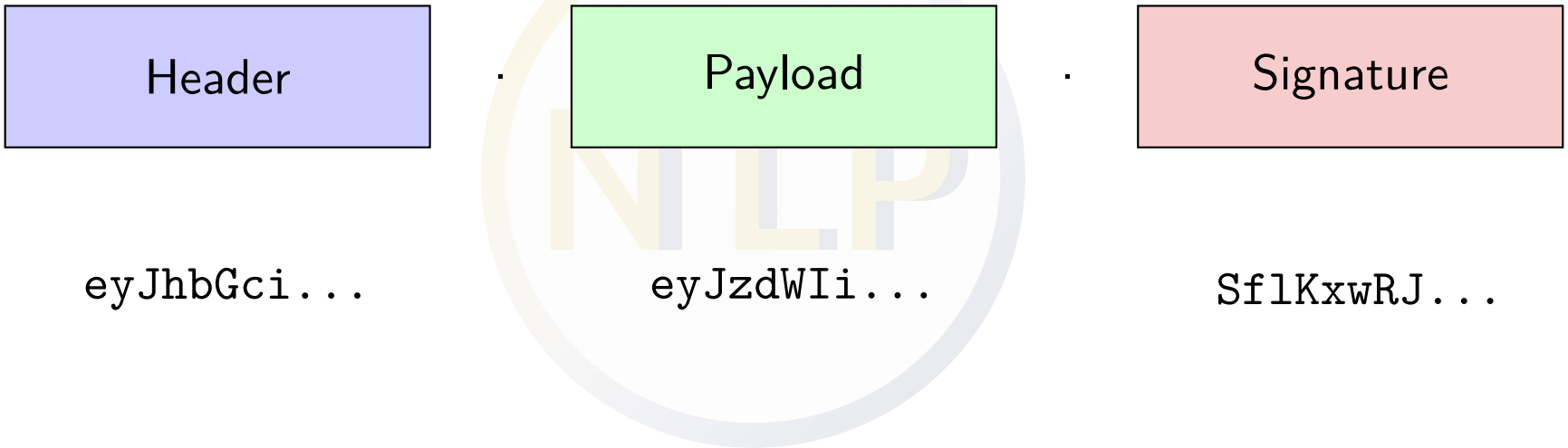
JSON Web Token

- An open standard (RFC 7519)
- Self-contained way to securely transmit information
- Information is digitally signed
- Can be verified and trusted
- Compact format: Header.Payload.Signature

Common Use Cases

- Authentication
- Information Exchange
- Authorization
- API Security

JWT Structure



JWT Header

Header Contains:

```
{
  "alg": "HS256",  // Algorithm used
  "typ": "JWT"     // Token type
}

```

Common Algorithms (alg):

- **HS256:** HMAC with SHA-256
- **RS256:** RSA with SHA-256
- **ES256:** ECDSA with SHA-256

JWT Payload

Contains Claims:

```
{
  // Registered claims
  "iss": "auth0.com",    // Issuer
  "sub": "123456",       // Subject
  "exp": 1516239022,     // Expiration Time
  "iat": 1516235422,     // Issued At

  // Custom claims
  "name": "John Doe",
  "role": "admin",
  "permissions": ["read", "write"]
}

```

JWT Signature

Signature Creation:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)

```

Purpose

- Verifies sender identity
- Ensures message wasn't changed
- Prevents token forgery
- Maintains data integrity

JWT Best Practices

Security Guidelines

- Keep tokens secure (use HTTPS)
- Set appropriate expiration times
- Use strong secret keys
- Don't store sensitive data in payload
- Implement token refresh mechanism

Common Mistakes

- Using weak secret keys
- Storing tokens insecurely
- Not validating claims
- Putting sensitive data in payload

JWT vs Session Tokens

JWT

- Stateless
- Self-contained
- Larger size
- Can't be invalidated
- Good for microservices

Session Tokens

- Stateful
- Server-side storage
- Smaller size
- Can be invalidated
- Better for monoliths

Practice in Django

Traditional Django Authentication

Built-in Authentication

- Based on `django.contrib.auth`
- Session-based authentication
- Uses database backend
- Integrated with Django templates

```python
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.sessions',
]

MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
]

```

Session Authentication Flow

1. User submits login form
2. Django validates credentials
3. Creates session in database
4. Sets session ID in cookie
5. Subsequent requests include cookie

```python
from django.contrib.auth import login, authenticate

def login_view(request):
    user = authenticate(
        username=username,
        password=password
    )
    if user:
        login(request, user)  # Creates session

```

Modern Django REST Authentication

Challenges with Traditional Approach:

- Frontend/Backend separation
- Mobile app support
- Cross-origin requests
- Stateless architecture

Solution:

- Django REST Framework
- JWT Authentication
- Token-based approach

Setting Up JWT in Django

```python
# settings.py
INSTALLED_APPS = [
    'rest_framework',
    'rest_framework_simplejwt',
]

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ],
}

# JWT settings
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
    'ROTATE_REFRESH_TOKENS': False,
}

```

JWT Views and URLs

```python
# urls.py
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
)

urlpatterns = [
    path('api/token/',
         TokenObtainPairView.as_view()),
    path('api/token/refresh/',
         TokenRefreshView.as_view()),
]

```

Authentication Flow:

1. POST credentials to `/api/token/`
2. Receive access & refresh tokens
3. Include token in requests

Protecting Views with JWT

```python
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated
@api_view(['GET'])
@permission_classes([IsAuthenticated])
def protected_view(request):
    return Response({'message': 'Protected data','user': request.user.
     username})

```

Key Points

- DRF handles token validation
- User automatically added to request
- Permissions checked before view execution

Custom Token Claims

```python
from rest_framework_simplejwt.serializers import (
    TokenObtainPairSerializer
)

class CustomTokenSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)

        # Add custom claims
        token['username'] = user.username
        token['email'] = user.email
        token['role'] = user.role

        return token

```

- Extend token payload
- Add user-specific data
- Available in frontend

CORS Configuration

```
INSTALLED_APPS = [
    'corsheaders',
]

MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
]

CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",
    "https://frontend.example.com",
]

CORS_ALLOW_CREDENTIALS = True

```

Testing JWT Authentication

```python
from rest_framework.test import APITestCase
from rest_framework_simplejwt.tokens import RefreshToken

class AuthTests(APITestCase):
    def setUp(self):
        self.user = User.objects.create_user(
            username='test',
            password='test123'
        )
        self.token = RefreshToken.for_user(self.user)
        self.client.credentials(
            HTTP_AUTHORIZATION=f'Bearer {self.token.access_token}'
        )

    def test_protected_view(self):
        response = self.client.get('/api/protected/')
        self.assertEqual(response.status_code, 200)

```

Hasura AuthN/AuthZ

Multi-Service Architecture

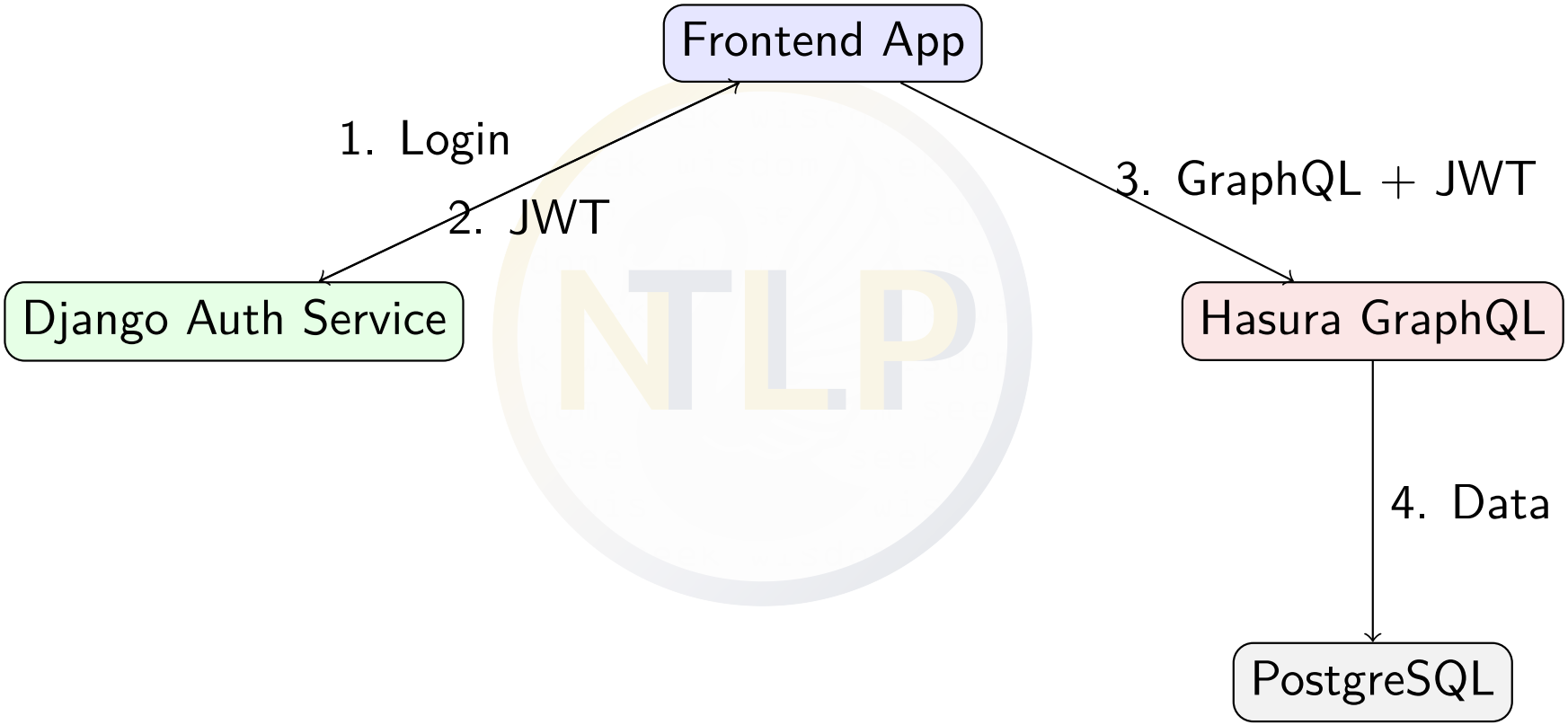


Figure: Multi-Service JWT Flow

Hasura Authentication Overview

Hasura JWT Requirements

- Hasura expects specific JWT claim format
- Uses `x-hasura-*` custom claims
- Role-based access control (RBAC)
- Row-level security (RLS)

```
{
  "sub": "1234567890",
  "https://hasura.io/jwt/claims": {
    "x-hasura-allowed-roles": ["user", "admin"],
    "x-hasura-default-role": "user",
    "x-hasura-user-id": "1234567890"
  }
}

```

Django Custom Claims for Hasura

```python
from rest_framework_simplejwt.serializers import (
    TokenObtainPairSerializer
)

class HasuraTokenSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)

        # Add Hasura-specific claims
        hasura_claims = {
            "https://hasura.io/jwt/claims": {
                "x-hasura-allowed-roles": [
                    user.role,
                    "user"
                ],
                "x-hasura-default-role": "user",
                "x-hasura-user-id": str(user.id),
                "x-hasura-org-id": str(user.organization_id)
            }
        }
        token.update(hasura_claims)
        return token
```

Hasura Configuration

```
# environment variable
HASURA_GRAPHQL_ADMIN_SECRET: "${ADMIN_SECRET}"
HASURA_GRAPHQL_UNAUTHORIZED_ROLE: "anonymous"

```

Important Settings

- JWT secret must match Django's
- Configure allowed roles
- Set unauthorized role

Hasura Permissions

Role-Based Access Control:

```
# Example permission rule
{"filter": {
    "_or": [
      {"org_id": {"_eq": "X-Hasura-Org-Id"}},
      {"created_by": {"_eq": "X-Hasura-User-Id"}}]}}

```

Permission Types

- Row-level select permissions
- Column-level restrictions
- Operation-based (insert/update/delete)
- Dynamic variables from JWT claims

Row and Column Level secure

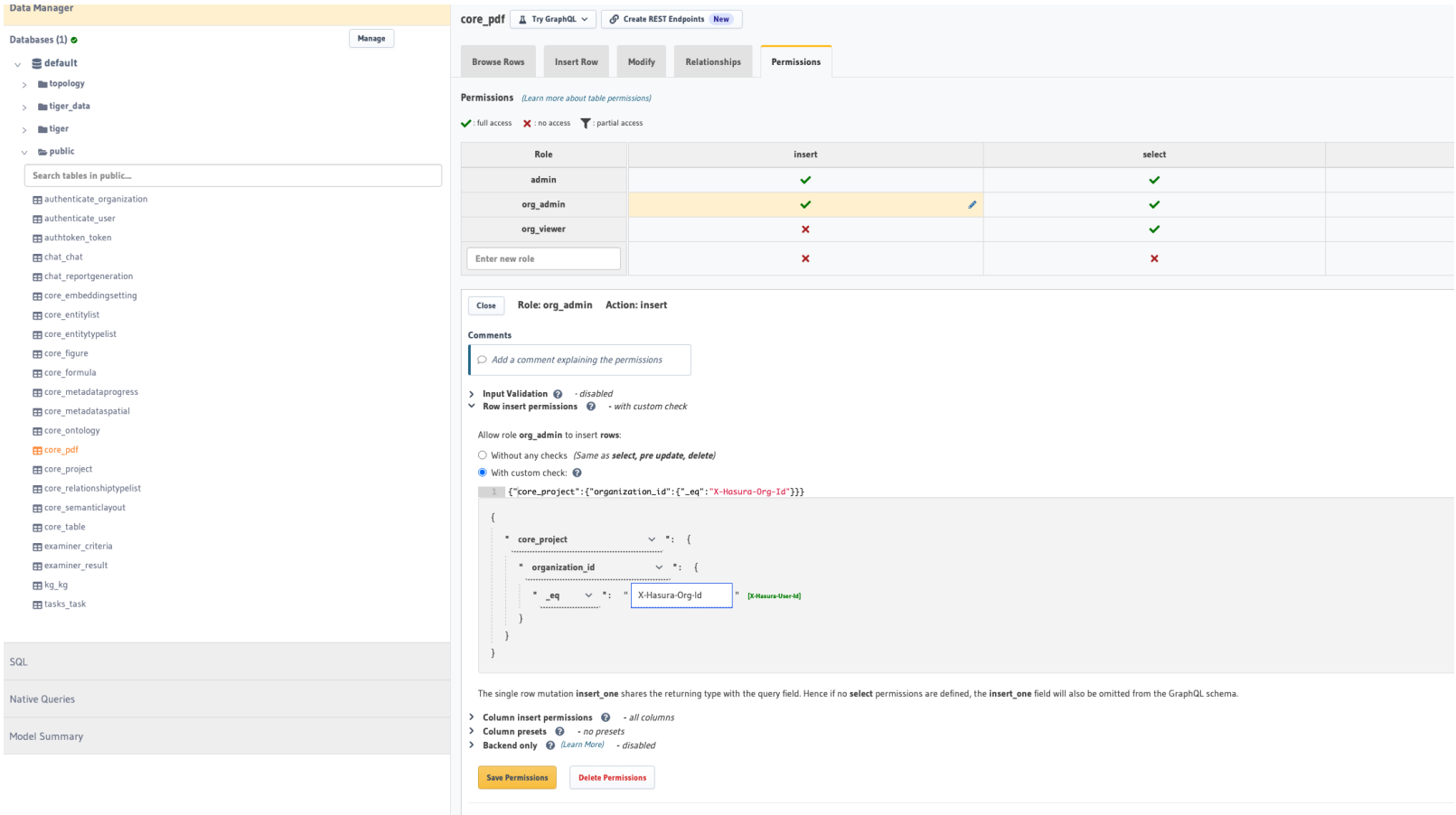


Figure: Hasura Permission Control Setting

Testing Hasura Integration

```python
class HasuraIntegrationTests(APITestCase):
    def test_jwt_claims_format(self):
        user = User.objects.create_user(
            username='test',
            password='test123',
            role='admin'
        )
        token = HasuraTokenSerializer.get_token(user)
        claims = token.payload

        self.assertIn('https://hasura.io/jwt/claims', claims)
        self.assertEqual(
            claims['https://hasura.io/jwt/claims']
            ['x-hasura-default-role'],
            'user'
        )

```

Frontend

Frontend Authentication Overview

Core Concepts

- JWT stored in cookies/localStorage
- Route protection based on JWT payload
- Role-based access control (RBAC)
- Dynamic UI based on permissions

Key Implementation Points:

1. Call api to get the JWT token
2. Store and manage JWT
3. Decode JWT payload
4. Check permissions
5. Protect components

JWT in Frontend

```
// types/auth.ts
interface JWTPayload {
  user_id: string;
  roles: string[];
  permissions: string[];
  exp: number;
}

// utils/jwt.ts
import jwt_decode from 'jwt-decode';
export function getPayload(): JWTPayload | null {
  const token = localStorage.getItem('token');
  if (!token) return null;

  try {
    return jwt_decode(token);
  } catch {
    return null;
  }
}

```

JWT in Frontend

```
export function hasPermission(permission: string): boolean {
  const payload = getPayload();
  return payload?.permissions.includes(permission) ?? false;
}

```

Permission-Based UI Components

```
// components/ProtectedComponent.tsx
interface Props {
  requiredPermission: string;
  children: React.ReactNode;
  fallback?: React.ReactNode;
}
export function ProtectedComponent({
  requiredPermission,
  children,
  fallback = null
}: Props) {
  const hasAccess = hasPermission(requiredPermission);

  if (!hasAccess) return fallback;

  return children;
}

```

Permission-Based UI Components

```
// continue from last one
// Usage example
<ProtectedComponent
  requiredPermission="posts.create"
  fallback={<p>No access</p>}
>
  <CreatePostButton />
</ProtectedComponent>

```

Role-Based Layout

```
// layouts/DashboardLayout.tsx
export default function DashboardLayout({
  children
}: {
  children: React.ReactNode
}) {
  const payload = getPayload();
  const role = payload?.roles[0];

```

Role-Based Layout

```
  return (
    <div className="layout">
      <Sidebar>
        {role === 'admin' && (
          <>
            <NavItem href="/admin/users">Users</NavItem>
            <NavItem href="/admin/settings">Settings</NavItem>
          </>
        )}
        {role === 'editor' && (
          <>
            <NavItem href="/posts">Posts</NavItem>
            <NavItem href="/media">Media</NavItem>
          </>
        )}
      </Sidebar>
      <main>{children}</main>
    </div>
  );
}

```

Complete Flow

1. **Login & Store Token**
   - Receive JWT from backend
   - Store in secure location
   - Decode payload for use
2. **Route Protection**
   - Check token existence
   - Verify permissions
   - Redirect if needed
3. **UI Rendering**
   - Show/hide based on permissions
   - Dynamic navigation
   - Protected components

Python Package

Secure API Access with Django REST Framework

- Token-based authentication flow:
  1. User requests token with credentials
  2. Server validates and returns token
  3. Client includes token in API requests

Server Setup: Django REST Framework

```
# settings.py
INSTALLED_APPS = [
    'rest_framework',
    'rest_framework.authtoken',
]

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ]
}

# views.py
from rest_framework.authtoken.views import obtain_auth_token
from rest_framework.permissions import IsAuthenticated


```

Client Usage: Python Package

```python
import requests
# Use token
def api_request(token):
    headers = {'Authorization': f'Token {token}'}
    response = requests.get(
        'https://api.example.com/api/data',
        headers=headers
    )
    return response.json()

# Usage example
token = xxxx
data = api_request(token)

```

Expanded discussion

Authentication & Authorization: Now and future

Authentication Flow Evolution

- Session-based → Token-based
- Traditional → JWT → OAuth
- Single-factor → Multi-factor

Modern Implementation

- Django REST + JWT
- Hasura custom claims
- Microservices architecture

Security Best Practices

- Proper JWT configuration
- HTTPS everywhere
- Secure token storage
- Regular security audits

Key Considerations

- Scalability needs
- Security requirements
- User experience
- Compliance standards

Remember

Authentication (who you are) + Authorization (what you can do) = Secure Application?

Demo

Homework: Setup JWT for both Hasura and Django

1. Implement the Django JWT Setup with Endpoints
2. Setup Hasura to make sure it has the same JWT Secret Key
3. Login and Protected view in frontend

Q&A

Questions?

Feel free to ask anything