Designing Functional Authentication and Authorization Systems

Arunava

In this article, we are going to talk about a system for performing authentication and authorization securely. To start off with lets understand, what is the difference between Authentication and Authorization.

- **Authentication** is the process of proving, you are, who you say you are, when accessing an application.
- **Authorization** is the process of defining and enforcing access policies - that is, what you can do once you're authenticated.

In this article, we will see:

- Why Authentication and Authorization matter

- How Authentication works
- How Authorization works
- Authentication System Design
- Conclusion

Why Authentication and Authorization matter?

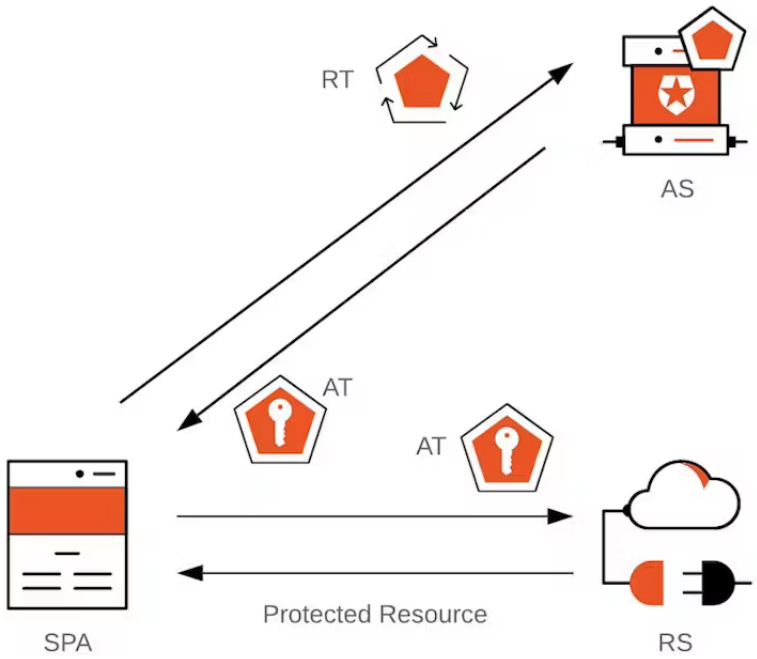
Authentication and Authorization (source: Jeffrey Marvin Forones|Geek Culture. Modified)

Lets say we are in a meeting, and you are the leading the conversation. To ask for updates/status for something to the right person, you need to identify (ie **Authenticate**) the person. Even to share some sensitive data with a person, you need to authenticate

the person correctly. And that is where authentication comes in.

Now say, in the same meeting, a few decisions needs to be made. So for that, people who have the right for taking those decisions should be the one taking the call, we cant just allow everyone to do everything. Obviously some people are not catered enough to make some decisions, and some for sure will try to make the worst out of it. So that brings **Authorization**, that gives certain people the rights permissions for certain activities.

How Authentication works?



Token Based Authentication; Access Token and Refresh Token [SPA=SinglePageApplication, RT=RefreshToken, AT=AccessToken, RS=RefreshServer, AS=AuthorizationServer] (Source : Okta)

To authenticate a person, we can assign a unique phrase to each person, and given the person tells the phrase correctly and their name. We can say that ok, we have identified the person. This is the usual usernames and passwords approach. When the right credentials are given, a system considers the identity valid and grants access. This is known as 1FA or ***Single-factor authentication***(SFA).

SFA is considered fairly insecure. Why? Because users are notoriously bad at keeping their login information secure. Multi-factor authentication (MFA) is a more secure alternative that requires users to prove their identity in more than one way. Some such ways are:

- Single-use PIN numbers / OTP
- Authentication apps run by secure 3rd party (ex. Google/Microsoft Authenticator)

- Biometrics

Once authenticated, the person would keep performing actions freely on the application. And the application is expected to have the person recognized throughout their journey without forgetting them. Ideally, it would be too much to ask the user to provide the password everytime they move to a different page, or they do some activity. So we need a way to keep the user authenticated after they have entered their credentials and they have been authenticated once. This is called **Session Management**.

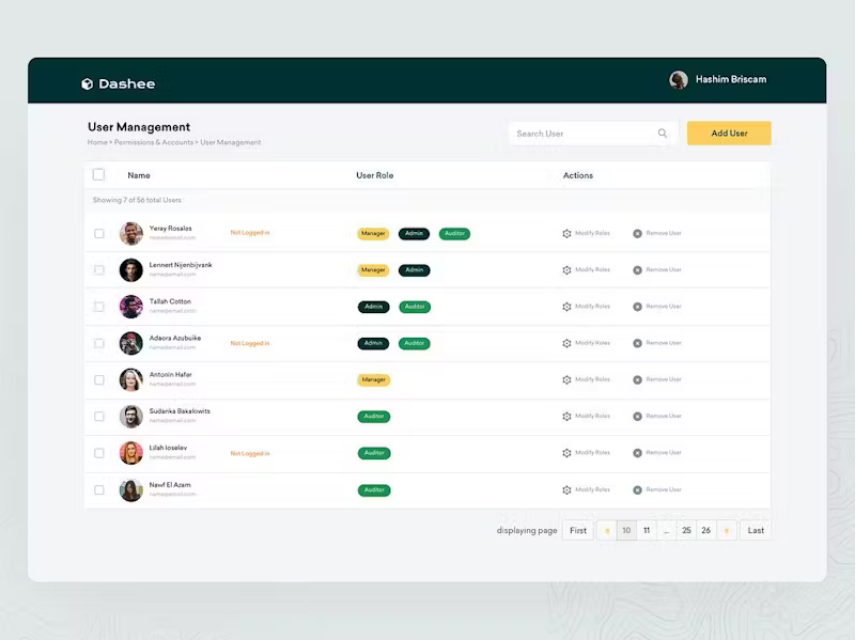
2 ways to keep the user authenticated:

- **Session-based authentication**: When a user logins to a website on a browser, the server creates a session for that user and assigns a sessionid. This sessionid is stored by the server for reference and is sent back to the user to be stored in a cookie in the browser. Now each time the user would make a request the browser will send the sessionid along with the request. Which will help in authenticating the request. And, thus, preserving authentication while the user is on the site.
- **Token-based authentication**: For this, the server creates a encrypted token which is sent to the user and is saved by the browser only, as HttpOnly Cookies. All required information such as the user information, permissions and expiry of the token are encrypted within the token. The token is sent along for calls to the server. The server simply decrypts the token with the secret key and verifies the user. This token is refreshed at intervals.

The main differences between these two approaches would be that token-based authn is **Stateless**, cause the token neednot be stored on the server side. But for session-based authentication, the token are needed to be stored on the server side as well, which makes it **Stateful**. Which brings up complications, when the system is scaled or the number of users grows.

For token-based authentication, we mostly use JWTs (JSON Web Tokens).

How Authorization works?

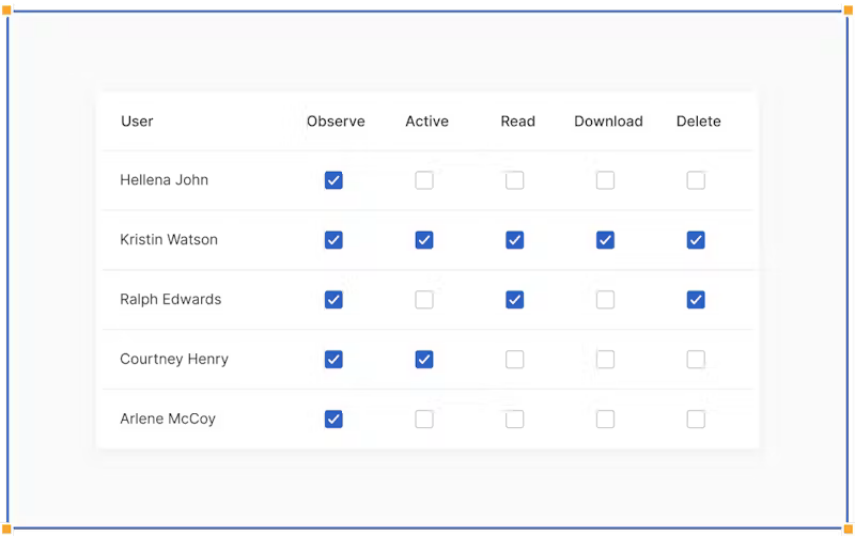


Role Based Authorization Control (RBAC) (source: Ajay Shekhawat | Dribble)

Once the user is authenticated, we would still need to ensure they're only allowed to access resources that they have permissions to

access. Unauthorized access to sensitive data can be a disaster. By the principle of least priviledge, companies would usually set up access policies such that by default you have access to what is required for you absolutely. And then in progression to that you have additional access. Common ways to segment access are:

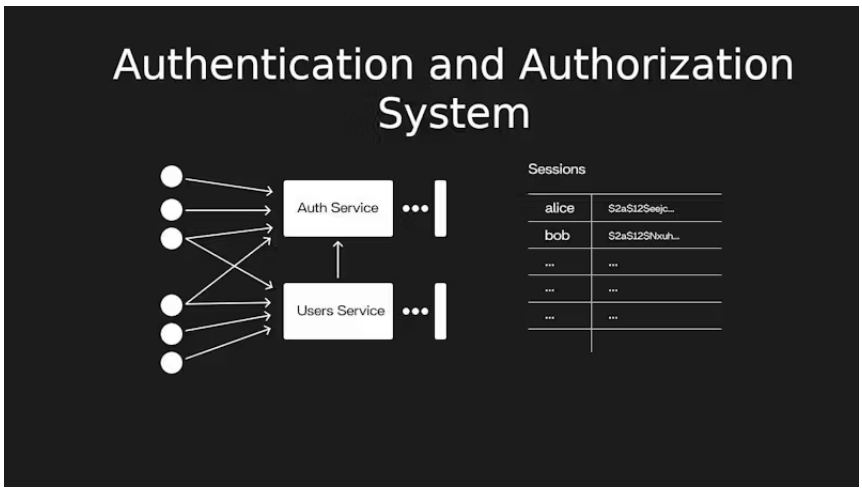
- **Role-based Access Control (RBAC)** : Users are assigned to a certain group/role that comes with set permissions. Examples: admin, member, owner.
- **Policy-based Access Control (PBAC)** : Dynamically determines access privileges during authorization based on policies and rules. Policies are based on user roles, job functions, and organizational requirements.
- **Attribute-based Access Control (ABAC)** : Users are permitted access according to attributes like title, certification, training, and/or environmental factors like location.
- **Access Control Lists (ACLs)** : Each user or entity has individual permissions that can be turned on or off, similar to installing a new app on your phone and deciding which permissions to grant (location services, contacts, etc.)



A screen for Access Control Lists (source: Povio)

ACL is frequently used at granular level than either ABAC or RBAC - for example, to grant individual users access to a certain file. ABAC and RBAC are generally instituted as company-wide policies.

Authentication System Design



Authentication and Authorization System Design (source: InterviewPen. Modified)

Requirements

Lets first start with defining the ***Functional requirements*** of the system:

- **Registration** : Allow users to register by providing necessary information.
- **Login** : Authenticate users based on their credentials.
- **Session Management** : Efficiently manage user sessions to ensure security.
- **Password Recovery** : Provide a secure process for users to recover their passwords.
- **Access Control** : Define roles and permissions for different user types.
- **Audit Trail** : Maintain detailed logs of authentication events for auditing.
- **Performance** : Ensure low latency and quick response times.

A few ***Non-functional requirements*** that we are not going to consider for the scope of this article are:

- **Multi-Factor Authentication (MFA)** : Implement a robust MFA system.
- **Security** : Prioritize data security through encryption, secure

storage, and secure communication.
- **Scalability** : Design the system to handle a growing number of users and transactions.
- **Reliability** : Minimize system downtime and ensure high availability.
- **Usability** : Develop an intuitive user interface for a seamless experience.

Capacity Estimation

Traffic Estimation

First lets start with ***Traffic Estimation***. Assuming an average traffic of 100,000 per month.We are estimating a 100k user traffic per month. Which translates to 0.04 request per second. We would need to respond to each request within 500ms 90% of the time, ie we require a p90 latency of 500ms.

```
assumed_traffic_per_month = 100000 #requests
assumed_traffic_per_day = assumed_traffic_per_month / 30
                        ~= 3350 (assuming on higher end; 3333.33 to be
precise)
estimated_time_per_request = 500 #ms; P90 of 500ms
traffic_per_second = (assumed_traffic_per_month) / (30*24*60*60)
                   = 0.04
```

Service Level Objective (SLO) : 500ms (maximal acceptable latency, immaterial of the load on the system) The average capacity 1 instance can take, based on our calculations is approximately 35ms to serve a request, assuming there are no heavy processing happening for the particular request.

Lets generate two more *derived metrics* using the above metrics.

- ***Capacity*** : Acceptable backlog per instance : Maximum number of requests(load) that can be accepted by an instance,

without compromising SLO.

- ***Demand*** : Backlog per instance : Total number of requests (load) that flows into a unit/instance based on current traffic.

Thus,

```
SLO = 500ms
approx_response_time_for_one_request = 35 #ms
capacity = SLO/approx_response_time_for_one_request
         = 500 / 35
         ~= 20

load_on_one_instance = 0.04
instances_available = 1
demand = traffic_per_second / instances_available
       = 0.04
```

With the demand and capacity available, lets calculate total number of instances required.

```
total_units_required = demand / capacity
                     = 0.04 / 20
                     = 0.002
                     ~= 1
```

Thus, we would be easily be able to handle 100k requests per month, with 0.04 requestsper second, with 1 instance. Where each unit can handle 20 requests per second without compromising SLO.

Storage Estimation

We would ideally need to store the user details for each user for authentication and authorization access. Assuming, 5kb /user

```
monthly_new_users = 500
monthly_additional_storage = 500 * 5kb
```

```
                = 2500kb
                ~= 2GB
```

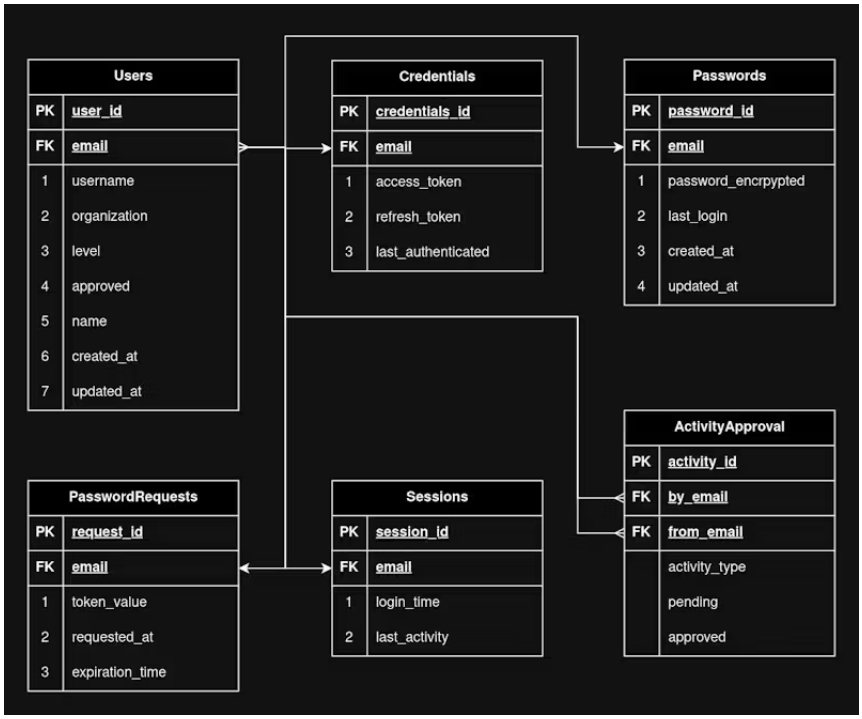
So every month, assuming we will onboard 500 new users, we will require 2GB more storage. Incase we would like to keep authentication logs. Each authentication request is expected to take 2kb to store.

```
auth_request_size = 2kb #assumption
monthly_storage = monthly_visitors * auth_request_size
                = 100,000 * 2KB
                ~= 200MB
```

Thus, each month we would require an additional of 200MB, assuming a monthly traffic of 100k.

Database Design

Now that we have the capacity estimation done. Lets create the schemas of the database required to support the functional requirements.



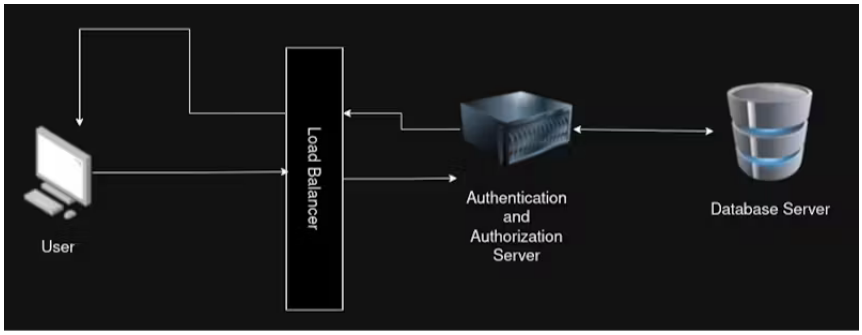
Authentication and Authotization Database Schema

Lets quickly go over the tables. We are using 6 tables.

1. Users - To store all the user information
2. Credentials - To store the access/refresh credentials once the user has been authorized.

3. Passwords - To store the user encrypted user passwords.
4. PasswordRequests - To store the password change requests that comes for a particular user.
5. Sessions - To store when the user had an active session and when was their last activity.
6. ActivityApproval - To store approval requests for a activity performed by a particular user, that would be verified by admin.

High-Level Design for the Authentication System



Authentication and Authotization HLD

System Endpoints

Endpoint	Description
/login	Authenticate user credentials.
/logout	End user session and revoke authentication tokens.
/register	Create a new user.
/update/:userId	Update user information.
/delete/:userId	Delete a user account.

/grant/:userId/:permission	Grant specific permissions to a user.
/revoke/:userId/:permission	Revoke permissions from a user.
/check/:userId/:resource	Check user's access to a specific resource.
/create/:userId	Create a new user session.
/expire/:sessionId	Expire a user session.
/validate/:sessionId	Validate an active user session.

Requirements Fulfilment

Now, with all the things in place lets see how we can complete all the requirements.

Registration

- **Requirement** - When a new user visits our application. We need to store user details so we can authorize/identify the user next time they visit.
- **Fulfilled** - When a new user visits the application, and enters user details along with their email and password. That will get captured in the database. The user details will be stored in the User table. And the password will be stored in the credentials table in an encrypted form.

Login

- **Requirement** - When an existing user visits our application. We need to identify the user so we can authorize/identify their actions and show them data that belongs to them.
- **Fulfilled** - When an existing user visits the application, and enters their details, email and password. We hash the password and match the hash against the hash stored for the user in the Credentials Table. If it matches then we have been able to identify the user successfully. Else they need to enter the correct password that they entered while registering. This process is called ***Authentication***.

Session Management

- **Requirement** - When a user has authenticated themselves, by entering their user and password. We need to make sure that they stay logged in as they perform their future actions / try to see sensitive data, without having them re-entering their password again and again.
- **Fulfilled** - When the user authenticates successfully. The Authentication server will share 2 tokens with the client. A **access_token** and a **refresh_token**. The access_token can have encrypted data with it and it has a short expiry time cause of security reasons. Once the client has the access_token, it sends the access_token back along with every request which helps in authenticating the request. When the access_token expires, the client has to ask for a new access_token using the refresh_token provided. This helps in maintaining a session.

Password Recovery

- **Requirement** - When a user forgets their password, they would need to be able to reset their password securely.
- **Fulfilled** - When the user forgets their password, they can submit their email address in the forgot password page, and we will generate a one time code and send the link to their email. When the user clicks on this link with the code and email address. We will securely be able to identify that the password recovery request is authentic. And provide the user to set their new password. And thus being able to recover the password.

Access Control

- **Requirement** - When a user performs a certain action, we need to make sure that the user is authenticated to perform that action and only then allow the action to happen.
- **Fulfilled** - We will maintain a list of actions, lets say 1-12, for simplicity. For each user we will maintain the authenticated

actions for that user. So now lets say the user tries to perform an action #id 4. We will check if the user has the permissions to perform action 4. If yes, we will go ahead and complete the request. Else we show that the request isnt successful cause of lack of permissions.

Audit Trail

- **Requirement** - Incase of a security incident, we should be able to have enough logs to look through and have a plausible reason as to what might have happened / or any leads as to what might have been a possible cause of it.
- **Fulfilled** - For such scenarios, we can keep logs for every authentication action that happens on the server. (a). For each login request, we will keep a log as to when the authentication happened, from where, ips and other relevant details. (b). For each password recovery request, we will keep a log as to when it was initiated, from where, ips, whether the request was complete and other relevant details. (c). Additionally, we will keep logs around each time a user is authorized / unauthorized for an action and by who. These logs in all, should be able to indicate what might have happened incase trying to understand some scenario.

Performance

- **Requirement** - Requirement for performance as discussed in capacity estimation section, is 0.04 requests/second and 100k requests per month.
- **Fulfilled** - We have already handled the requirement with enough servers in the capacity estimation section.

Conclusion

AUTHENTICATION	AUTHORIZATION
• Usually the first step of a security access control	• Usually comes after authentication
• Verifies the user's identity	• Grants or denies permissions to the user do something
• Common methods include: username, password, answer to a security question, code sent via SMS or email	• Permissions are granted and monitored by the organization
• Uses biometric data like fingerprint, face recognition, retinal scan	• Common methods include: role-based access control and attribute-based access control
• It's visible by the user	• It's not visible by the user
• It's changeable by the user	• Cannot be changed by the user

Authentication vs Authorization (source: OutSystems)

In this article, we started by understanding what is the difference between Authentication and Authorization. Next, we created a Authentication and Authorization System. That is safe, secure, delivers performance while catering to industry standards and

meeting all the desired requirements. Going forward i might update certain parts of the article to make it stay relevant as well as to cover more information and insights in building such a system.