# Resources, Actions and Accounts in the context of Autonomous and Disconnected Challenges

nicola-gallo
Published in ztauth
19 min read · Nov 28, 2024



Chapter 1


In the introduction , we presented the main ideas behind **ZTAuth*.** This chapter begins to define three key elements — ***Resources***, ***Actions*** and ***Accounts*** — and explores their role in addressing the *challenges of autonomous and disconnected* systems. These foundational concepts are essential for understanding how **ZTAuth*** operates and will provide the necessary context for the topics covered in the following chapters.

> *Before we begin, I want to highlight that part of the content covered in this chapter simplifies and explains concepts from my contribution to the* ***Nitro Agility****'s paper,* ***"Multi-Account and Multi-Tenant Policy-Based Access Control (PBAC) Approach for Distributed Systems Augmented with Risk Scores Generation."*** *The full paper is available* here *and is licensed under CC BY-NC-ND 4.0.*

Zero Trust introduces three core principles which are important to keep in mind:

- **Never trust, always verify:** Never trust implicitly; always verify the identity and context of users, devices, and applications before granting access.
- **Least privilege access:** Grant the minimum level of access necessary for a task, ensuring users or systems only interact with the resources they truly need.
- **Assume breach:** Operate under the assumption that a breach could occur at any time, designing systems to contain potential damage and prevent lateral movement.

> *Zero Trust, introduced by John Kindervag from Forrester in 2010, redefines security by requiring continuous verification and rejecting implicit trust. This chapter will not cover the foundations or history of Zero Trust, as these topics are beyond its scope. For those interested, there are many detailed papers and authors who explore this subject extensively. Instead, I will focus on how to apply Zero Trust principles across the software ecosystem. While most discussions concentrate on network devices and infrastructure, shifting the focus to software systems provides new and valuable insights..*

Looking at these principles from a software perspective, we aim to understand how they can be interpreted and applied across different systems and contexts:

- **Never trust, always verify:** Never trust any interaction implicitly. Always verify the identity and context of users, services, and software processes before granting access or executing operations within the software environment.
- **Least privilege access:** Apply the principle of least privilege to software flows, ensuring that users, services, and processes can access only the specific resources and actions required for their role. Access privileges should always be contextual, dynamically adapting to the current state of the system and the identity of the requester at the moment of the interaction. Privileges must also be time-bound, ensuring they expire or are revoked as soon as they are no longer needed.
- **Assume breach:** Design software systems with the assumption that a breach can occur at any time. Implement safeguards to limit potential damage and prevent unauthorized movement between components or workflows. Ensure strong segmentation and employ continuous monitoring to quickly detect and isolate suspicious activities. Additionally, implement robust auditing mechanisms to

track and verify the true identity behind every action, even in cases of delegation or impersonation, ensuring accountability and transparency.

From a software perspective, *security revolves around controlling access to and manipulation of information*, which can reside in various resources such as *storage systems, disks, APIs, event streams, etc*. These elements, collectively referred to as **resources**, require safeguarding to ensure their **integrity** and **confidentiality**. On these resources, specific **actions** can be performed — such as *reading data, writing* or *updating information*, or *triggering operations through an API* — making it critical to regulate and monitor access to prevent unauthorized alterations or misuse.

The technologies and interactions involved in performing these actions — such as accessing the file system, invoking APIs, or processing events — introduce an additional layer of **access control**, often managed at the **infrastructure level**. However, within software systems, access must be governed by ***strict policies tied to the requester's identity, role, and permissions***. This ensures that access is both appropriate and secure. As a result, security in software systems focuses on two main components:

- **Resources**: Identifying and defining what needs to be protected.
- **Actions**: Controlling and managing which actions can be performed on these resources, and ensuring these actions are authorized based on the requester's identity.

This layered approach ensures that both the infrastructure and the software systems work together to provide comprehensive security.

In software systems, a resource represents elements such as data, events, services, or other entities that actions can target. Securing the system requires more than just implementing access controls — it requires ***conceptualizing***, ***contextualizing***, ***designing***, and ***representing resources*** and ***actions*** in a structured, *machine-readable* format. This allows the system to understand their relationships and enforce precise security measures. The goal is to ensure that ***only authorized identities can access specific resources or perform specific actions***, safeguarding the system and its operations.

This **authorization model** must be carefully designed as part of the broader system architecture. Its design and representation follow the same principles used in

**modeling business domains.**

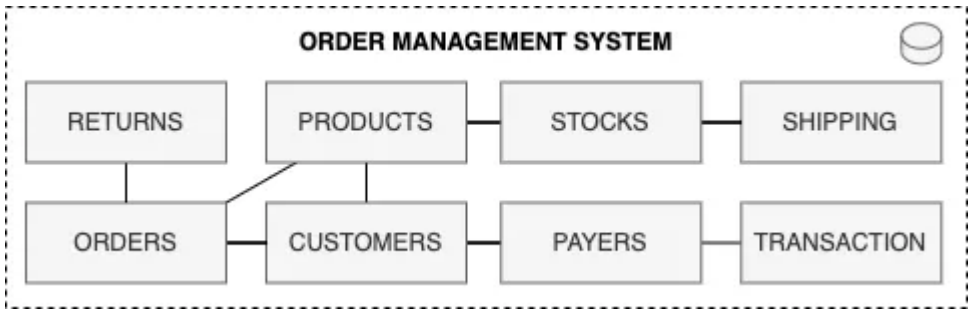
But wait — resources, actions and business domains? This may sound familiar. When abstracted to map resources to entities, these concepts align closely with the principles of **Domain-Driven Design (DDD).** Traditionally used to model *complex business domains,* DDD can also be applied to software security. By *modeling* ***resources*** *and* ***actions*** within the domain, DDD offers a structured framework to integrate security requirements directly into the system's design. This approach makes access controls, authorization rules, and security policies part of the core logic, embedding security into the architecture itself.

> *Once again, I will not explain what Domain-Driven Design is, as there are plenty of resources available on this topic. This concept, introduced by Eric Evans, could even be a book on its own — in fact, there are many books about it.*

To make this clearer, let's imagine we are developing a software application to manage **Inventory**, **Orders**, and **Payments.**

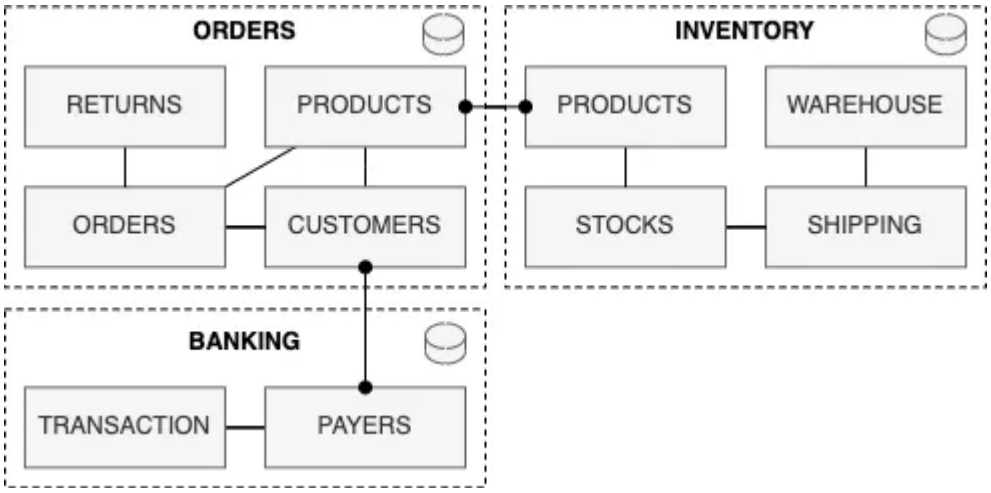
> ***Note:*** *In this explanation, we are focusing on the approach. To keep things clear and straightforward, I will analyze only a small subset of entities — the most relevant ones — to illustrate the concept. This simplification avoids unnecessary complexity, as analyzing every single entity would add little value to understanding the core idea.*

We could choose to implement it as a large monolithic software application, using a single database to store all the data.



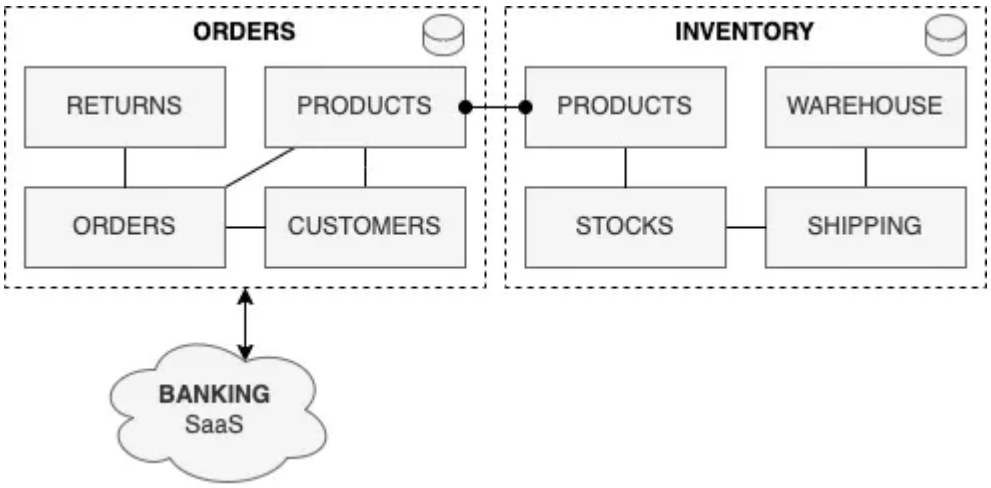

Architecture 1: Monolithic Order Management System

Alternatively, we could choose to implement it using **microservices** and distributed across multiple **databases.**




Architecture 2: Microservices Order Management System

As another option, we could take it a step further by implementing **Inventory** and **Orders** as *microservices* while leveraging a **SaaS** solution for **Payments.**




Architecture 3: Microservices Order Management System with SaaS Integration

Looking at these *three architectures,* it becomes evident that the same data can be distributed across different databases and partitioned among various systems or software applications. Furthermore, the communication and data flow may take different routes, rely on various technologies, and traverse distinct networks. This highlights the importance of recognizing that the same entities can exist in different contexts, which means it is not enough to identify an entity; its context must also be considered. Proper contextualization ensures that actions and access are evaluated based on the specific circumstances in which the entity operates.

Here, things start to get interesting as we need to explore and understand a few key concepts:

- How to define and represent Resources and Actions

- How to map Resources and Actions to security policies within the software context
- The role of partitioning in securing Resources and Actions
- Steps to align software security with Zero Trust principles

Let's begin by addressing each point step by step, starting with **How to define and represent Resources and Actions**.

I want to ensure that the core principles of Zero Trust are applied when expressing an authorization model:

- **Never trust, always verify:** The model must always be evaluated, consistent, machine-readable, and verifiable to guarantee that every interaction is properly authenticated and authorized.
- **Least privilege access:** The model must allow me to contextualize, design, and specify access rights clearly, ensuring that permissions are limited to what is strictly necessary.
- **Assume breach:** Assuming a breach, the model must make it easy to identify which resources and actions were targeted. To achieve this, it must be structured, easy to monitor, and auditable.

If we apply Domain-Driven Design (DDD) practices to build an authorization model, it becomes clear that this approach can achieve these goals:

- **Never trust, always verify:** A domain-specific language can represent the authorization model, making it verifiable and understandable within the domain.
- **Least privilege access:** DDD allows us to contextualize and differentiate the meaning of resources and actions across various contexts, ensuring precise and scoped access.
- **Assume breach:** With a static, pre-defined model, it becomes easier to integrate structured auditing and monitoring. This provides clear insights into potential breaches and ensures the system is designed for resilience.

It is clear that ***DDD*** is our ally in this process. To fully leverage its potential, we need to define a domain-specific language and establish how to implement it effectively. The most intuitive approach is to represent this as a ***YAML*** file, which allows us to clearly describe domains, resources, and actions. I would call this an **Authorization Model Schema**. Let's explore how such a schema might look like.

In the context of *Architecture 1, the monolithic one*, we don't have distinct domains; instead, we have a single, broad domain that we need to secure:

```yaml
domains:
  - name: order-management
    description: Operations for managing orders and products
    resources:
      - name: products
        description: Management of product-related operations
        actions:
          - name: view
            description: View product details
          - name: update
            description: Update product information
          - name: delete
            description: Remove a product from the catalog
          - name: add
            description: Add a new product to the catalog
      - name: orders
        description: Management of customer orders
        actions:
          - name: create
            description: Place a new order
          - name: view
            description: View details of an existing order
          - name: cancel
            description: Cancel an existing order
          - name: update-status
            description: Update the status of an order
```

Instead, in the context of *Architecture 1, the microservices-based one*, we have distinct domains, each representing a specific business function or service that can be secured independently.

```yaml
domains:
  - name: order-management
```

```
    description: Operations related to managing orders
    resources:
      - name: order
        actions:
          - name: create
            description: Place a new order for a customer
          - name: view
            description: View details of an existing order
          - name: cancel
            description: Cancel an order
          - name: update-status
            description: Update the status of an order

  - name: inventory-management
    description: Operations related to inventory and stock management
    resources:
      - name: order
        actions:
          - name: view
            description: View orders related to inventory and stock levels
          - name: update
            description: Update the order status based on stock availability
          - name: approve
            description: Approve orders for restocking
```

Finally, let's look at the context of Architecture 3, the microservices-based architecture integrated with a SaaS payment system.

```
domains:
  - name: order-management
    description: Operations related to managing orders
    resources:
      - name: order
        actions:
          - name: create
            description: Place a new order for a customer
          - name: view
            description: View details of an existing order
          - name: cancel
            description: Cancel an order
          - name: update-status
            description: Update the status of an order

  - name: payment
    description: Operations related to processing payments, integrated with an
    resources:
      - name: payment
        actions:
```

```
    - name: initiate
      description: Initiate a payment process through the external SaaS p
    - name: confirm
      description: Confirm payment completion from the external SaaS prov
    - name: refund
      description: Request a refund via the external SaaS payment provide
    - name: status-check
      description: Check payment status through the external SaaS provide
```

By now, it should be clear what we mean by a ***schema***. This YAML file is designed to be **machine-readable** and verifiable, allowing for ***clear contextualization and the enforcement of access limits***.

It's important to note that having a **strict and well-defined authorization model** not only **reduces** the number of **bugs** but also minimizes **vulnerabilities**, as a smaller attack surface is created by coding errors.

Furthermore, **auditing is clearly structured**, **understood**, and can be integrated seamlessly into business **risk management processes** and other **platforms**. Below is an example of a possible audit log format:

```
"timestamp": "2024-11-27T12:34:56Z", "resource": "order", "resource_id": "ORD12
"timestamp": "2024-11-27T13:10:22Z", "resource": "order", "resource_id": "ORD12
"timestamp": "2024-11-27T14:45:10Z", "resource": "order", "resource_id": "ORD12
"timestamp": "2024-11-27T15:00:30Z", "resource": "order", "resource_id": "ORD12
```

> This approach uses configuration languages tied to a central schema, ensuring simplicity and security. Unlike flexible policy languages, it reduces ambiguity, minimizes errors, and streamlines security without extensive testing.

It's no coincidence that **cloud providers** have developed solutions in this direction. Allowing IAM policies in general-purpose languages would create security gaps and require complex testing. By using **configuration languages** tied to a central schema

of resources and actions, they minimize ambiguity, reduce errors, and ensure streamlined, secure operations.

Anyway, let's set this aside for now, as we will discuss these aspects in detail in the upcoming chapters.

Now it's time to move on to **How to map Resources and Actions to security policies within the software context.**

An ***Authorization Model Schema*** is intended to *model* the *resources* and *actions* we want to **permit.** Of course, we need a **configuration language**, and this configuration will also need to express conditions in order to filter permissions.

When evaluating if an identity has permissions on a resource:

- A **RequestContext** must be created, including the identity and the system context needed to evaluate the conditions.
- The **policy engine** will process the Schema, Policies, and RequestContext.
- If the **evaluation** is positive, the operation will be executed.

Below is a hypothetical, simplified example of how this might look in code. Naturally, in a real-world implementation, it would be more abstract and structured, involving a more complex production-level system with various security, policy evaluation, and permission handling layers.

```
def authorize(identity, resource, action, system_context, policies):
    request_context = RequestContext(identity, system_context)
    policy_engine = PolicyEngine()
    if policy_engine.evaluate(request_context, resource, action, policies):
        print("Permission granted, operation can proceed.")
    else:
        print("Permission denied, operation is not allowed.")
```

From my experience working with real-world systems, it's clear that how resources and actions are structured can greatly affect both the performance and maintainability of the system. While it's easy to assume that a simple, tightly coupled approach might work for basic use cases, this approach often leads to

significant challenges as the system grows and scales. Below are the key takeaways based on real-world scenarios.

**Key Takeaways:**

- **Tight Coupling of Resources and Actions:** *Resources* and *actions* are inherently tied to the system design, and only the system architect fully understands their structure. This creates challenges when trying to scale or modify the system.
- **Request Context and External Dependencies:** Extracting necessary information often requires building a request context, which might need access to external systems. This dependency can introduce delays, especially when accessing external services or databases.
- **Performance Impact:** When external systems are involved, high latency or a large number of authorization requests can cause significant performance degradation. This becomes a bottleneck, especially in large-scale systems with frequent authorization checks.
- **1-to-1 Mapping of Resources and Actions:** Directly mapping resources and actions to data resources can lead to a slow and unusable solution. This mapping approach can quickly become impractical, as it doesn't scale well with the growing complexity of real-world systems.
- **Strong Coupling Between Policies and Resources:** Tight coupling between policies and system resources creates a complex environment that's hard to maintain. It slows down development and makes the system harder to evolve as requirements change, leading to significant technical debt.

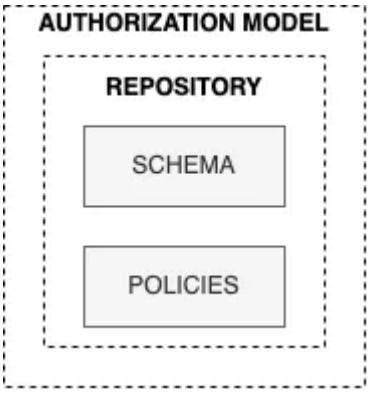
> *These takeaways represent key principles to keep in mind when designing systems that adopt ZTAuth*. However, there are solutions to mitigate these problems and still build high-performing real-world systems based on these concepts. We will explore these solutions in more detail in the following chapters, where we'll better understand how to address these challenges while maintaining scalability and performance.*

Let's now dive into understanding The role of **partitioning** in securing Resources and Actions.

**Partitioning** plays a crucial role in enhancing security by **isolating resources** and **actions** into distinct **segments** or **domains**, *reducing the attack surface* and *limiting the*

*scope of potential breaches*.

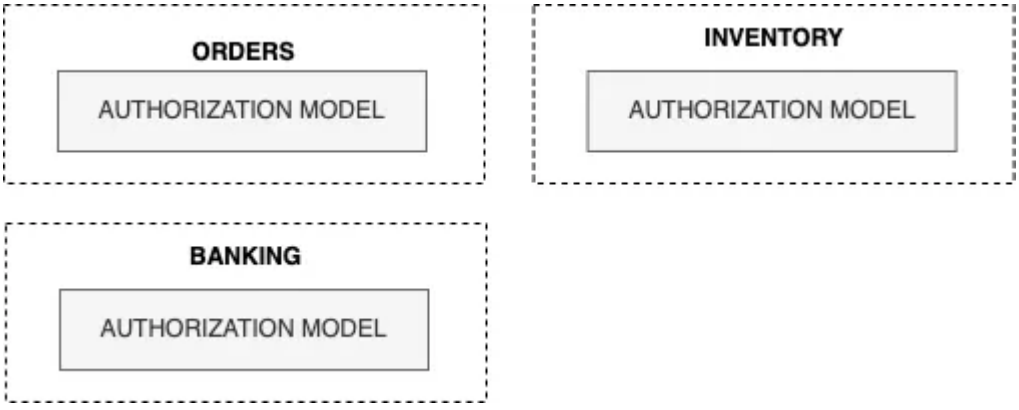
By partitioning, we refer to managing access across different parts of the software system, with each part having its own ***schema*** and ***policies***. Ideally, access to resources and actions is controlled within each **partition**, *reducing risks* and simplifying secure management of the system. This brings us to a new concept: the **Repository**, which contains the **schemas** and **policies** for each **partition**.



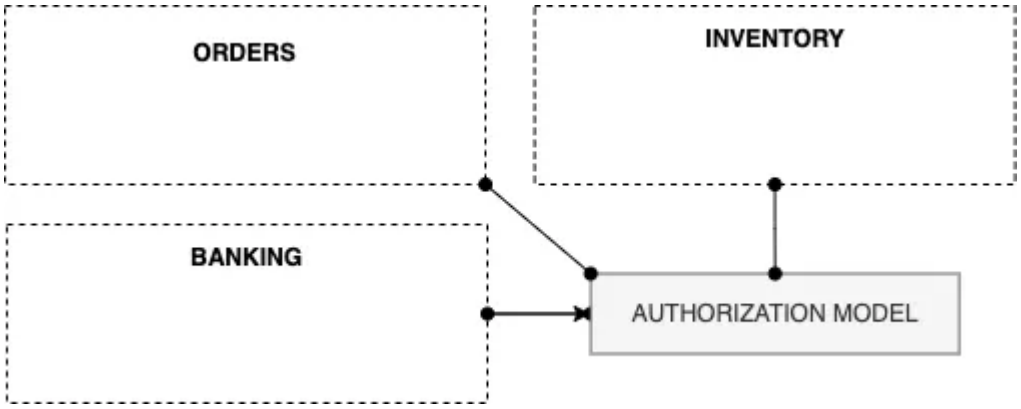

Authorization Model

Working on different software solutions using this approach, I realized that these ***repositories*** need to be **easy to clone**, **capable of coexisting**, and each **version** of the software must be linked to a specific repository. When a new version of the system is developed, a new version of the repository must also be created to ensure that we can roll back the authorization model, which, as you've understood by now, is a critical part of the application.

It is important to point out that the partitioning of the system is in the hands of the software architect. In fact, if we take the microservices architecture as an example, we can see that it can be developed in multiple ways. Here, I will show two options.




Architecture 2: Partitioned Authorization Models




Architecture 2: Single Central Authorization Models

> *Of course, practices may emerge from this approach, but we will explore them further while reviewing the examples in the upcoming chapters.*

We've started exploring the concepts, and by now, you might be wondering why we are focusing only on authorization. If so, you're right, as ***ZTAuth**** works with ***AuthN***, ***AuthZ***, and ***Trusted Delegations***.

Before moving forward, it's useful to clarify where **AuthN** fits into the picture. As the first step, let's first look at the concept of a Zero Trust architecture.

> *To approach this, let's refer to **NIST Special Publication 800–207**, which provides a comprehensive framework for understanding Zero Trust architecture. These guidelines are widely recognized and offer a solid foundation for building secure, scalable systems. I recommend reading them if you haven't already. However, we will interpret these guidelines from a software perspective as we move forward.*

As per the ***NIST*** document, **Zero Trust Access** requires an abstract model where a **Subject** — in our context, an **Identity** — needs to access an enterprise resource. Access is granted through a **Policy Decision Point** (**PDP**), which evaluates the request, and a **Policy Enforcement Point** (**PEP**), which enforces the decision.

In our Authentication Model, the Subject is defined as an ***Identity***, which represents either a **user** or a **role**. We can break this down as follows:

- **Principal:** A Principal is a human user or workload with granted permissions who authenticates and makes requests. Specifically, a user or an assumed role.
- **User:** A User is an identity representing a single person or a Function Identifier (FID) with specific permissions.

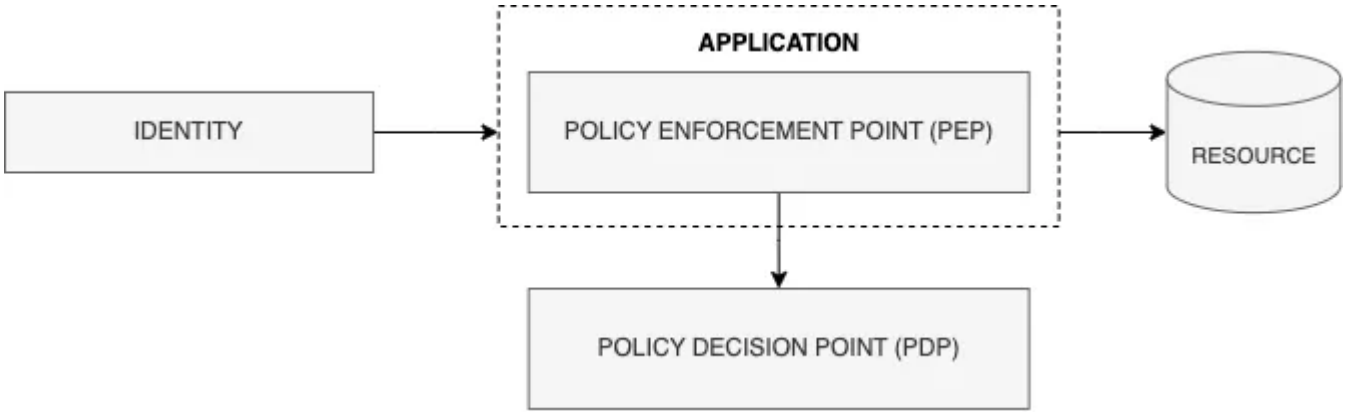
- **Role:** A Role is an identity within an account that has associated permissions. A Role can be temporarily assumed by a Principal identity.

There are many ***established standards*** for ***managing identities,*** and it is not necessary to redefine them here. Instead, we need to *integrate* with these *standards*. In our context, the approach is focused on **Bring Your Own Identity (BYOI)**, which allows our **Authentication Model** to simply connect to an external identity provider and *synchronize identities*.

In our case, we **enrich** those **identities** with additional **metadata** and information that the **identity provider** may not already have. We define the identity provider as the **identity source**, and we may have multiple such sources.

> *Note: Authentication should be implemented according to widely recognized standards, using certified and reliable implementations and solutions.*

To recap, the ***identity*** attempts to access ***resources*** through the ***application***. The application itself acts as the **PEP**, which validates the access request against the ***PDP*** and, if successful, grants access to the resource.



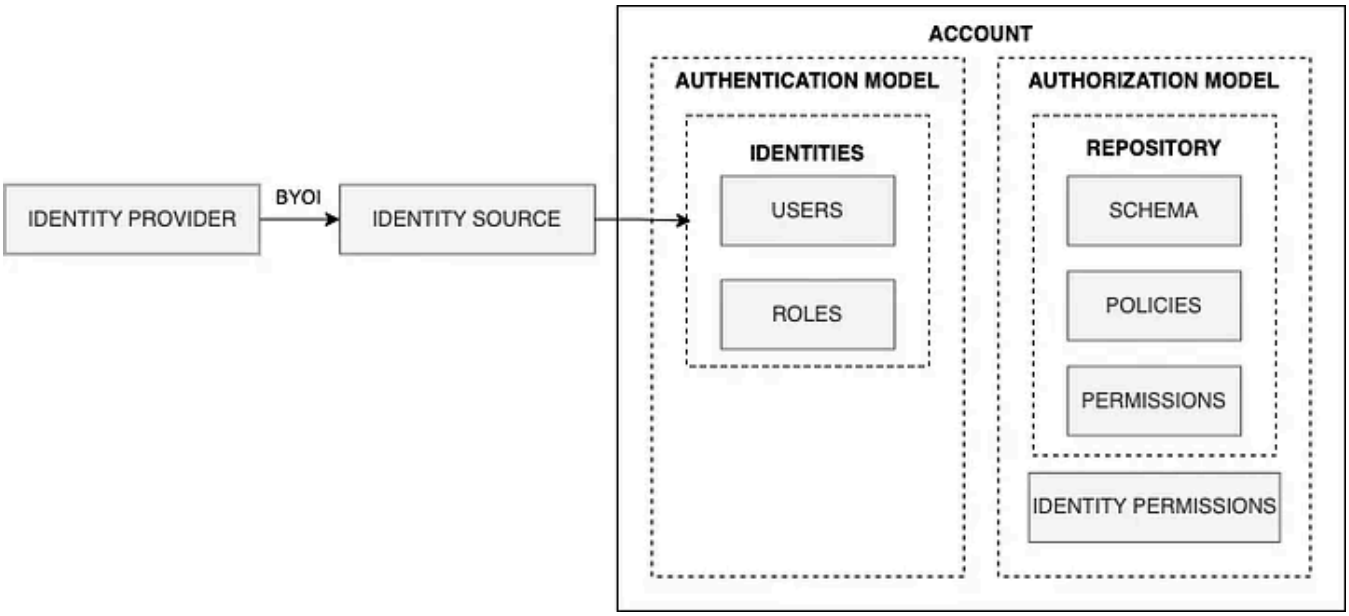

Zero Trust — Policy Enforcement Point (PEP) and Policy Decision Point (PDP)

The **PDP** needs certain elements to evaluate a request: the **identity** of the user or entity making the request, the **resource** being accessed, the **action** being performed, and the **policies** that define what is allowed or denied. But how does the **PDP** actually know about these policies?

Here's where permissions come into play. An **identity** is associated with **permissions**, which act as a logical representation of policies. These permissions define what ***actions are permitted or forbidden*** in the system. At the same time,

policies themselves are the rules that determine what is allowed or not, forming the backbone of the *authorization model*.

**Permissions** and **policies** are stored in the **repository** and are part of the *authorization mode*l. However, identities work differently — they belong to the **Authentication Model** and are not directly linked to the repository.




Account: Authentication Model, Authorization Model and BYOI with Identity Sources

This brings us to the idea of an **Account**. An *account* acts as a container that ties everything together. It includes the ***authentication mode*l**, which manages *identity sources* and *identities*, and the ***authorization model***, which handles repositories and the mapping between identities and permissions. This structure makes it easier to manage both identity and access in a unified way.

Finally, we've reached the point where we can explore the **Steps to align software security with Zero Trust principles**.

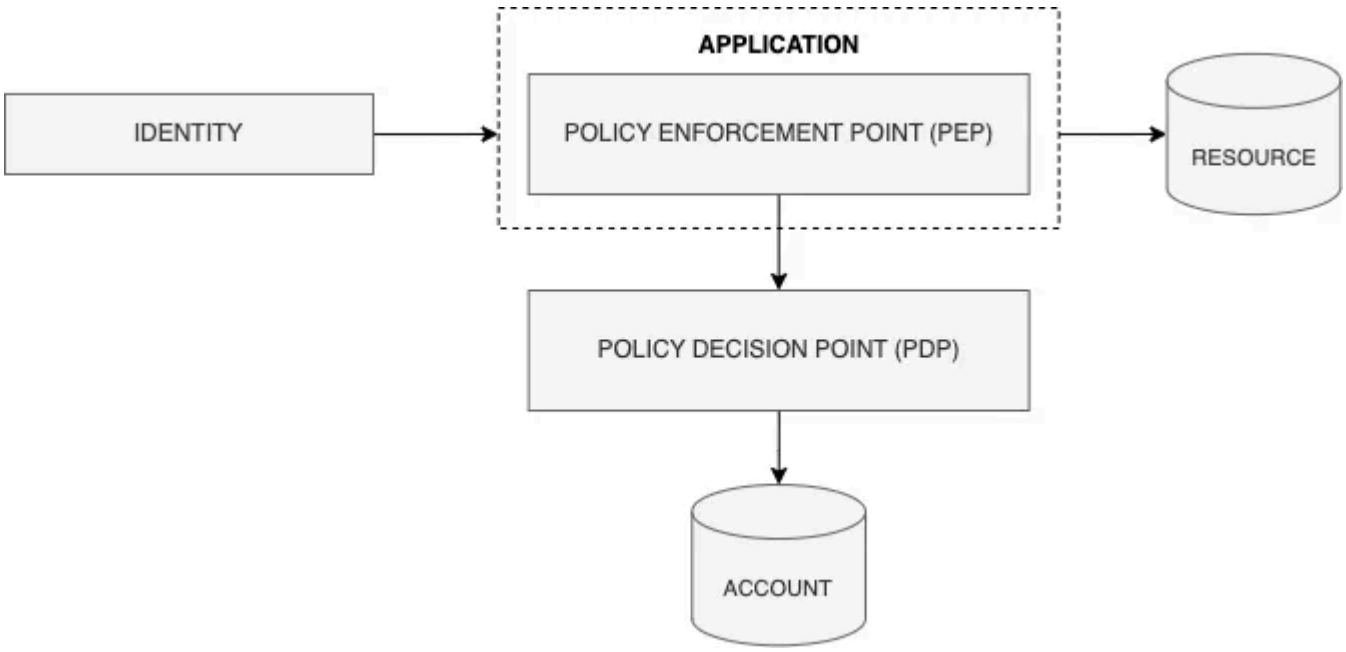
To ensure compliance with **Zero Trust principles**, we must focus on three core tenets:

- **Never trust, always verify:** By leveraging the PEP (Policy Enforcement Point) and PDP (Policy Decision Point), we ensure that trust is never assumed. Every action on a resource is evaluated against policies defined by the PDP. This guarantees that access is explicitly verified for each request.
- **Least privilege access:** Permissions are granted through the Authorization Model, meaning only assigned permissions are evaluated. Most importantly, we

rely on a solid schema that enforces least privilege, even in complex scenarios where an entity exists across multiple domains. In such cases, permissions are contextualized and segmented to avoid overexposure. Additionally, permissions can be time-boxed or revoked as needed to maintain security and adapt to changing requirements.

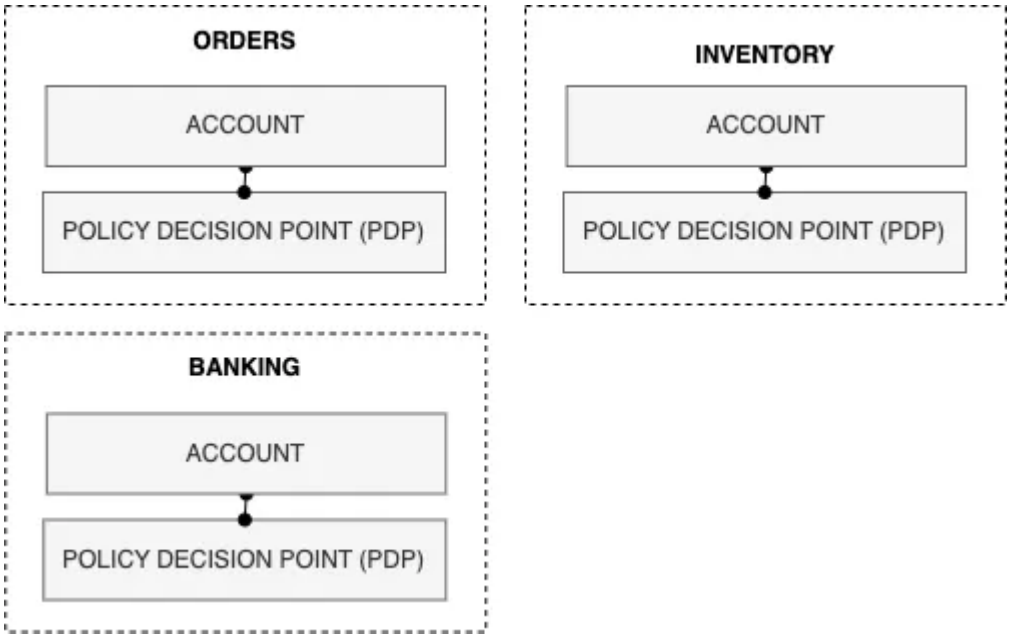
- **Assume breach:** Any software component accessing a resource must verify both the identity and the access request. This is achieved through the combined use of the Authentication Model and Authorization Model, ensuring that every request is validated before granting access.

An *account* acts as a container that holds all the necessary information within the ***Authentication*** and ***Authorization Models***, which can be collectively referred to as **Auth*.** Each **PDP** is associated with a single ***Account***. However, it is possible for multiple PDPs to share the same account information when required.

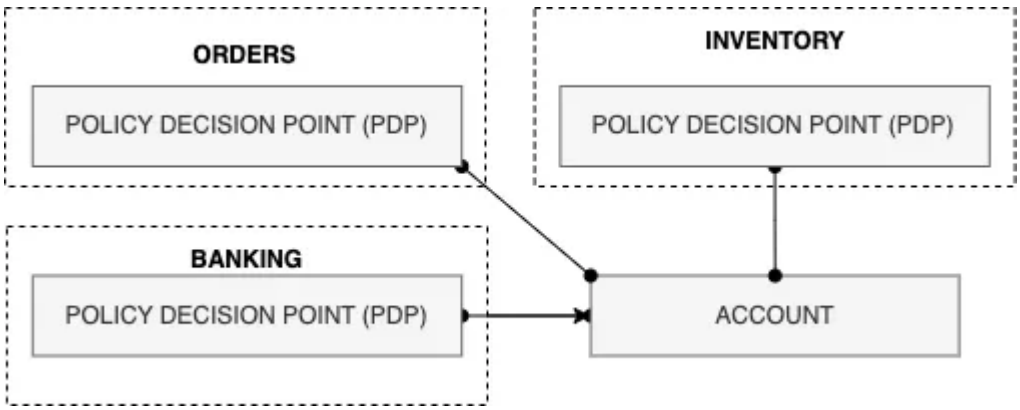



Zero Trust —PEP, PDP and Account

Taking the microservices architecture as an example, we can see that PDPs and accounts can be designed in multiple ways. Here, I will illustrate two possible approaches:




Architecture 2: Multiple Accounts

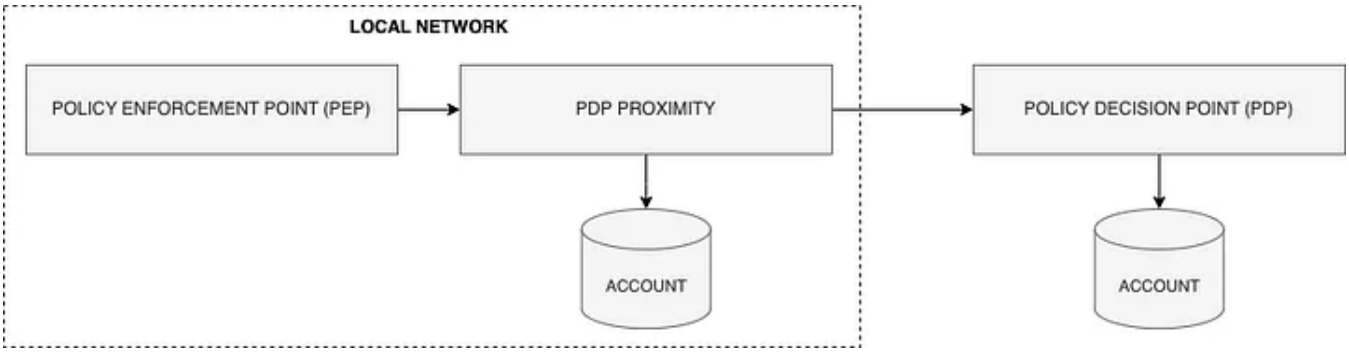



Architecture 2: Single Shared Account

It is important to note that the ***PDP*** should be located close to the ***PEP*** to minimize latency. This is why it's possible to deploy a dedicated ***PDP*** near each ***PEP***, a setup referred to as a **Proximity PDP**. For example:

- In a **Kubernetes** setup, the **Proximity PDP** could be implemented as a **sidecar** container running alongside the application.
- In a **serverless** architecture, it could function as a **Proximity Service** specifically designed to handle authorization requests efficiently.

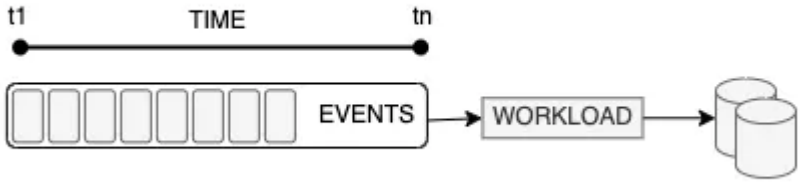
This approach ensures faster ***decision-making*** and **reduces delays** in access verification.




Proximity PDP

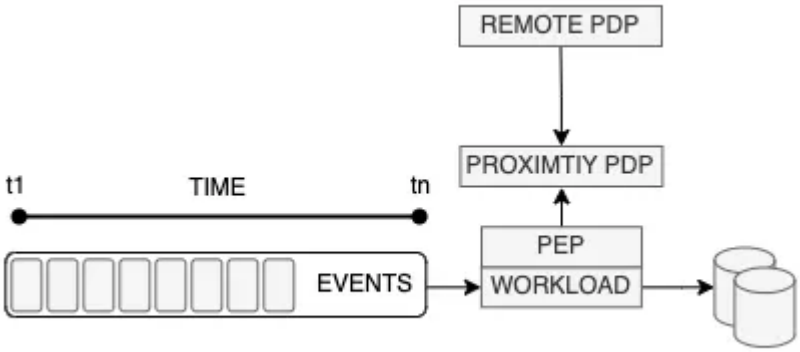
With this, we've covered the main concepts. But wait a second — you might be wondering, what about the **Autonomous and Disconnected Challenges**?

You're absolutely right; this isn't the right chapter to dive deep into those topics. However, I can give you a quick overview for now. The full explanation will come in the next chapter, as there are several additional concepts to map out before we can address it comprehensively.



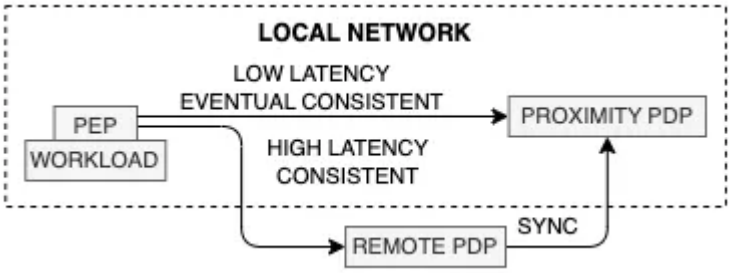

Apache Kafka, Workload and Database

Let's consider an example involving Apache Kafka and a workload where an outage disrupts communication with external systems, preventing access to the PDP.




Apache Kafka, Workload, Database, Proximity PDP, Remote PDP

In this scenario, Apache Kafka can still communicate with the workload, and the workload can communicate with the database. However, the workload cannot connect to the remote ***PDP***. This brings us to the ***CAP Theorem*** and eventual consistency. To address this, the ***PEP*** needs to communicate with a ***Proximity PDP***, which is eventually consistent with the central ***remote PDP***. By maintaining a locally synchronized copy of the Auth* model with eventual consistency, the system can continue operating seamlessly even if strict consistency is temporarily sacrificed.




Workload, Proximity PDP, Remote PDP and Eventual Consistency

Now, think about it — how often do permissions actually change? Most systems could easily handle a 20-minute outage without major issues. And consider this: if you weren't using ***caching*** for your policies, permissions, authorization, and authentication, how slow would your system be? Guess what — your cache operates on eventual consistency too!

> *If you're not very familiar with this topic, I recommend exploring the Command Query Responsibility Segregation (CQRS) pattern, popularized by Greg Young, to gain a deeper understanding of eventual consistency and its implications.*

But what about **Zero Trust**? This is where things get interesting. To align with **Zero Trust principles** and achieve **ZTAuth***, we need to ensure the Auth* model is secure and reliable. This can be accomplished by adhering to a few fundamental principles:

- **Transferable and Verifiable:** The model must seamlessly function across systems and environments while ensuring its origin is certified by the central PDP.
- **Versionable and Immutable:** Ensures integrity, auditability, and backward compatibility for secure operations.

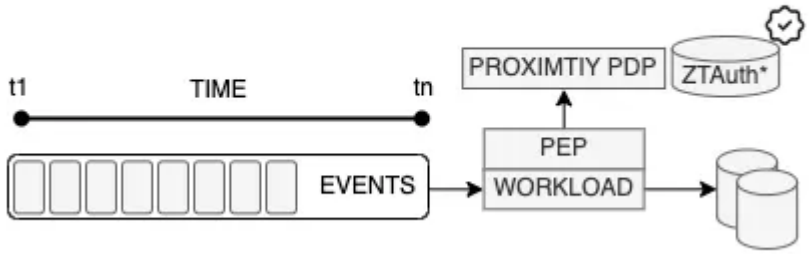
- **Resilient to Disconnection:** Supports eventual consistency, allowing functionality even in disconnected environments.

There are, of course, other concepts to consider, such as external data provided by a **Policy Information Point (PIP).** This data integrates similarly with the system, but we'll dive deeper into these aspects in the following chapters.

It should be clear by now, but some of you might still be wondering: "***Cool, but where do we store these ZTAuth* models?***" A valid concern — what if an attacker reads and extracts valuable information?

Don't worry, there are solutions. For instance, consider using **HSMs** (**Hardware Security Modules**) combined with a **OpenTDF (Trusted Data Format).**

> *OpenTDF, created by Virtru, is an open-source project for data-centric security. It uses the Trust Data Format (TDF) to cryptographically bind attribute-based access control (ABAC) policies to data, ensuring policies stay with the data. Learn more at OpenTDF on GitHub.*




Apache Kafka, Workload, Database, Proximity PDP and ZTDF

The **ZTAuth* model**, with all its principles, will be transmitted and persisted using **OpenTDF** to ensure secure and interoperable data handling. **OpenTDF** is one potential solution worth exploring to understand how such a system can be implemented. Additionally, the model will be securely stored using **Hardware Security Modules (HSMs)** to ensure maximum protection and integrity throughout its lifecycle.

As you can see, there are many **techniques** and **strategies** to explore. Stay tuned for more insights!

**Next Chapter:** Unlocking Zero Trust Delegation through Permissions and Policies

Ztauth Zero Trust Policy As Code Security Cybersecurity

ZTAuth*

Following

## Published in ztauth

4 Followers · Last published 6 days ago

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