WHITEPAPER

Enhancing Data Security with Dynamic Authorization

A Guide to Mitigating Data Access Vulnerabilities with PlainID

plainID

TABLE OF CONTENTS

Policy-Based Access Control: And Its Importance to Protecting Data

Everything is Data. While this phrase may be a generalization, it holds true. In today's digital world, everything ultimately boils down to entries in some form of data medium. Streaming movies? That's just accessing media data in a sophisticated way. Checking your bank account? You're retrieving rows in a database. Moving money or purchasing stocks? These actions translate to updating data entries.

Organizations rely on vast amounts of data to fuel decision-making and innovation. However, without robust access control mechanisms, this data is unnecessarily exposed to risk. Often, discussions about data security center on protecting the database itself—through encryption, hashing, or permissions at the table or column level. Yet, this narrow focus overlooks the critical need to protect data consistently across every layer of the technology stack. Organizations inadvertently overexpose data by relying solely on the databases' permissions and security mechanisms, leaving vulnerabilities throughout their systems.

These gaps have real-world consequences. In 2022, 70% of breaches were attributed to unauthorized access, highlighting the shortcomings of conventional security measures in addressing sophisticated attack vectors. This alarming trend underscores the need for granular, context-aware data security strategies to effectively mitigate unauthorized access risks.

This is where **Policy-Based Access Control (PBAC)** becomes essential. PBAC enables organizations to implement dynamic, context-driven security policies, ensuring data is protected consistently across the tech stack.

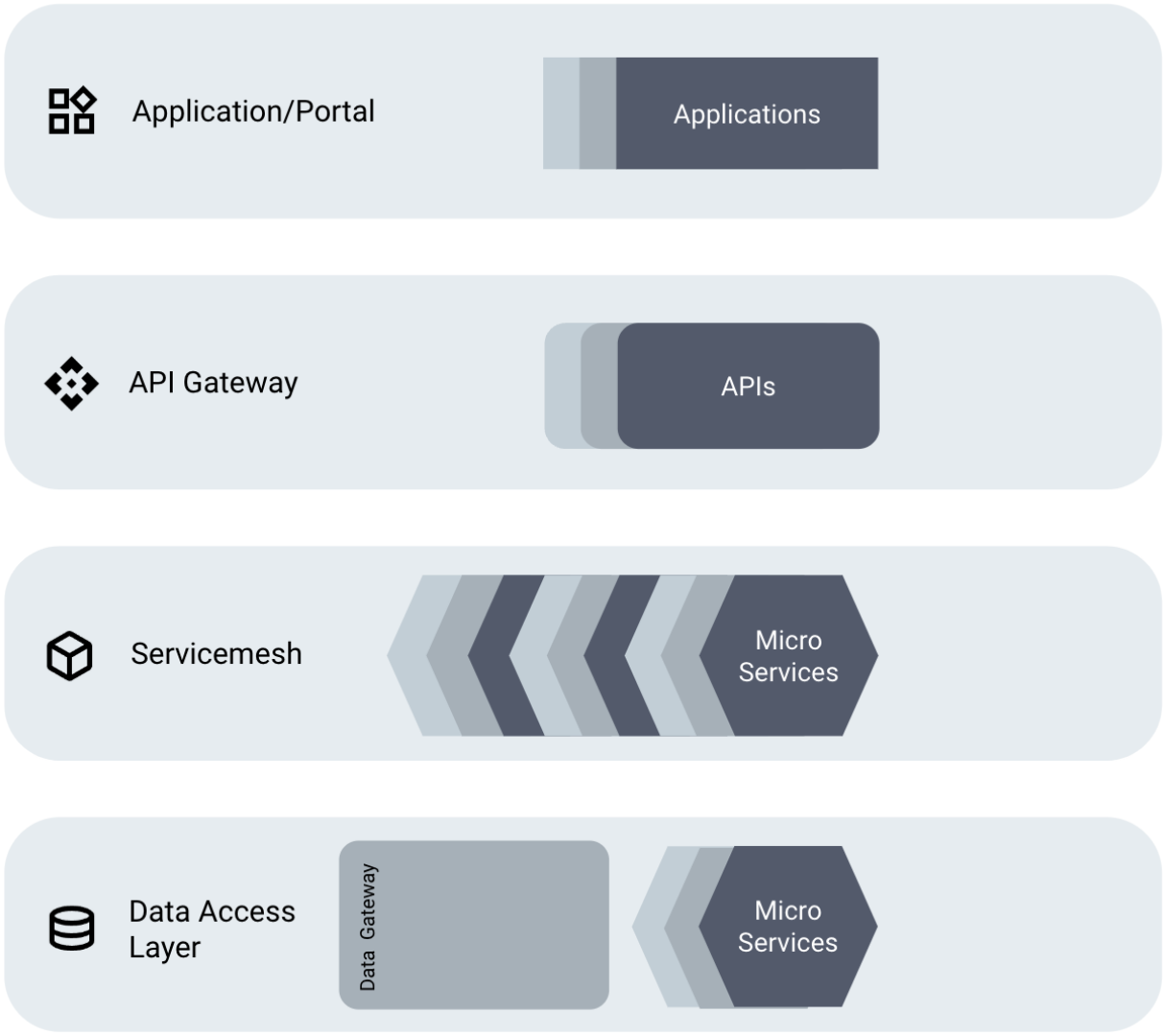
Why PBAC?

- **Identity-Aware Authorization**: PBAC leverages identity attributes—such as user roles, departments, and contextual factors—to make access decisions that are highly specific and relevant. This ensures that only the right individuals access the right data at the right time.
- **Granular Access:** Policies are designed to align with organizational objectives and compliance requirements, ensuring precise control over operational data.
- **Enhanced Security:** Contextual policies account for factors like user location, device security, or data sensitivity, reducing the risk of unauthorized access.
- **Scalability:** PBAC automates and centralizes access management, making it adaptable to the needs of growing organizations.

By implementing identity-aware, policy-driven authorization, PBAC empowers organizations to harness their operational data securely and effectively, driving innovation while maintaining compliance.

The Problem

To truly understand the problem with our current mindset that data security is solely in the database, we will need to break down a typical business case and show its components. Take a typical (simplified) tech stack:



Imagine a banking application designed for portfolio or wealth managers at a financial firm. When a user logs in, they see a tailored set of accounts or portfolios they have permission to access.

How does this process work behind the scenes?
Here's what happens step by step:

1. **User Authentication:** The application consumes the IDP token to identify the logged-in user.
2. **API Request:** The application makes a REST call to an API endpoint, such as `/accounts`, exposed by the API gateway.
    - (If you've read earlier content on securing APIs and addressing BOLA/BOPLA issues, you'll understand the importance of protecting such endpoints to avoid exposing all accounts unnecessarily.)
3. **Microservice Trigger:** The API call triggers a microservice, which connects to the database using a service account.
4. **Database Query:** The microservice queries the database to retrieve only the accounts the user is authorized to access.
5. **Response Assembly:** The retrieved data is formatted into a JSON object and returned to the application.
6. **User View:** The application displays the relevant accounts or portfolios to the user.

This streamlined process highlights how secure data access works, from authentication to the user interface, ensuring proper permissions are maintained throughout.

Where are the Gaps?

When deciding how to protect this data, there are three common approaches to consider, each with a critical flaw.

1. Traditional Data Access Control Products

In traditional data access control strategies, the protection of account data is handled at the database level. Permissions for viewing the account table and any restrictions on users are defined as database policies or rules. However, this approach has a key limitation: the database can only enforce policies on users with database accounts. In most cases, portfolio managers do not have database accounts, so the database cannot directly enforce granular data access controls for them. That said, database policies can still support this use case indirectly. They can limit the access of the service account connecting to the database, ensuring it has only the permissions necessary for all application users. While this does not provide identity-aware controls for portfolio managers, it does help narrow the service account's access.

2. API and Service Mesh layer Protection: Protection at the API and microservice layers works well for safeguarding digital assets and small-scale data interactions. For example, API or service mesh enforcement would be ideal if this use case involved viewing a single account or making a specific trade. Trades, for instance, boil down to updating rows in a database after passing through the appropriate layers. Our previously published API white paper explores numerous scenarios where API and service mesh protections are suitable, demonstrating their effectiveness for focused data actions.

3. Embedded/Coded Enforcement in the MicroService: Embedding data access control directly within the microservice is a popular approach. While this method offers strong technical protection—since anything can theoretically be coded—it falls short in key areas: **manageability, visibility, auditability, and scalability**.

This simple use case involves a single microservice, but most modern applications consist of 40–50+ interconnected microservices. Managing access policies in such a decentralized environment becomes chaotic. Centralized policy management is essential for maintaining consistency and sanity in a standard tech stack.

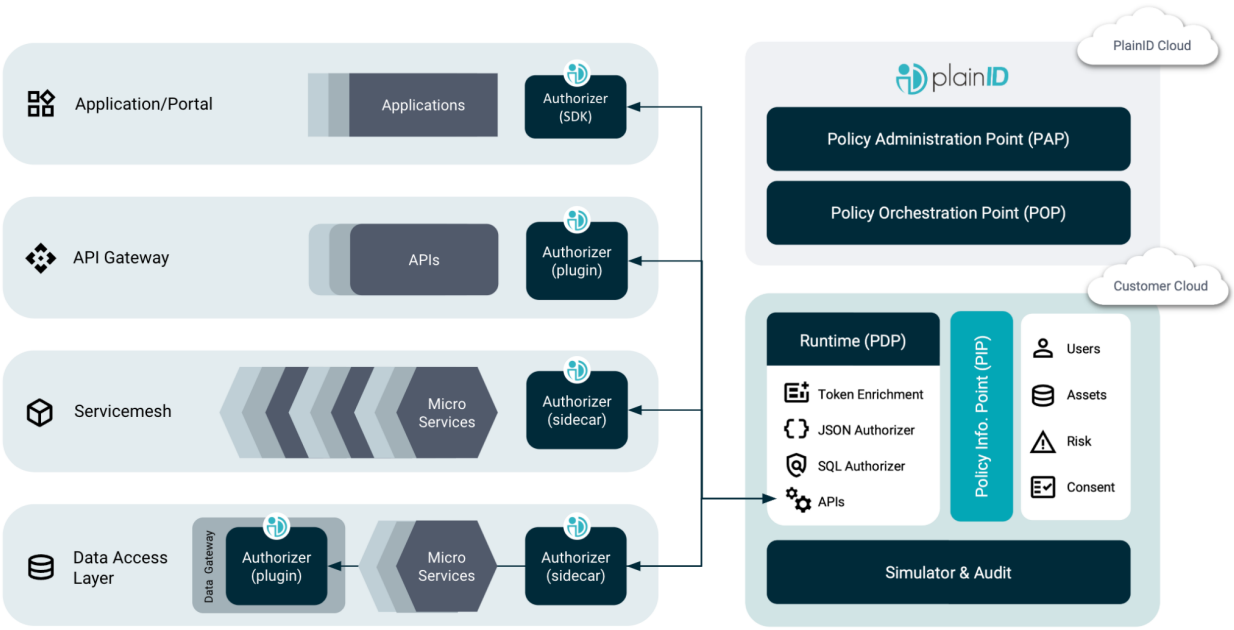
Another significant challenge arises when policies need to change. For instance, if the business decides that "Portfolio managers can see all portfolios assigned to them" should change to "Portfolio managers can see all portfolios in their region, state, or wealth management range," the embedded approach requires updating code across multiple microservices. This lack of flexibility and centralization makes the method unsustainable for dynamic environments.

How Does PlainID Address Data Access Control?

PlainID's Dynamic Authorization service provides a robust solution that dynamically adjusts access controls and permissions in real-time, ensuring secure and appropriate access across various resources within an organization's digital ecosystem. This service empowers organizations to implement fine-grained access policies that respond immediately to changes in user roles, contexts, or actions, enhancing security while supporting compliance and operational efficiency.

Key Components of PlainID’s Dynamic Authorization Service Architecture

PlainID’s architecture is designed to be highly adaptable, ensuring seamless integration with existing IT infrastructures while providing comprehensive access control capabilities.



The architecture has several key components:

- **Policy Decision Point (PDP)**: The Policy Decision Point is at the heart of the architecture, which is responsible for making real-time authorization decisions. The PDP evaluates access requests against the defined policies, considering the current context and attributes associated with each request.
- **Policy Administration Point (PAP)**: The Policy Administration Point provides a centralized interface for creating, managing, and updating access policies. Utilizing a user-friendly graphical interface, the PAP enables technical and non-technical stakeholders to define access rules and policies.
- **Policy Information Point (PIP)**: This component retrieves relevant attribute data from various sources within the organization's IT environment. The PIP supplies the PDP with the necessary contextual information and attributes required to make informed authorization decisions.
- **PlainID Authorizers (PEP)**: Located at the access request points within the system, the Policy Enforcement Point intercepts access requests and forwards them to the PDP for evaluation. Once a decision is made, the PEP enforces it by granting or denying access based on the PDP's decision.

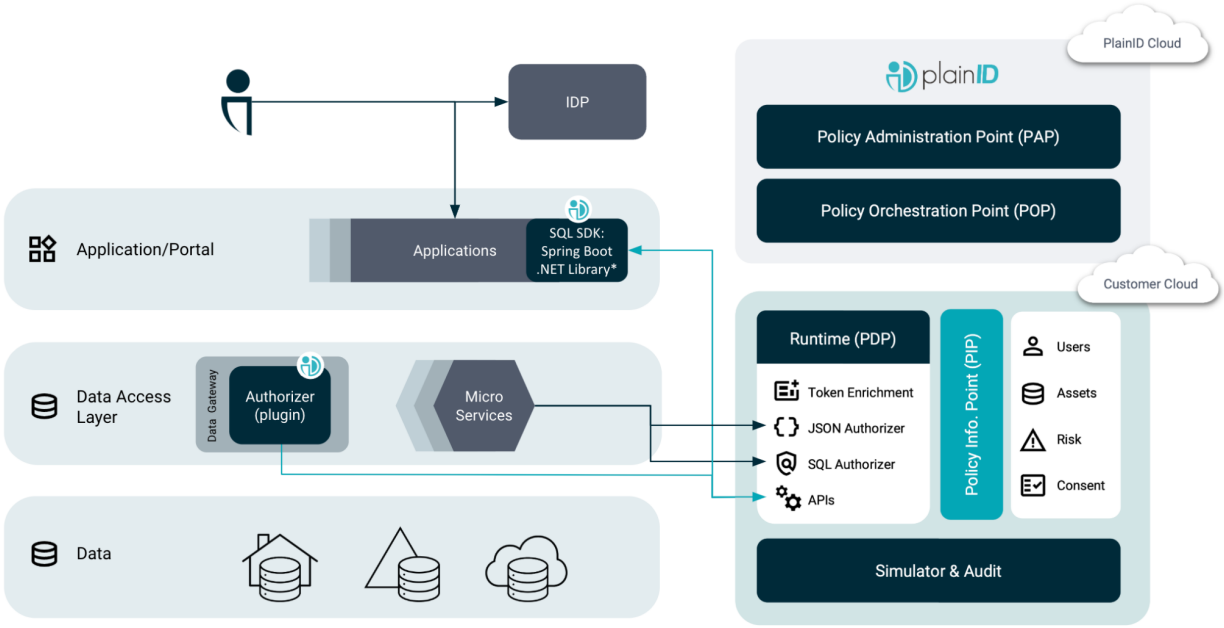
PBAC for Data Access Control

PlainID's methodology in protecting data against overexposure and unauthorized access is distinguished by its PBAC implementation. This approach is bolstered by deploying authorizers in strategic locations such as inside the application libraries, as a plugin inside a data access layer, or even as a helper service in the microservice layer. Such a design ensures a thorough

security measure spanning across various layers of the digital infrastructure to prevent unauthorized access effectively.

By integrating Authorizers within these critical points, PlainID offers a layered defense strategy. This enhances the security of applications against sophisticated threats and vulnerabilities, ensuring that access controls are precisely applied and managed across different levels of the application stack.

PlainID for Data Access Control



Policy Administration Point: By leveraging the Policy Administration functionality, PlainID allows organizations to craft policies that directly address vulnerabilities introduced by relying solely on niche Data Access Products. These policies can define precise access controls, specifying which users can access which pieces of data, both column and row protections, and how, allowing you to provide data masking.

Policy Information Point: The Policy Information component plays a crucial role in collecting up-to-date information about users and the data being accessed. The Policy Information Point can be integrated with your data cataloging solution to make sure that data is protected appropriately by its label and the compliance rules associated with it. This ensures that all access decisions are made based on the latest context and categorizations enhancing security.

Policy Decision Point: With the Policy Decision functionality, PlainID evaluates access requests against the defined policies, considering the current information about the user and the collection of data they are trying to access. It makes informed decisions on whether the

requester should be allowed to access the specific set of data as well as which pieces in the data set they should be allowed to see.

Data Access Enforcement: Finally, the Authorizers enforce these access decisions, ensuring that only valid and authorized data requests are executed. These mechanisms are pivotal in preventing unauthorized access to data:

- **Data Access Layer Authorizers:** these are plugins that are installed directly into your Data Access Layer product. These plugins intercept the SQL call sent to the Data Access Product and can manipulate the SQL statement itself based on authorization rules. Go from “select * from clients” to “select * from clients where region.client = ‘US’” without having to change a single line of code in your entire environment.
- **Data Access Libraries:** These PlainID libraries are built to interface with two of the most popular frameworks for accessing data (Spring and .NET). Once included within an application, these libraries will override the access controls in the framework and inject PlainID, allowing PlainID to enforce that only authorized SQL statements are run against the database. While this does require the application to be redeployed, it does not require any code to be changed.
- **SQL Authorizer:** The PlainID SQL Authorizer is a companion service designed for seamless deployment within an environment. It enables organizations to enhance data security by sending SQL statements to the service along with the identity of the user requesting the data. The Authorizer then modifies the SQL query to enforce user-specific authorization, returning a fully-formed query that ensures access is tailored precisely to the user's permissions.
- **JSON Authorizer:** The PlainID JSON Authorizer is primarily designed for use in API and Service Mesh environments but can also control access to data. For instance, in the earlier example, data retrieved from the database was transformed into a JSON object before being sent to the application layer. The PlainID JSON Authorizer could integrate with the service mesh or API Gateway to redact any unauthorized information from the JSON response, ensuring the user only receives data they are permitted to access.

By integrating these components, PlainID offers a dynamic and robust solution to safeguard against vulnerabilities, ensuring that access to data through the many egresses of your ecosystem are protected. What makes PlainID unique is that although there are a few different avenues to protecting data as mentioned above. All of this is boiled down to managing a few policies. The policies themselves are not dependent on which type of enforcement (or authorizer) is used in the solution.

How to Setup and Configure PlainID Components

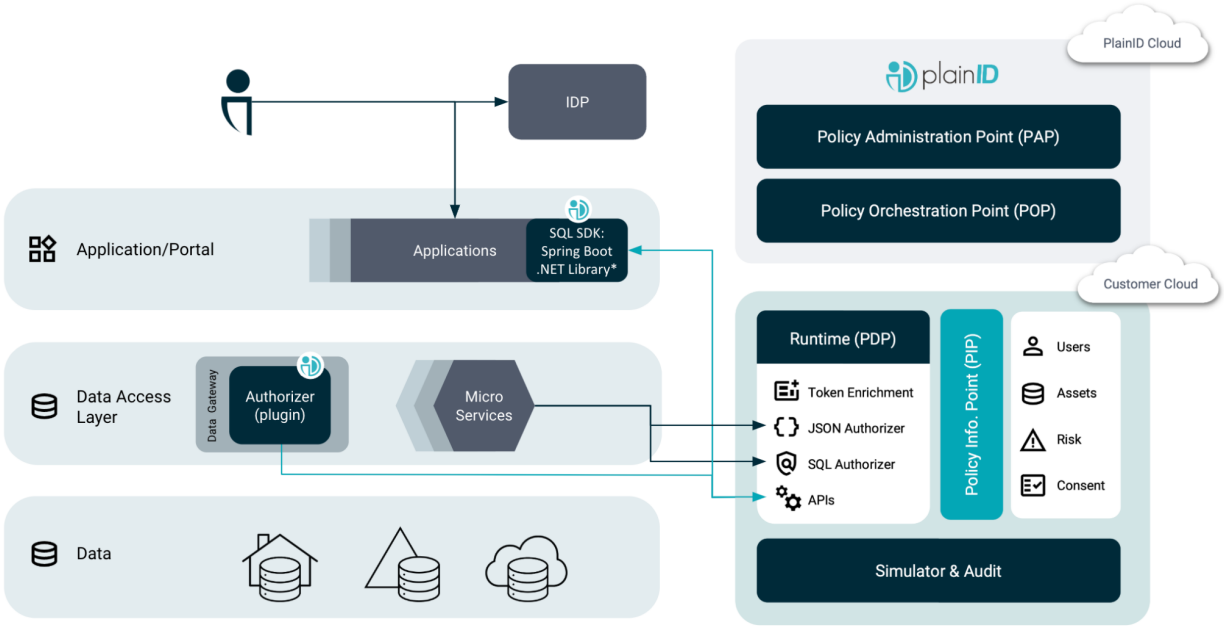
In this next section, we will review the steps to mitigate these vulnerabilities with PlainID.

- How to build a policy in PlainID that protects a digital asset
- How to use PlainID building blocks of assets and Data Mappers to teach PlainID how that digital asset is represented in the Database

- How Authorizers work in this specific interaction

In the sections, we will use a sample use case for protecting retail bank accounts, controlling who can view the balances, and allowing transactions to and from the bank accounts.

See below for the sample architecture:

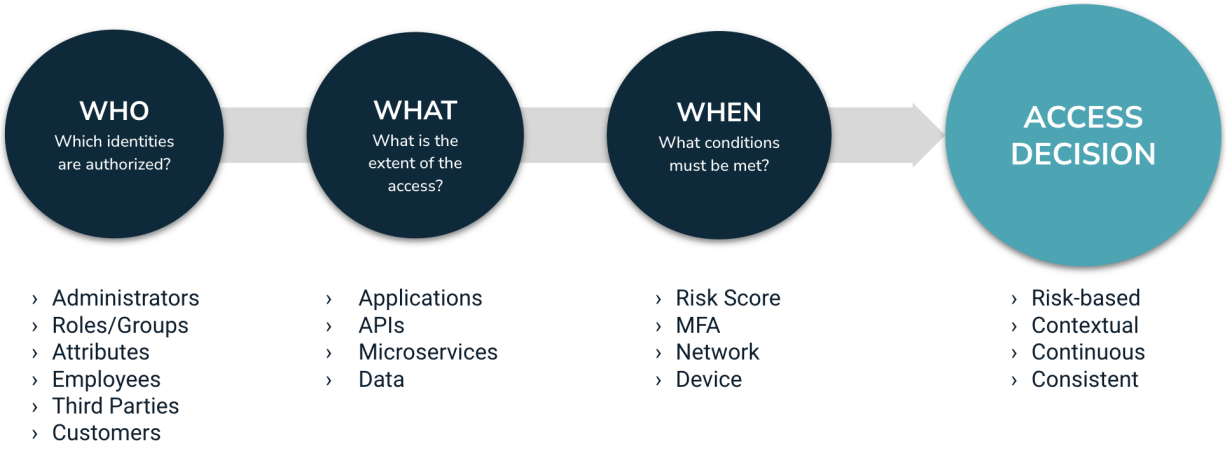


How to Build a Policy

Building a policy in PlainID involves a series of steps designed to tailor access controls precisely to your needs.

1. Define the subjects (ie. who) the policy applies to, such as users or groups.
2. Specify the actions (ie. what) these subjects can perform such as read, write, or delete.
3. Determine the resources (ie. on what assets) these actions can be applied to, such as specific files or databases.
4. Set the conditions (ie. when) under which these permissions are valid, potentially including time-based restrictions or context-specific rules.

This structured approach ensures a robust and flexible implementation of access control policies.



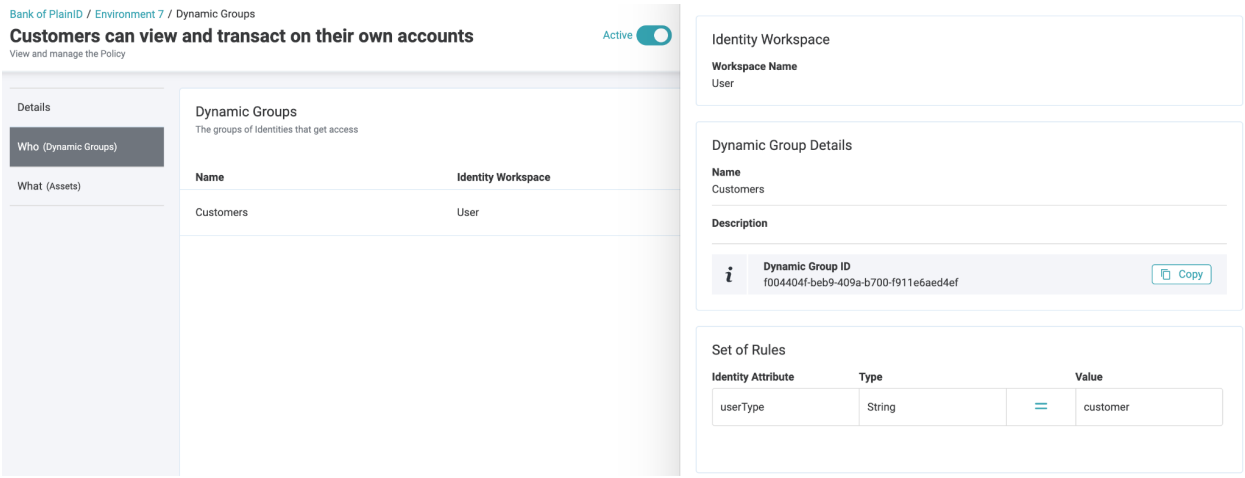
In this example, the policy used is represented in plain language as:

- "Portfolio Managers can view – all Portfolios that have been assigned to them in the CRM."

For more in-depth information on policy creation and best practices, please see the PlainID Policy Administration guide.

Users (Who)

Contained within each policy is a 'Who' section, a place to choose which Dynamic Group this policy will be granted access for. It is important to know that there are no "users" in PlainID when configuring a Dynamic Group. You are configuring a set of logic that will run at runtime to determine if the user attempting an action matches any policy. In the example above, we will configure a Dynamic Group of customers. To determine if a person is a customer, we will have the PIP configured to look in the CIAM user database and see if they exist there. It is always the job of the PIP to fetch any and all Identity fabric information from any source or several sources and normalize it for the PDP to use. For more information on configuring the identity workspace and the PIP, please refer to our official documentation.



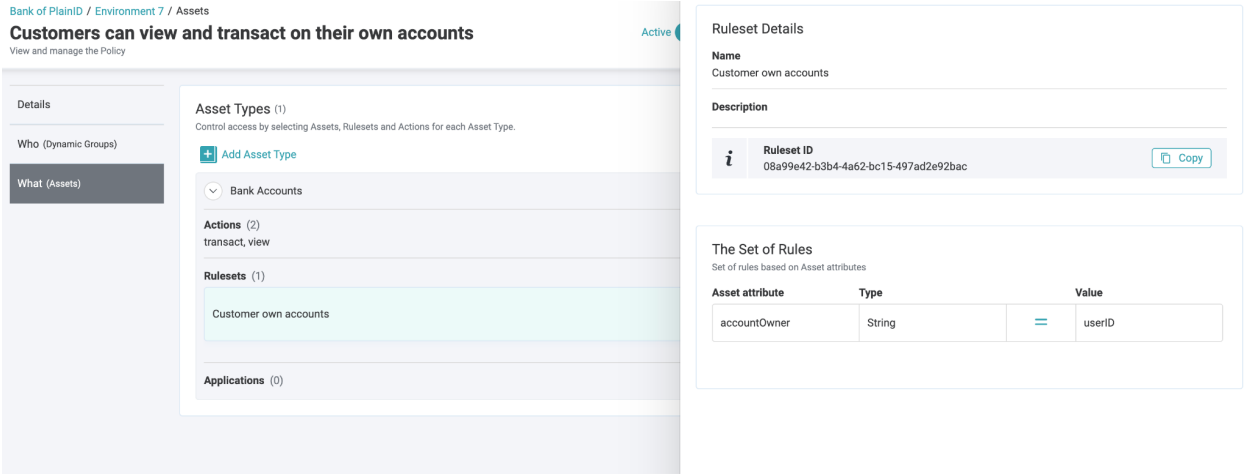
Assets (What)

Contained within each policy is an asset you are granting access to – this is where the PlainID solution starts to differ from others and why it can solve the gaps in data access. In PlainID, an asset is the digital asset you want to protect. While this whitepaper refers to database tables, the database tables are not the asset.

The **asset** represents the digital record you aim to protect, such as a bank account, insurance claim, or healthcare record. SQL serves as the mechanism to query this data while enforcing access controls.

In this example, we'll configure the assets in PlainID using **Portfolios** as a placeholder. This involves connecting the **PIP (Policy Information Point)** to the **Portfolio Account Database**, which stores details like account numbers, current balances, and transaction history.

For our use case, PlainID will be configured to fetch the **accountNumber**, ensuring that the user attempting to view account details is explicitly assigned to work on that account.



Connecting the Database Structure to PlainID Assets

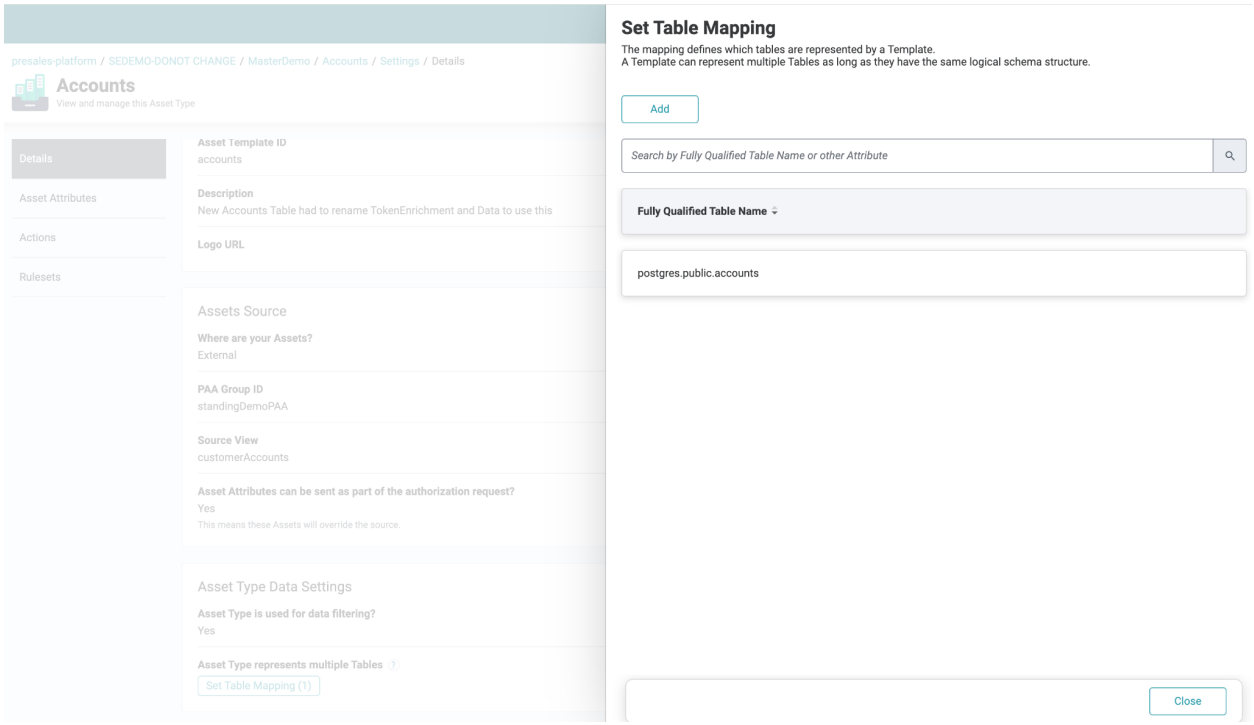
Now that we have our basic building blocks and have built and tested our policy to see that an end user can view and make transactions against their own bank accounts using the PlainID Policy Simulation tool. Now, we connect our Data structures to our policy structures. This is done using what PlainID calls Table Mappers. For more information, please see the official documentation.

Table Asset Mappers

The PlainID Asset that represents the accounts will be used to apply row-level filtering to the SQL queries that need to be executed on behalf of the user. For instance, in the example above, this will eventually turn a generic base SQL query that would be the base for all Customer data being pulled out of the database to one that is specifically crafted for the authenticated user and respects the authorization policies, making sure that data only authorized to be viewed by the user comes back. An example base SQL query would be `select first_name, last_name, account_number, account_balance from postgres.public.accounts` eventually this will need to be changed to `select first_name, last_name, account_number, account_balance from postgres.public.accounts where accountOwner = {{logged_in_username}}` the issue you can see arising is that the table name and the asset name are not always so cut and dry simple.

Table Mappers are the bridge from the technical world to the business world. In the business world, we know we need policies to protect accounts and data associated with those accounts, but in the technical world, that might be bankingdatabase.prod234f.prod_accounts_table or something even more obscure.

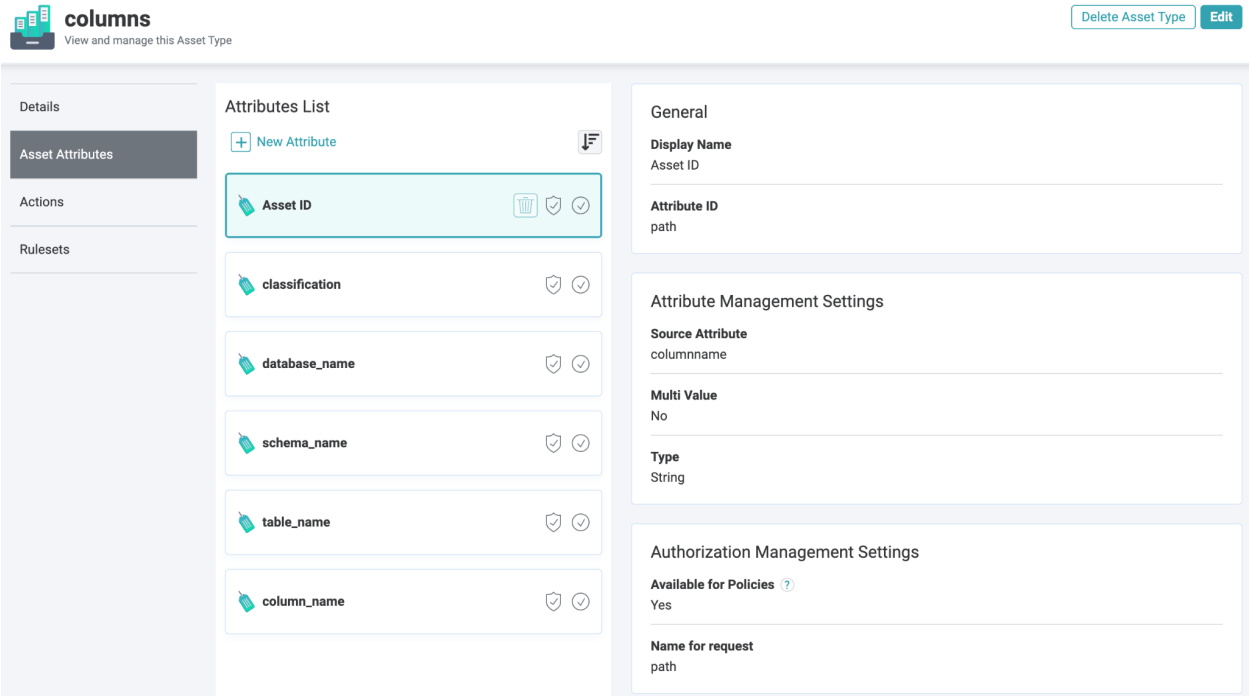
With PlainID, you can create and test your policies as we have done before, then come back and configure the Table Asset Mappers, which can help PlainID understand that the accounts table is in one or many databases, all of which need to be governed by the same set of policies.



This configuration will help PlainID use the information in the base SQL query sent to the authorizer to determine which policies should be evaluated. The evaluation of these policies is what will drive everything from redaction and masking in JSON to row filtering and column masking in SQL statements.

Columns/Keys Asset Type

PlainID can also add masking or redaction of columns or JSON keys; to do this we will create another Asset Type in PlainID that is representative of the specific data inside the table or JSON objects we wish to apply data access control on. The way to think about it is which accounts we want to give someone access to are controlled by policies that grant actions (such as “view”) over the Accounts Asset Typ,e which is mapped to the various tables and can be paired with which elements under that account the end user should have specific actions over such as “view” or even “view as masked”. This Asset type will be another “What” we attach to the policy from above. Taking a look at the Column/Key Asset Type we will use in this example looks like this.

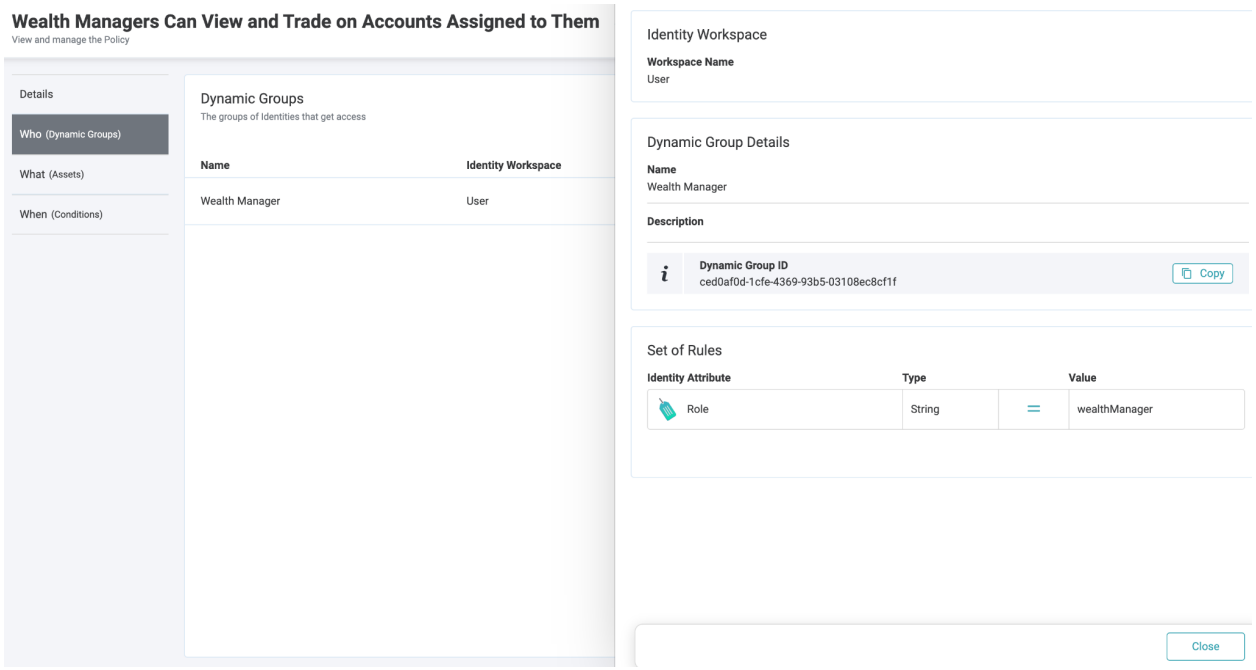


In this example, you will also notice that classification is an attribute tied to all the columns and keys. This is because, similar to the Accounts Asset Type where the information comes from the Accounts table in the Columns/Key Asset Type the information used in the policy evaluation is coming from a Data Governance Catalog. This allows enterprises to apply Data Compliance Policies while applying identity-aware security to their ecosystem.

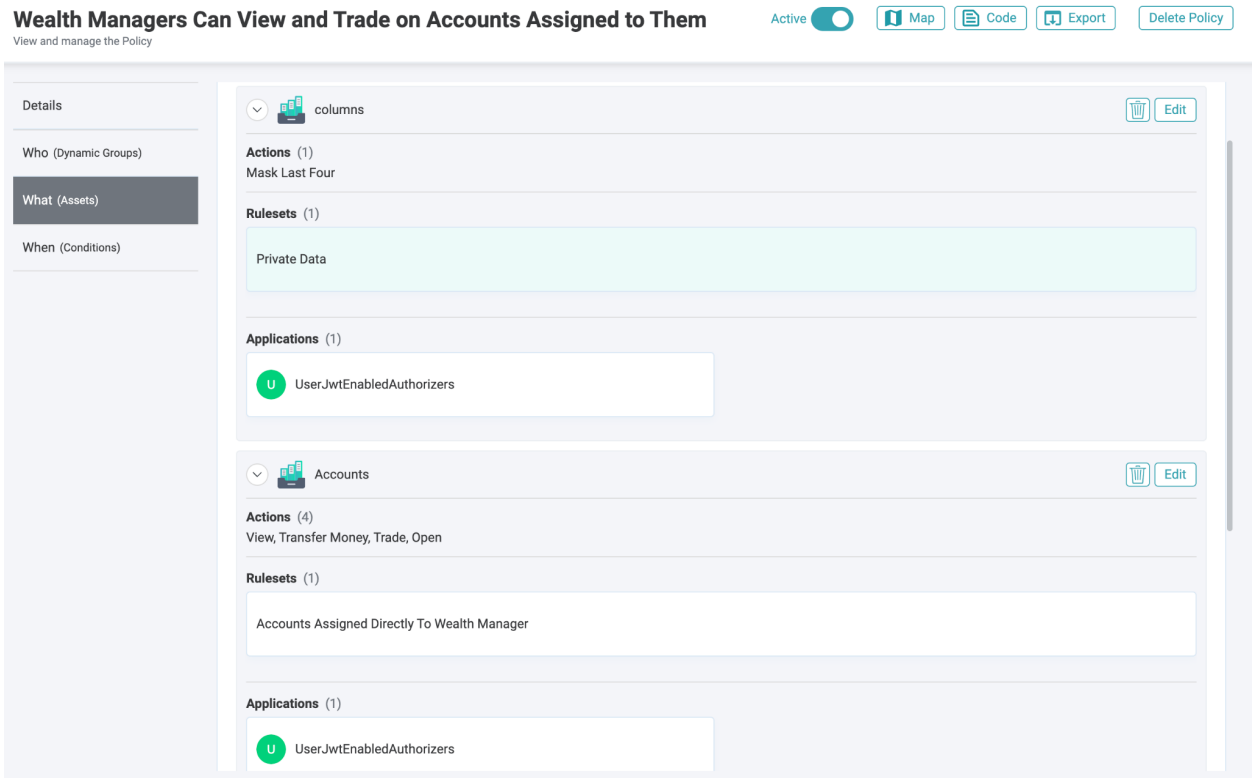
See The End Result

Now that we have everything in our policies, from which Accounts a user can see to which information (columns or keys) users can view, let's see a policy in action. This Policy states that **“Wealth Managers Can View and Trade on Accounts Assigned to Them.”** It protected the same assets as the policy above, with end users being able to view their accounts but has the added pieces for ensuring that the Wealth Managers can not see any data in the accounts not pertaining to their job.

WHO

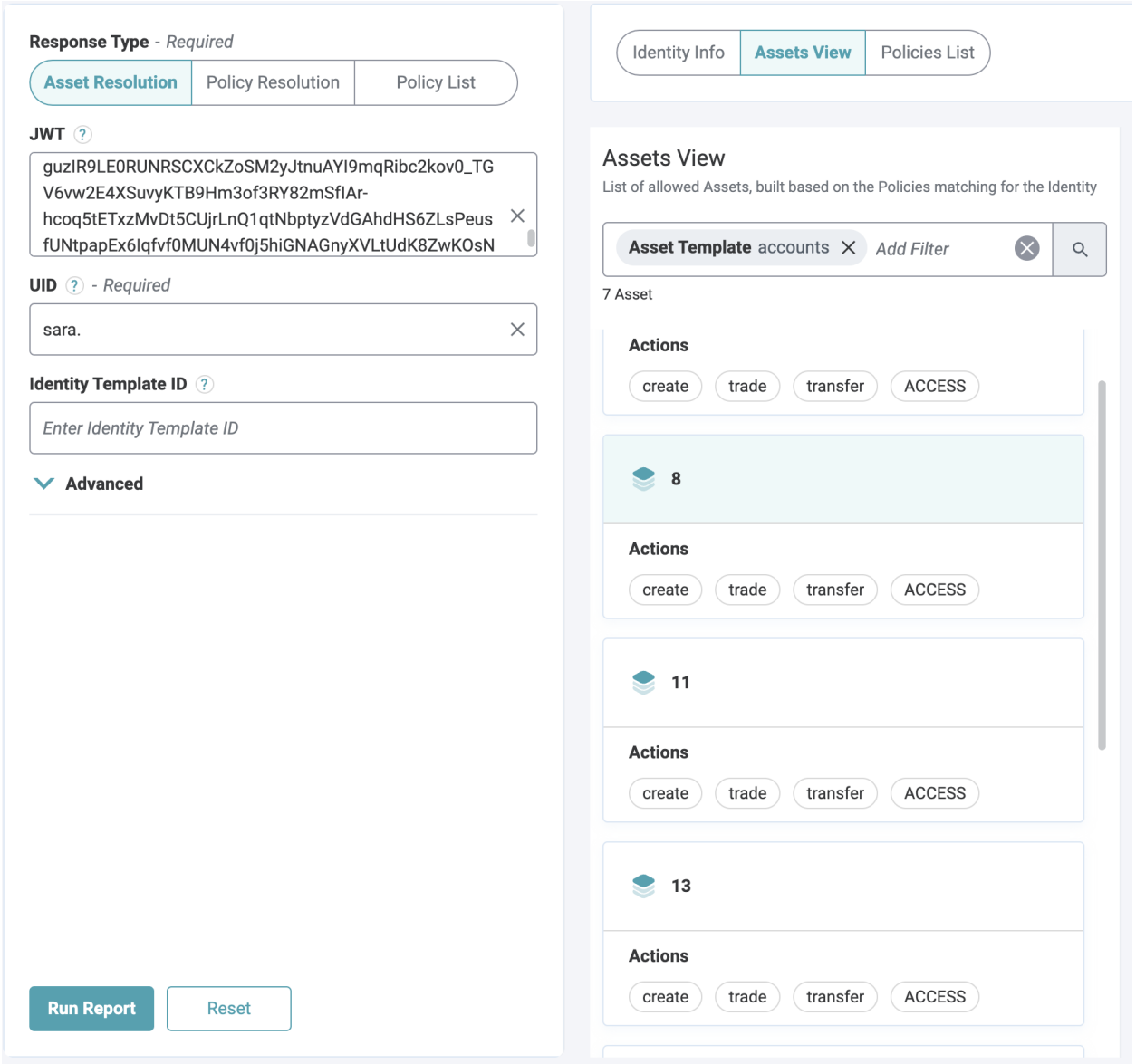

Wealth Managers Can View and Trade on Accounts Assigned to Them
View and manage the Policy
Details
Who (Dynamic Groups)
What (Assets)
When (Conditions)
Dynamic Groups
The groups of Identities that get access
Name
Identity Workspace
Wealth Manager
User
Identity Workspace
Workspace Name
User
Dynamic Group Details
Name
Wealth Manager
Description
Dynamic Group ID
ced0af0d-1cfe-4369-93b5-03108ec8cf1f
Copy
Set of Rules
Identity Attribute
Type
Value
Role
String
=
wealthManager
Close

WHAT

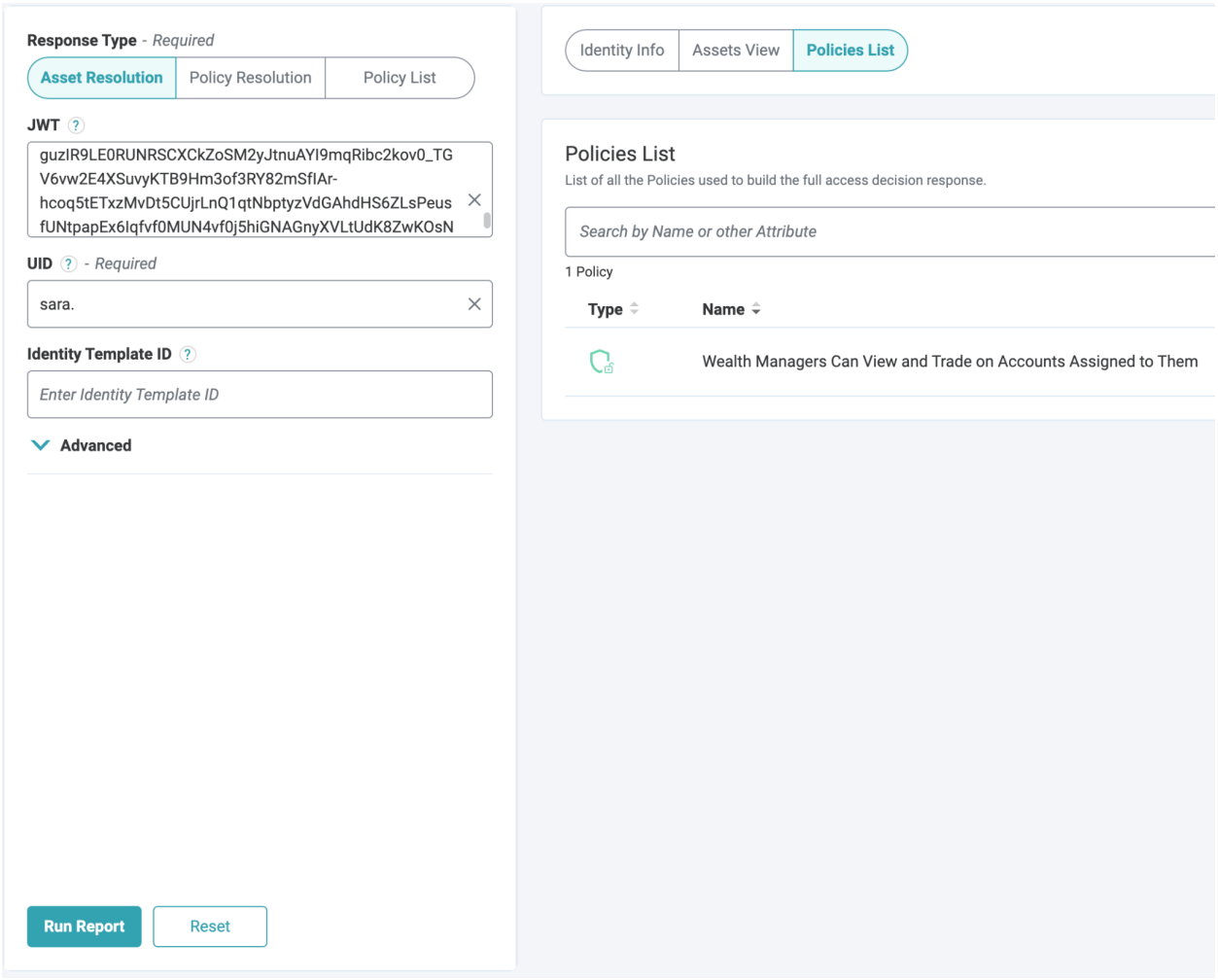

Wealth Managers Can View and Trade on Accounts Assigned to Them
View and manage the Policy
Active
Map
Code
Export
Delete Policy
Details
Who (Dynamic Groups)
What (Assets)
When (Conditions)
columns
Edit
Actions (1)
Mask Last Four
Rulesets (1)
Private Data
Applications (1)
U
UserJwtEnabledAuthorizers
Accounts
Edit
Actions (4)
View, Transfer Money, Trade, Open
Rulesets (1)
Accounts Assigned Directly To Wealth Manager
Applications (1)
U
UserJwtEnabledAuthorizers

Evaluation

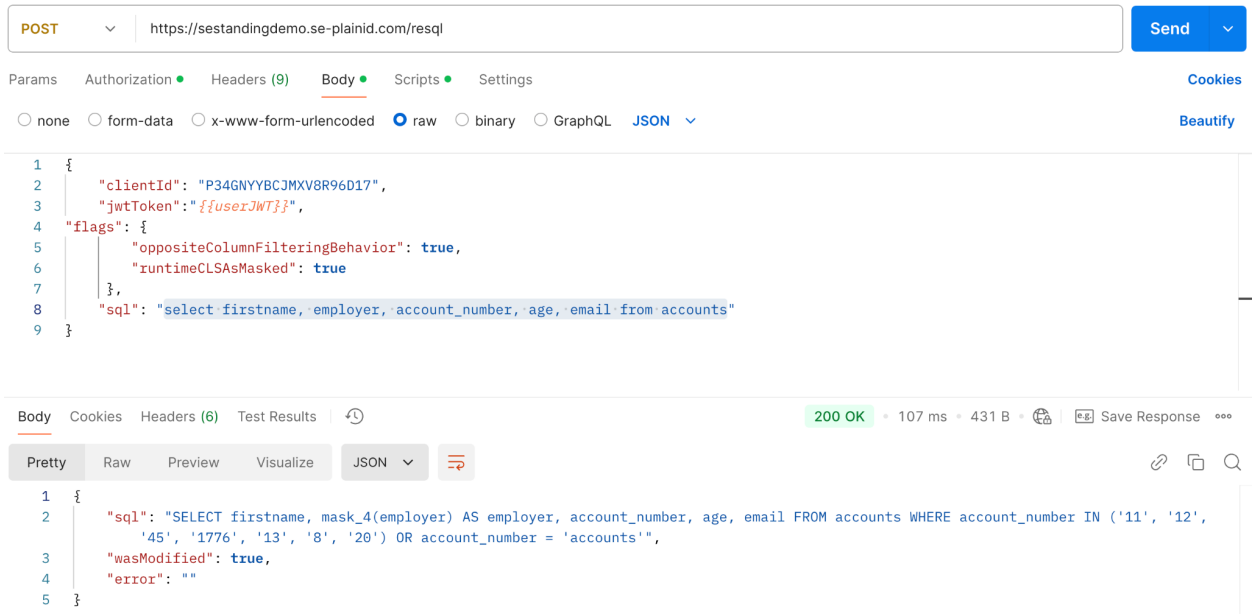
Let’s use PlainID Policy Simulator to test the Policy without needing to know the technical side of the use case first.



As you can see from the image Sara.Jameson who is our wealth manager in this use case has access to quite a few accounts. In this case she should be able to view accounts 8,11,13,20,45,1776. And, this access was granted by the Policy we looked at in the previous section.



Now that we know our policy is acting in accordance with the business. Let's see how PlainID works at applying this Policy to the protection of data. If you recall from the Problem section where we lined up the environment for this use case, we have an application triggering a REST call to get all the accounts the end user should be able to access. This, in turn, triggers a microservice whose job is to get data out of the database. For the following example, we will pretend to be that microservice. This microservice will be integrated with PlainID and knows that when an end user is requesting to see accounts, it should reach out to PlainID and use its service to get an authorized fully formed SQL query as outlined in the "PlainID for Data Access Control" section. The microservice will send the end user's JWT token that was passed to it from the application through the API and its base query 'select firstname, employer, account_number, age, email from accounts' to the PlainID SQL Rewrite authorizer and receive the SQL statement it should run against the database in return.



As you can see here, the where clause was added in accordance with the same accounts you saw in the policy simulator used by the business to test the policies, and the employer name of the user who owns the account will be masked using a stored procedure in the database. The microservice will then run this SQL statement instead of the base one.

Conclusion

PlainID's Dynamic Authorization Service, consisting of Policy Decision Points, Policy Information Points, Authorizers, and API Mappers within a Policy-Based Access Control framework, offers a sophisticated solution to address Data Access control vulnerabilities that revolve around identity-aware security controls. By leveraging real-time data and policy-driven decision-making, PlainID ensures that only authorized access to data occurs, effectively mitigating risks due to the gaps in protecting operational data using identities not part of the Data Mediums. This significantly enhances an organization's identity security posture and closes significant gaps in cybersecurity by ensuring granular, real-time authorization decisions, which protect against unauthorized access and data breaches.