auth0

Whitepaper

Fine-Grained Authorization (FGA): A technical primer

Context, concepts, and how to get started with implementing the precise and flexible access control today's SaaS apps require

okta — The World's Identity Company™

Table of contents

Introduction

In the broader domain of **Identity and Access Management (IAM)**, access controls determine what a user is authorized to do when interacting with a system.

Many access control methods exist today, ranging from the coarse-grained **role-based access control (RBAC)** — in which permissions are associated with roles, to which users are assigned — and fine-grained options including **attribute-based access control (ABAC)** and **relationship-based access control (ReBAC)**.

In the former (i.e., ABAC), access is granted based on a set of attributes and policies (e.g., user location, time, privileges); in the latter (i.e., ReBAC), access is determined by the relationship between objects and resources (e.g., a user can view a document because someone shared that document with that user or the user is the owner of the folder within which the document resides).

In practice, ReBAC enables the implementation of the extremely fine-grained authorization policies — hence the term **fine-grained authorization (FGA)** — that today's highly collaborative software-as-a-service (SaaS) apps demand.

About this technical primer

The primary aim of this document is to provide introductory technical content to help you understand what FGA is and how you can implement this type of authorization in your applications.

To those ends, we begin by briefly examining how access control requirements have evolved — particularly in recent years as digital transformation, SaaS adoption, and expectations for collaboration have imposed new security, compliance, and usability needs.

With that context established, we next summarize different approaches to authorization and access control, before focusing on FGA.

From there, we compare two different FGA solutions — Auth0 FGA and OpenFGA — and show how they are related to Google's Zanzibar.

The remainder of the document explores Auth0 FGA, including:

1. Key concepts and terminology
2. Auth0 FGA configuration language
3. A three-part guide to getting started with Auth0 FGA

An overview of access control requirements

As the world continues to move to a more digital, collaborative ecosystem of applications with ever-increasing data, carefully controlling user access is critical. Accordingly, access controls have evolved to provide more precision and greater flexibility.

Access controls contribute to a strong security posture

An effective access control solution greatly simplifies implementing and maintaining Identity-related controls, which are essential for building and maintaining a strong security posture.

When authorization is done improperly, people — or, more generally, entities — may find themselves without access to essential resources or worse, from a security perspective, unauthorized access.

Least-privilege access is the practice of limiting each user's access — and their granular rights such as read, write, execute, share, comment, etc. — to only those applications, resources, and other assets needed to perform a specific activity.

This is a security best practice designed to mitigate risks associated with application access in an increasingly complex technology landscape that encompasses remote work, cloud services, and more.

Broken authorization occupies three of the top five places on the OWASP (Open Web Application Security Project) API Security Risks 2023:

1. Broken Object Level Authorization
2. Broken Object Property Level Authorization
3. Broken Function Level Authorization

In general, the more 'coarse' an access control, the more impractical it is to precisely manage user access and the more overhang (i.e., unnecessary access) exists throughout the environment.

Okta's The State of Zero Trust Security 2023 revealed that:

61%

of organizations now have a defined Zero Trust security initiative in place

35%

of organizations plan to implement one within the next 18 months

91%

of survey respondents said that Identity is important to their Zero Trust strategy

Zero Trust calls for tighter controls

Identity and Access Management is an important element of any security strategy, and Zero Trust is no exception.

In fact, "Identity" is the first of five key pillars within the Zero Trust Maturity Model (ZTMM) developed by the Cybersecurity and Infrastructure Security Agency (CISA) to assist agencies as they implement zero trust architectures.

The term "Zero Trust" was first coined in 2010 by Forrester researcher John Kindervag to conveniently encapsulate the growing need for a "never trust, always verify" security ideal.

Zero Trust gained a major boost in May 2021, with Executive Order (EO) 14028 "Improving the Nation's Cybersecurity" — which pushed United States government agencies to adopt Zero Trust cybersecurity principles and adjust their network architectures accordingly.

The Identity subsection of the CISA ZTMM specifically advises that:

1. Agencies should ensure and enforce user and entity access to the right resources at the right time for the right purpose without granting excessive access.
2. Agencies should integrate identity, credential, and access management solutions where possible throughout their enterprise to enforce strong authentication, grant tailored context-based authorization, and assess identity risk for agency users and entities.
3. Agencies should integrate their identity stores and management systems, where appropriate, to enhance awareness of enterprise identities and their associated responsibilities and authorities.

Since its debut, Zero Trust has progressed rapidly from cool philosophy, to stretch goal, to everyday business reality.

Access controls are imperative for meeting regulatory, framework, and standards obligations

Because of Identity's essential role in securely connecting users to the technology and resources they need, Identity requirements are frequently included within a wide range of regulations, frameworks, and standards.[1]

For example:

1. Because most of the data making up corporate financial statements is created by information technology systems, carefully controlling access to these systems via IAM and related controls is vital to **Sarbanes-Oxley** compliance.
2. With threat actors increasingly targeting Identity to gain initial access and to execute intrusions, robust Identity-related controls are an essential component of cybersecurity frameworks including **SOC 2 (Service Organization Control 2).**
3. The **Payment Card Industry Data Security Standard (PCI DSS, or simply PCI)** explains industry best practices, including certain requirements about limiting
    - to the absolute minimum
    - the number of employees who can access payment card data.

Collaboration and SaaS demand more granular and flexible access controls

Use of software-as-a-service (SaaS) applications is more widespread than ever — with consumers, employees and professionals of all types logging in and out of multiple apps daily to do everything from collaborating on a work project, to checking on test results after a doctor visit, to accessing mobile banking resources.

In recent years, highly collaborative apps like Canva, Notion, and Figma have seen rapid rates of adoption,[2] putting pressure on legacy SaaS providers to add more innovative features to their own offerings.

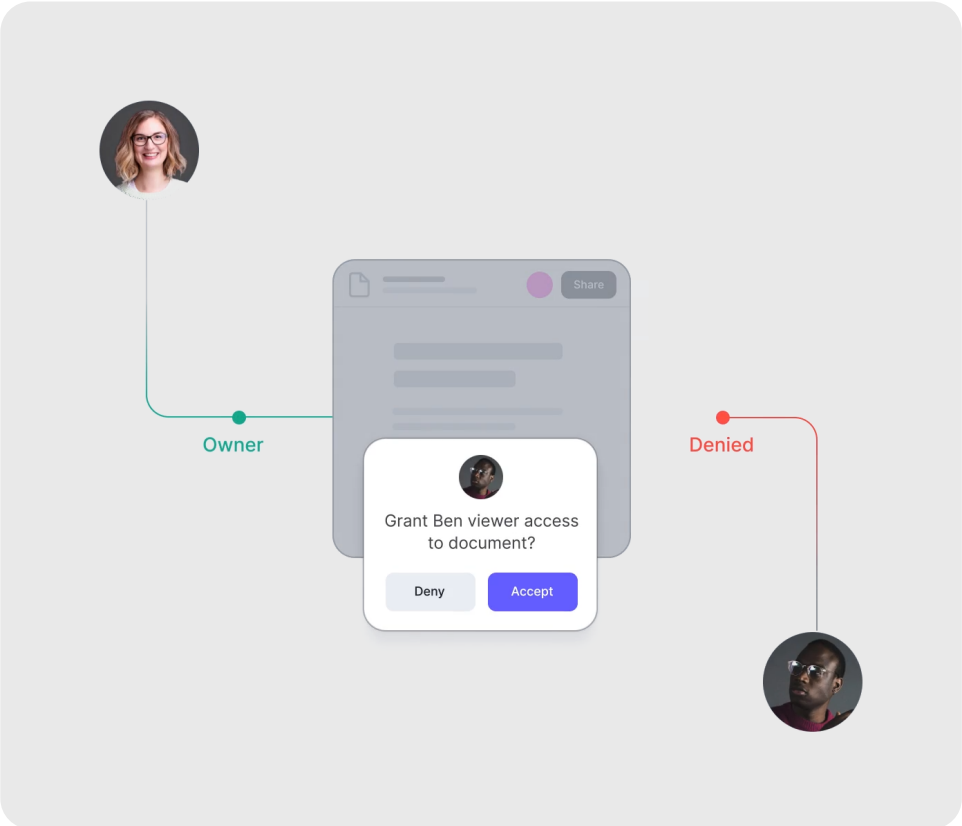
[1] Meeting Regulatory, Framework, and Standards Obligations with Okta Identity Governance, 2023

[2] Businesses at Work, 2024

But as SaaS applications grow increasingly sophisticated, collaborative, and feature-rich, authorization becomes more complex.

For example, a single document may have multiple permissions (e.g., owner, editor, commenter, view-only) that can be assigned to multiple users, according to a host of parameters (e.g., role, location, title, etc.). Relatedly, levels/hierarchies come into play when access is 'inherited' from a higher level (e.g., an organization, a parent folder).

Developers need to determine how to provide the right level of access for a growing number of applications and assets — and do it for a user base that may span employees, partners, customers, contractors and more.



Usability requires speed and availability

Access control implementation has a direct effect on application usability, because every millisecond spent in authorization can impact an application's overall latency.

Of course, access controls also need to be available. If a system becomes unavailable to some or all users — whether due to an infrastructure outage, an inability to scale with demand, or any other reason — reputation and revenue can be harmed.

Meeting speed and scale requirements is especially important for collaborative SaaS applications, where performance bottlenecks will quickly become apparent to users.

What's needed to make authorization work in a modern SaaS landscape?
Traditional methods for securing and managing access — including first-generation point solutions and in-application DIY coding — simply don't fulfill modern authorization requirements:

- **Centralized:** Developers need a means to take authorization out of the application code and implement access control and permissions centrally, across the SaaS landscape.
- **Flexible:** Authorization capabilities need to go beyond RBAC and be flexible to address the varied and constantly evolving access requirements of the SaaS user base.
- **Efficient:** Re-creating the authorization wheel every time a new application comes online is inefficient and keeps development teams from shipping new innovation quickly. Authorization solutions must be easy to deploy, adjust, and scale as needs change over time.
- **Secure and compliant:** Authorization logic that is difficult to audit and manage leads to security and compliance risks. The goal should always be to reduce points of vulnerability, not add to them.

Comparing different approaches to authorization and access control

Authorization and access control work hand in hand, but are subtly different: if authorization involves defining a policy, access control puts the policies to work. Admittedly, as both authorization policies and access controls have evolved, the line distinguishing the two terms has blurred — but the subtle differences are nevertheless something to be aware of whenever either term is encountered.

It's also worth noting that:

Authorization isn't only about human users: systems can also authorize services, APIs, devices, and other subjects to perform operations.

Access controls can be enforced without user authentication: for example, access can be defined based upon location, network, certificate, etc.

Access control methods

In general, access controls confirm that the right people have the right access to the right resources when they need it — ideally with the least amount of friction.

In addition to implementing these controls, organizations may also be required to produce reports (e.g., to achieve certification, at the request of auditors, etc.) that capture who has what level of access to what resources today and who had what level of access to what resources in the past. Such requirements make an access control solution's logging and auditability features vitally important to achieving governance objectives.

The table below summarizes four major access control methods, presented in the order in which they were developed.

Role-Based Access Control (RBAC)	In RBAC, permissions are assigned to users based on their role in a system or organization. **Example:** a user needs the editor role to edit content.access that information to make authorization decisions.

Role-Based Access Control (RBAC)

RBAC systems enable you to define users, groups, roles, and permissions, then store them in a centralized location. Applications access that information to make authorization decisions.

The main advantage of RBAC is its simplicity and ease of user management, as a single role is easier to remove than many individual permissions at a user level.

However, RBAC quickly becomes impractical when you need to specify permissions per specific resource
(e.g., a user can edit a specific document, etc.)

RBAC can also lead to 'role explosion,' as organizations attempt to group permissions in different sets
(e.g., "Title X - Team Y - Location Z").

Attribute-Based Access Control (ABAC)

In ABAC, permissions are granted based on a set of attributes that a user or resource possesses (e.g., department, location, seniority, work duties).

Example: a user assigned both marketing and manager attributes is entitled to publish and delete posts that have a marketing attribute.

Applications implementing ABAC need to retrieve information stored in multiple data sources — like RBAC services, user directories, and application-specific data sources — to make authorization decisions.

ABAC systems support much more granular permissions than RBAC, but have the downside that they require retrieving all the data required to evaluate access policies up front by querying multiple databases and services, potentially creating noticeable latency.

Policy-Based Access Control (PBAC)

PBAC is the ability to manage authorization policies in a centralized way that's external to the application code.

Policy-Based Access Control (PBAC)

PBAC is an extension of ABAC where additional dimensions (policies) can be created on top of attributes and roles. PBAC typically requires a policy engine and a policy definition language to define and enforce rules. Policy engines can be centralized or deployed with the application and policy rules are typically defined in the form of code/rules.

While moving authorization logic out of application code offers advantages, PBAC systems still need to retrieve data from multiple sources to inform the policy, and policies themselves must be managed.

Relationship-Based Access Control (ReBAC)

ReBAC enables user access rules to be conditional on relations that a given user has with a given object and that object's relationship with other objects.

Example: a given user can view a given document if the user has access to the document's parent folder.

ReBAC is a superset of RBAC. ReBAC also lets you natively solve for ABAC when attributes can be expressed in the form of relationships.

Example: 'a user's manager', 'the parent folder', 'the owner of a document', 'the user's department' can be defined as relationships.

The ReBAC and PBAC methods contain similar components (e.g., an engine and a schema/language model); however, they differ in how authorization is evaluated. With ReBAC, the access is based on a relationship (graph) between resources, stored in a centralized engine/database, allowing companies to implement authorization based on unique relationship types like hierarchy or nested relations.

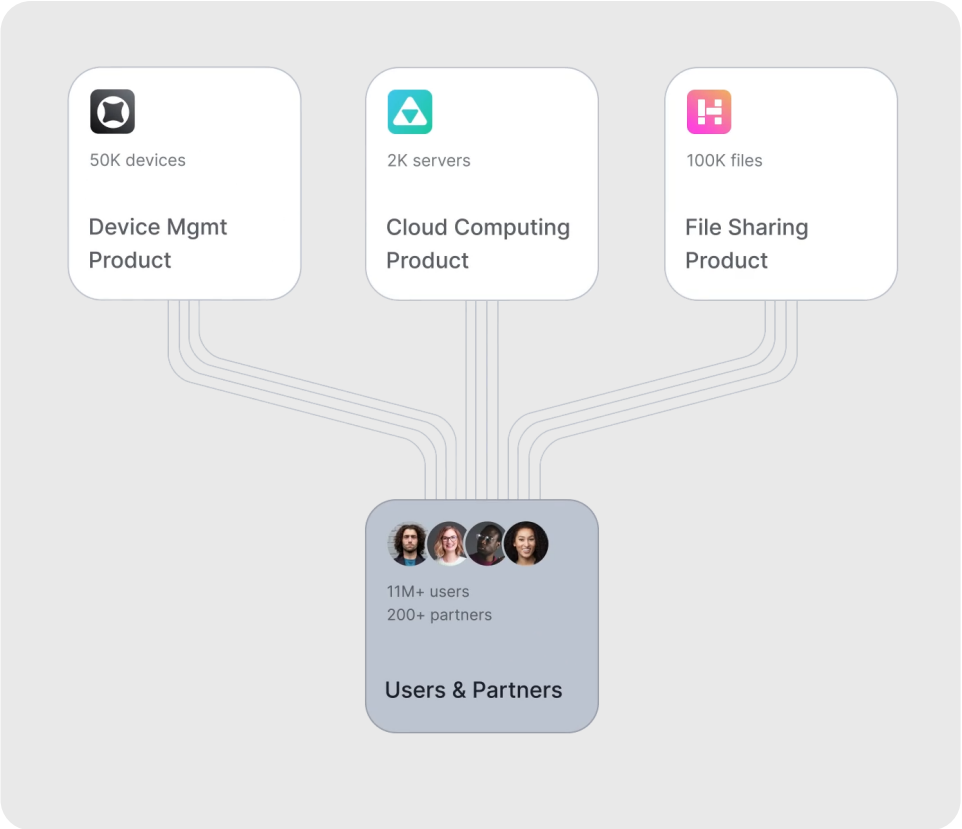
ReBAC systems offer the dual advantages that both authorization logic and authorization data are centralized in the ReBAC database; however, data synchronization processes must be in place to write the data to the ReBAC database.

Fine-grained authorization (FGA)

Fine-grained authorization implies the ability to grant specific users permission to perform certain actions in specific resources. The actions themselves can be tremendously precise; for example, a user could be assigned permission to:

- View all records
- View a specific record
- View only a subset of fields within a specific record

Well-designed FGA systems allow you to manage permissions for millions of objects and users — even permissions that can change rapidly as a system continually adds objects and updates access permissions for its users.



FGA in practice: Zanzibar, Auth0 FGA, and OpenFGA

Google released the Zanzibar whitepaper in 2019, describing the platform that handles creating, storing, and checking trillions of user permissions across all of Google's applications (e.g., Calendar, Cloud, Drive, Maps, Photos, and Youtube) and laying the groundwork for a globally consistent and uniform access control service.

Based on ReBAC, Zanzibar uses object-relation-user tuples to store relationship data, then checks those relations for a match between a user and an object.

Consider the example of Google Drive: access can be granted either to documents or to folders, as well as to individual users or users as a group, and access rights regularly change as new documents are created and shared with specific users or groups.

> ReBAC systems based on Zanzibar store the data necessary to make authorization decisions in a centralized database, and applications only need to call an API to make authorization decisions.

As Google's whitepaper only described the theory behind Zanzibar, many proprietary implementations and open source alternatives have emerged.

Designed by Okta and donated to the Cloud Native Computing Foundation (CNCF) — of which Okta is a key maintainer — OpenFGA is a Zanzibar-based open-source authorization solution that allows developers to build granular access control using an easy-to-read modeling language and friendly APIs.

However, organizations that choose OpenFGA must also take on a long list of operational management responsibilities (see table, below).

In contrast, Auth0 FGA uses OpenFGA as the authorization engine, but takes care of the complexities of hosting, scaling, etc. — so customers can focus on implementing authorization, rather than building and maintaining infrastructure.

Feature	OpenFGA	Auth0 FGA
Availability	Organizations are responsible for availability. OpenFGA currently supports Postgres and MySQL, which must be failed over to another replica in a data emergency.	Deployed in two cloud regions per jurisdiction (US/ Australia/Europe) and uses a database configured with Active-Active replication, to be able to survive a regional AWS failure.
Scalability	Organizations must run their own performance and load testing.	Auth0 FGA has validated this with 1M RPS and 100 billion relationship tuples.
Cloud security	Organizations are responsible for securing the cloud perimeter.	Okta is responsible for securing the cloud perimeter.
Database migrations	Organizations must run their own database migrations, which can lead to downtime.	Okta runs database migrations with no downtime.
Backups	Organizations must run their own database backups.	Auth0 FGA database supports point-in-time recovery and is backed up frequently.
Security patches	Organizations must update their OpenFGA version.	Okta updates OpenFGA with the latest security patches.
Monitoring	Organizations must monitor the uptime/latency and handle production issues.	Okta monitors uptime and latency and is responsible for resolving production issues with the product.
Status page	Organizations must manage their own OpenFGA communications.	Okta provides a status page to monitor availability.
Support	No support is provided.	Okta provides enterprise support with Technical Account Managers, 24x7 pager support, Premier support options, and SLAs, in accordance with the customer's support level.
Dashboard	No dashboard is available.	Okta offers an SSO-enabled dashboard, where multiple users can collaborate on FGA stores and models, and where admins/developers can manage API keys.
Cloud infrastructure provisioning	Organizations must manage the cloud infrastructure.	Okta provisions and manages the cloud services required to run Auth0 FGA.
Autoscaling	Organizations must configure their own autoscaling policies.	Okta configures services and databases to autoscale.
Disaster recovery	Organizations must implement their own disaster recovery processes.	Okta has disaster recovery processes in place for Auth0 FGA.
Data residency	Organizations must ensure compliance with data residency laws.	Okta supports compliance with each country's data residency laws, including our own services and those of our subprocessors.

Key Auth0 FGA concepts and terminology

The Auth0 FGA service answers authorization checks by determining whether a relationship exists between an object and a user.

Each check request references the authorization model against all the known object relationships and returns an authorization decision (i.e., true or false).

To learn more...

Please visit the Fine-Grained Authorization (FGA) Concepts documentation page for additional details including illustrative snippets.

Authorization model	An **authorization model** combines one or more **type** definitions. This is used to define the permission model of a system. Together with **relationship tuples**, the **authorization model** determines whether a **relationship** exists between a **user** and an **object.**
Store	A **store** is an Auth0 FGA entity used to organize authorization check data. Each **store** contains one or more versions of an **authorization model** and can contain various **relationship tuples.**
Relationship tuple	A **relationship tuple** is a base tuple/triplet consisting of a: • **User**: Expressed as a combination of a **type**, an identifier, and an optional **relation**, a **user** is an entity in the system that can be related to an object. • **Relation**: A **relation** is a string defined in the **type** definition of an **authorization model**. **Relations** define a possible **relationship** between an **object** (of the same **type** as the **type definition**) and a **user** in the system. • **Object**: Expressed as a combination of a **type** and an identifier, an **object** represents an entity in the system. **Users' relationships** to it are defined by **relationship tuples** and the **authorization model**. Tuples may add an optional condition, like **conditional relationship tuples**. Relationship tuples are written and stored in Auth0 FGA.
Check request	A **check request** is a call to the Auth0 FGA check endpoint, returning whether the **user** has a certain **relationship** with an **object**.

Auth0 FGA Modeling Language

Auth0 FGA's Domain-Specific Language (DSL) builds a representation of a system's authorization model, which informs Auth0 FGA's API on the object types in the system and how they relate to each other.

To learn more...

For more information — including detailed examples and code snippets — please visit the Modeling Language documentation page.

DSL JSON

```
model
    schema 1.1

type user

type domain
    relations
        define member: [user]

type folder
    relations
        define can_share: writer
        define owner: [user, domain#member] or owner from parent_folder
        define parent_folder: [folder]
        define viewer: [user, domain#member] or writer or viewer from parent_folder
        define writer: [user, domain#member] or owner or writer from parent_folder

type document
    relations
        define can_share: writer
        define owner: [user, domain#member] or owner from parent_folder
        define parent_folder: [folder]
        define viewer: [user, domain#member] or writer or viewer from parent_folder
        define writer: [user, domain#member] or owner or writer from parent_folder
```

JSON syntax is used to call API directly or through the SDKs, while DSL is used to interact with Auth0 FGA in the Playground, dashboard, or CLI.

In the DSL example above:

- The authorization model describes four types of objects: user, domain, folder and document.
- The domain type definition has a single relation called member that only allows direct relationships.
- The folder and document type definitions each have five relations: parent_folder, owner, writer, viewer and can_share.

Getting started with Auth0 FGA

The online documentation details a three-step process to help developers get started with Auth0 FGA. The subsections introduce and summarize these steps, but are in no way a substitute for the online resources, which include extensive explanations and code snippets.

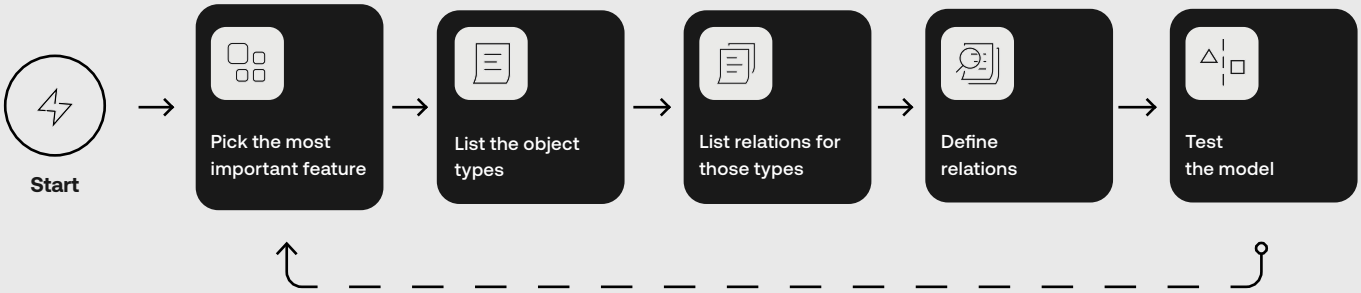
Define your authorization model

Defining an authorization model requires codifying an answer to the question, "Why could user U perform an action A on an object O?" for all use cases or actions in your system.

To learn more...

For more information — including detailed examples and code snippets — please visit the Modeling Language documentation page.

While this may seem intimidating at first, following the iterative process outlined below will help you quickly and completely fulfill your needs.



Step 1 - Pick the most important feature

A feature, in the context of this document, is an action or related set of actions your users can perform in your system.

We recommend starting with the most important feature, because you're probably more familiar with the authorization requirements for this feature than for less important ones.

Once you've picked a feature, describe its authorization related scope using simple language, for example:

- “A user can create a document in a drive if they are the owner of the drive.”
- “A user can share a document with another user or an organization as either editor or viewer if they are an owner or editor of a document or if they are an owner of the folder/drive that is the parent of the document.”

Tip: Avoid using the word "roles", as this ties you to an RBAC way of thinking.

Step 2 - List your object types

Next make a list of the types of objects in your system. For example:

- User
- Document
- Folder
- Organization
- Drive

Tip: You might be able to identify the objects in your system from your existing domain/database model.

Step 3 - List relations for those types

Each of the types you defined previously has a set of relations. To identify relations, we can perform an exercise similar to the previous step. Relations for a type {type} will be all of these:

- Any noun that is the {noun} of a "{noun} of a/an/the {type}" expression. **Tip**: These are typically the Foreign Keys in a database.
- Any verb or action that is the {action} of a "can {action} (in) a/an {type}" expression; these are typically the permissions for a type.

Continuing the example, the resulting lists of relations for each type look like:

Document

- parent
- owner
- editor
- viewer
- can_share
- can_write
- can_view
- can_change_owner

Folder

- owner
- viewer
- parent
- can_create_folder
- can_view
- can_create_ document

Organization

- member

Drive

- owner
- can_create_folder
- can_create_ document

Step 4 - Define relations

Next, you'll create a relation definition for each of the relations you identified. This stage encodes the answers to the question we asked at the beginning of this process: "Why could user U perform an action A on an object O?".

For example, combining the type definitions for document and organization, we have:

```
model
    schema 1.1

type user

type organization
    relations
        define member: [user, organization#member]

type document
    relations
        define owner: [user, organization#member]
        define editor: [user, organization#member]
        define viewer: [user, organization#member]
        define parent: [folder]
        define can_share: owner or editor or owner from parent
        define can_view: viewer or editor or owner or viewer from parent or editor
        from parent or ow
        define can_write: editor or owner or owner from parent
        define can_change_owner: owner
```

Step 5 - Test your model

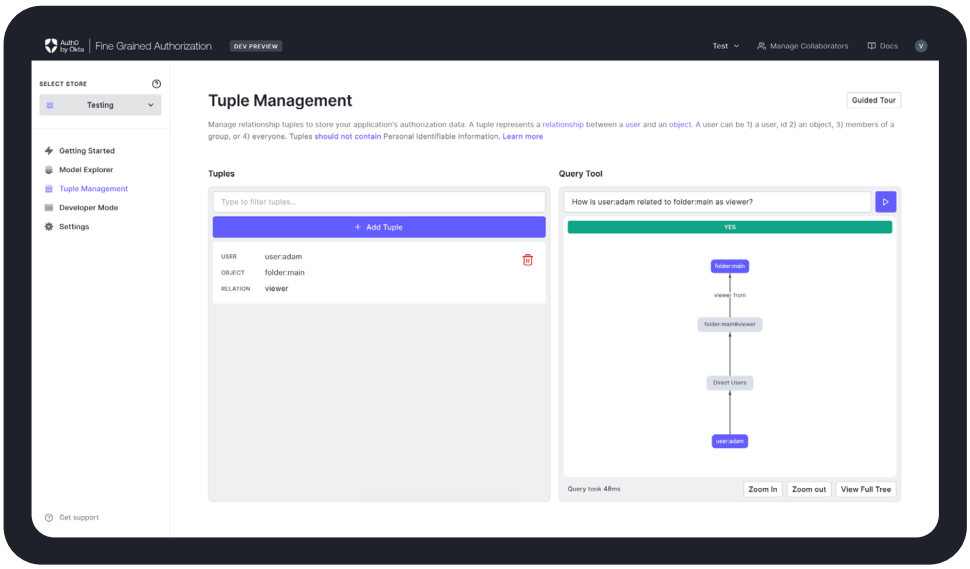
Once you've defined your organizational hierarchies as types, and the most important type for your feature, you want to ensure everything is working as expected.

This is akin to answering the question, "Can user U perform an action A on an object O?" in a variety of formulations that correspond to specific test cases.

To answer these questions, the Auth0 FGA service checks if a user has a particular relationship to an object, based on your authorization model and relationship tuples.

What you want is to ensure that, given your current authorization model and some sample relationship tuples, you get the expected results for those questions.

The Tuple Management Tool within the Auth0 FGA Dashboard provides visualization on any relationships between user and object.



Step 6 - Iterate

To complete your authorization model, we recommend starting small and writing tests to make sure your models perform the way you expect.

Then, once you've gotten a feel for things, iterate through your remaining features.

Write your authorization data

With your authorization model defined, you now need to programmatically write authorization-related data to Auth0 FGA.

To learn more...

Much greater detail is available in the Write Your Authorization Data documentation page.

The table below summarizes the main ways of doing so.

Action	What it does	When to use
Manage user access	Grants (or removes) a user's access to a particular object	Granting access with a relationship tuple is a core part of Auth0 FGA (without any relationship tuples, any check will fail). You should use: • authorization model to represent what relations are possible between the users and objects in your system • relationship tuples to represent the facts about the relationships between users and objects in your system
Manage group access	Grants (or removes) a group of users' access to a particular object	Adding a relationship tuple specifying that a group has a relation to an object is helpful in cases where you want to encompass a set of users with the same relation to an object. For example: • Grant a group of engineers viewer access to roadmap.doc • Create a block_list of members who can't access a document • Sharing a document with a team
Manage group membership	Update a user's membership within a group by adding and removing them from it	Group membership can be helpful as you do not need to iterate over all of the group's resources to add or revoke access to particular objects. You can add a relationship tuple indicating that a user belongs to a group, or delete a tuple to indicate that a user is no longer part of the group. For example: • An employee is hired at a company and thus gains access to all of the company's resources • An employee quits and thus loses access to all of the company's resources
Manage relationships between objects	Grant (or removes) a user's access to a particular object through a relationship with another object	Giving user access through a relationship with another object is helpful because it allows scaling as the number of objects grows. For example: • organization that owns many repos • team that administers many documents
Transactional writes	Updates multiple relationship tuples in a single transaction	Updating multiple relationship tuples is useful to keep system state consistent.

Add authorization to your API

With your authorization model defined, and knowing how to write authorization data to Auth0 FGA, the final step is to update your code to start authorizing user requests.

To learn more...

Much greater detail is available in the Add Authorization to Your API documentation page.

Broadly, doing so will involve:

1. Getting your API keys from the dashboard so that your app can call the Auth0 FGA API
2. Installing the SDK client for the language of your choice (Node.js, Go, .NET, Python, Java, CLI)
3. Configuring the SDK client to call Auth0 FGA
4. Updating relationships tuples by programmatically writing authorization data to an Auth0 FGA store
5. Performing a check against an Auth0 FGA store to validate functionality (e.g., to determine whether a user has a certain relationship with an object)
6. Performing a List Objects request against an Auth0 FGA store to determine all the objects of a given type a user has a specified relationship with
7. Integrating within a framework, such as Fastify or Fiber

Conclusion

The need for more granular and flexible access controls — without sacrificing security or compliance — is clear.

However, reaching this state is easier said than done, and the subjects of authorization and access control can be challenging, especially considering that Identity and Access Management is just one of many functions that app developers need to implement.

To ensure your authorization function delivers for you and your users, both today and well into the future, look for a fine-grained authorization solution that provides:

- **Low latency**: Every millisecond you spend in authorization will impact your application's overall latency.
- **Availability and reliability**: Authorization is mission critical and any downtime can disrupt user access in your application.
- **Scalability**: Your authorization solution should be capable of supporting your business as your user base grows — potentially even accommodating sudden and massive spikes.
- **Policy flexibility:** Support for different kinds of policies like RBAC or ABAC give you the ability to choose the best policy for your app, and to update policies as your needs evolve.

Okta can help

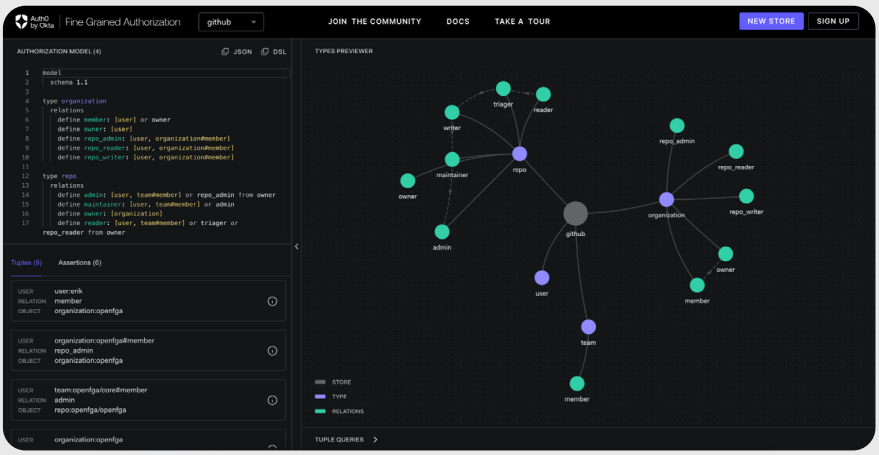
Auth0 FGA delivers authorization at scale and gives businesses the power to simplify access control across multiple applications, parameters, and users. It's authorization as a service, allowing developers to design and implement permissions in a way that's flexible, scalable, and easy to use.

- **Manage authorization at scale, without complexity.** With Auth0 FGA, developers can update and manage authorization policies from one centralized location without touching application code. Scale access controls as your product and user base grows.
- **Get as granular as needed, with absolute control.** Easily manage groups, teams, organizations, or any set of users, and assign them permissions on any resource or groups of resources. Developers can define access down to the finest detail, allowing greater security and compliance.
- **Be flexible with permissions and access.** With Auth0 FGA, app owners and developers have greater flexibility in defining how users grant permissions and access. Their users can then create authorization rules around multiple parameters, beyond just roles.
- **Reduce Latency.** Auth0 FGA is designed to provide high scalability while minimizing latency, by routing requests to the closest server and responding to authorization queries very quickly. Users can move at the speed of their business.
- **Save development time and resources.** Auth0 FGA saves development time and resources by making it easier to build and scale software with sophisticated authorization capabilities built in. It seamlessly integrates with a business' existing systems using developer-friendly tools like APIs, SDKs, CLIs and IDE integrations.
- **Backed by the Okta brand**. We pioneered trusted authorization, now we're pioneering FGA.

Additional developer resources

The latter sections of this document draw heavily upon the Auth0 FGA documentation, but what's summarized herein is a tiny fraction of that larger resource. For example, the online documentation includes extensive code/syntax examples that are both too deep and too extensive for this Technical Primer.

Closely related is the FGA Playground — an interactive environment that allows developers to get hands-on with Auth0 FGA and to put into practice the examples and lessons from the online documentation.



Finally, we also encourage developers to explore:

- fga.dev, which is the go-to starting point for all things Auth0 FGA
- Zanzibar Academy, which provides an assortment of resources explaining Google's Zanzibar — the ReBAC model underlying both Auth0 FGA and OpenFGA