Introducing the Identity Actor Model and Renaming Architecture Components for Better Clarity

nicola-gallo

Published in ztauth

7 min read · Just now

Listen Share More

Chapter 4

In the previous chapter, I introduced the **ZTAuth* Architecture**. In this chapter, I will focus on explaining the **Identity Actor Model** and renaming the architecture components to improve clarity and alignment.

Before starting, I want to highlight that this is a work of ideation and designing new concepts that are not yet well-known. This means we are on a journey of discovery, and in each chapter, besides adding new ideas, patterns, and design elements where needed, it is possible to revisit some earlier concepts and rethink them for better clarity.

Let's start discussing identities in terms of users, roles, and user groups.

Let's explore what they are:

- An **user** is a **unique identity** in a system. It can represent a human or a machine and is often associated with authentication credentials, such as a username and password.
- A **role**, on the other hand, is a **set of permissions** or **privileges** that defines what a user can do. Roles are assigned to users to simplify permission management, but they do not represent unique entities.
- A **group** is a collection of users who share common characteristics, such as belonging to the same department or project. Groups can have roles assigned to them, and all members inherit the related permissions.

Finally, we can say a user is a single identity, a role defines what they are allowed to do, and a group is a way to manage multiple users together.

Regarding **users**, we can identify two types:

- **Human:** A person who authenticates using a flow designed for individuals, such as entering a username and password or using multi-factor authentication.
- **Machine:** A service account that authenticates using a machine-to-machine flow, typically involving API keys or certificates.

> *In both cases, we can confidently state that they are both types of users. These users should be managed by an external Identity Provider (IdP). In our zero trust architecture, it is our responsibility to define and manage their roles and permissions. For authentication, we only need to synchronize these identities with our system.*

Roles, on the other hand, are what we have defined as **Identity Permissions**. Therefore, including the concept of roles in the architecture appears redundant. Although many Identity Providers currently use roles to define permissions, in the type of architecture we are building, roles seem to fall into a gray area. Grouping permissions lies somewhere between the responsibilities of the Authentication (AuthN) and Authorization (AuthZ) domains.

> *For this reason, we will remove the concept of roles from the architecture for now. However, in the future, we will revisit this topic to determine the best way to manage it.*

Finally, we have **groups**. I would say there's no need for further discussion on this topic, as managing groups should be the responsibility of the Identity Provider. Groups do not need to be part of our zero trust architecture.

> *In the future, we might decide to synchronize groups to gather their metadata for specific purposes. However, to keep things simple and avoid overengineering upfront, we will not synchronize them until we identify a clear and valuable use case.*

At this point, you might be thinking: "**Okay, we removed roles, but how are we going to implement the *Trusted Delegation* and *Trusted Elevation* scenarios discussed in the previous chapter?**"

This is a valid question. Indeed, we still need an identity model to implement these scenarios. However, roles are not the right solution for this purpose. Roles are primarily a way to group permissions and assign them to users, whereas delegation and elevation address entirely different concerns.

When working on this scenario, the question I asked myself was: "**How do I create an authorization context that can be associated with multiple identities?**"

> For instance, consider the API-to-Kafka use case: a user, Bob, would elevate to an authorization context to push a message to Kafka. The worker, authenticated with a service account, would then also need to elevate to the same authorization context.

This **highlights** that the **authorization context** is **not strictly tied** to the identity managed by an **Identity Provider**. Instead, it represents a shared authroiztion context that spans across different types of identities, enabling coordinated actions under a unified authorization framework.

Then I began **thinking in terms** of **Autonomous and Disconnected Challenges.** For instance, consider a **robot** equipped with **sensors** to receive **input**. It is authenticated using a **service account** that uniquely identifies that specific robot. Additionally, it is equipped with an **AI model** to act on **behalf of user** requests around it. This means it needs to handle data and actions for multiple identities, both human and machine. Clearly, it requires an authorization context to operate effectively.

Let’s take this further and abstract the scenario. Let’s set aside the specifics of “**who**” is acting — whether it’s a robot authenticated as a service account — and instead focus on what is performing the work. For instance, let’s define it as an AI agent. The concept here is that this agent operates as a kind of virtual identity, acting on behalf of other identities. This model introduces the notion of an **abstract identity** that exists not as a principal in the authentication model but solely within the authorization model.

This idea extends well to scenarios like **digital twins**, where a **virtual representation** acts independently but **reflects** the **identity** it represents. This leads us to the concept of an Identity **Actor**, which is an identity that shoud not be managed by an Identity Provider.

This approach allows for flexibility in designing systems that require **autonomous**, **shared**, or **context-driven authorization** while decoupling the strict identity management provided by an Identity Provider.

> *In this article, we are defining the concept of an Actor. We will explore the details in the upcoming chapters.*

Let’s define the concept of an Actor in a more structured way.

An **Actor** is a type of **virtual identity t**hat can be **temporarily assumed by a Principal** (e.g., a user or an entity represented by an Identity Provider — IdP). **Actors** are designed to **encapsulate specific** and **isolated contexts**, allowing **limited** and **purpose-driven permissions** for particular tasks. They can be configured for the following scenarios:

- **Role-Based Actor:** represents a predefined role with specific permissions for a given task or function, such as “Approvals Manager” or “Compliance Reviewer.”

> ***Example:*** *A Principal with the “Finance Specialist” role temporarily elevates their permissions to “Approvals Manager” to approve budget requests without gaining broader permissions.*

- **Digital Twin Actor:** a virtual representation of a Principal or Service Account that operates independently. A Digital Twin can assume its own virtual identity to perform specific tasks, such as scheduled processes or API operations, reflecting the capabilities of the original Principal.

> *Example: An application uses a Digital Twin configured by the IdP to represent an "Agent" performing periodic compliance checks across multiple tenants.*

Key Features and Benefits of the Actor Model:

- **Zero Trust Security:** Temporary elevation to an Actor enables the Principal to operate with the least privileged set of permissions necessary to complete a specific task. This approach minimizes risks, prevents unauthorized privilege escalation, and ensures full auditability of operations.

> *Example: A user elevates to the "Compliance Reviewer" role to access sensitive reports. Once the task is completed, the elevated permissions are automatically revoked.*

- **Role Isolation:** Each Actor is strictly confined to its assigned role or context. This isolation guarantees a clear separation of responsibilities and limits access to resources outside the scope of the active role.

> *Example: A Developer Actor cannot access production operations, while a "Production Deployer" Actor is restricted to deployments only.*

- **Future Federation:** The Actor model lays the groundwork for integrating permissions and identities across federated environments. Well-defined roles and permissions allow secure collaboration between systems managed by different IdPs while maintaining security boundaries

> *Example: An organization shares a Digital Twin Actor with a partner to access a subset of sensitive data without exposing the full identity of the original Principal.*

Advantages for Distributed and Multi-Organization Systems

- **Security:** Reduces the attack surface through least privilege and controlled elevation.

- **Flexibility:** Supports dynamic roles, digital twins, and integrations with Identity Providers.

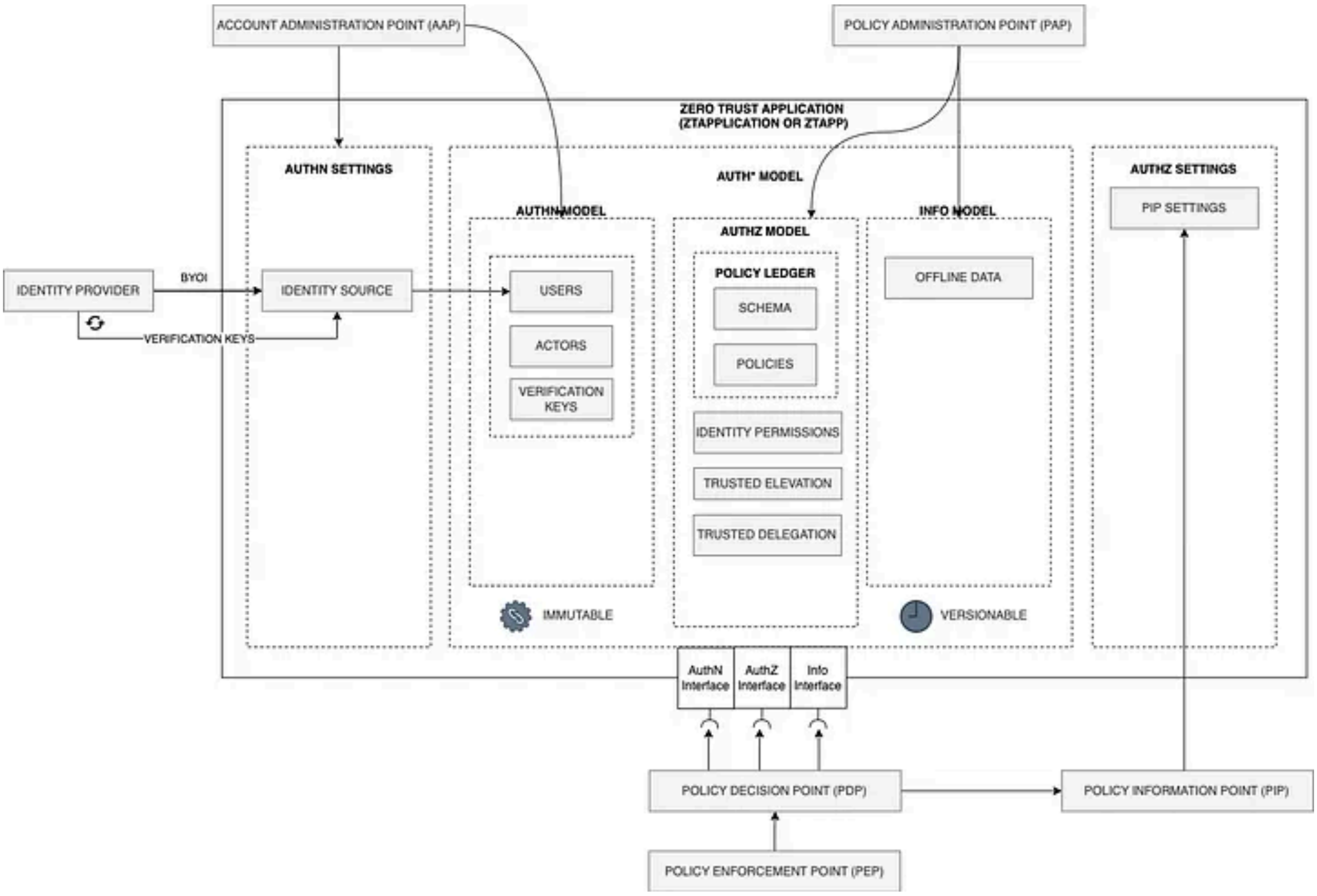
- **Scalability:** Suitable for complex and multi-tenant systems, with centralized management of granular permissions.

At this point, we have completed the definition of the **Actor model**, and it is time to update the **ZTAuth* Architecture**. For clarity, I noticed two concepts that might confuse people: ***ZTAccount*** and ***Repositories***, as their current names may not fully reflect their purposes. Therefore, I propose updating these names:

- **Zero Trust Account (ZTAccount):** I suggest renaming this to Zero Trust Application (**ZTApplication** or **ZTApp**), as it better reflects its purpose and is more understandable in the context of the architecture.
- **Repository:** I propose changing this to **Policy Ledger**, as this term better represents its functionality. It is an immutable, versionable structure that deals with policies, and the new name aligns with its role in the architecture.

Apart from these two changes, I also propose removing the explicit concept of Permissions. Instead, we can assume that Identity Permissions will directly map identities to their policies. After all, permissions are essentially policy-driven, so this change simplifies the architecture while keeping its functionality intact.

Below is the updated **ZTAuth* Architecture,** incorporating the proposed changes.



ZERO TRUST APPLICATION (ZTAPPLICATION or ZTAPP)

With this, we conclude this chapter. Stay tuned for the next one!

Zero Trust Architecture Security Actor Model Policy As Code

ZTAuth*

Following

Published in ztauth

4 Followers · Last published just now

ZTAuth*: Redefining AuthN, AuthZ, and Trusted Delegation with Transferable, Immutable, and Resilient Models for a Zero Trust World

Edit profile

Written by nicola-gallo

9 Followers · 4 Following

A tech entrepreneur specializing in cloud and enterprise strategic technology, with a focus on distributed systems.

No responses yet

•••

What are your thoughts?

Respond

More from nicola-gallo and ztauth