**ĐẠI HỌC QUỐC GIA TP. HCM**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**--------------------**

**LƯƠNG VẠN HUY**

**IMPLEMENT ATTRIBUTE-BASED ACCESS CONTROL AND PRIVACY PROTECTION IN**

**NOSQL DOCUMENT STORE**

**(Xây dựng hệ thống quyết định chính sách điều khiển truy cập**

**dựa trên thuộc tính bảo vệ tính riêng tư cho dữ liệu**

**NoSQL document store)**

Chuyên ngành : Khoa học máy tính

GVHD: ThS Trần Thị Quế Nguyệt

Ts. Trương Tuấn Anh

GVPB : ThS Lê Thị Kim Tuyến

**LUẬN VĂN TỐT NGHIỆP**

TP. HỒ CHÍ MINH, 6/2016

## Abstract

Nowadays, NoSQL databases have been rapidly becoming the popular data platform for big data and real-time web applications. Simpler horizontal scaling, flexible schema designing, high performance data access have made NoSQL databases to be alternative approaches for traditional relational databases . However, there are some disadvantages in NoSQL, among which the lack of effective suppprt for access control and privacy protection is the most serious ones. The huger data we have, the more challenge in data protection we have to face. In this thesis, we address this issue by implemeting a comprehensive framework for enforcing attribute-based security policies stored in JSON document. We use Polish notation for modeling conditional expressions which are the combination form of subject, resource, and environment attributes so that the policies are flexible, dynamic and fine grained. Moreover, with the approach of attribute-based access control, we have proposed a flexible model struture for privacy protection called as Attribute-Based Privacy Protection so that it can be evaluated not only by access purpose but also by subject, resource, environment attributes. We also build a web application which interacts to our framework so that administrators can easily define and review policies. The experiment is carried out to illustrate the relationship between the processing time for access decision and the complexity of policies.

Table of Contents

**ABSTRACT2**

**Section 1. INTRODUCTION4**

1.1 Thesis Introcution**4**

1.2 Related work**5**

1.3 Purpose and Scope**6**

1.4 Thesis structure**6**

**Section 2. NOSQL DATABASE AND MONGODB7**

2.1 NoSQL Database**7**

2.2 MongoDB**10**

**Section 3. ACCESS CONTROL MODELS15**

3.1 Defintion and Classical Access Control Models**15**

3.2 Attribute-Based Access Control**18**

3.3 XACML**19**

**Section 4. PRIVACY PROTECTION22**

**Section 5. ABAC AND PRIVACY PROTECTION MODEL24**

**Section 6. POLICY STRUCTURE25**

6.1 General structure**25**

6.2 Acess Contol Policy Structure**31**

6.3 Privacy Policy Structure**32**

6.4 Conflict Resolving Approach**34**

6.5 Advanced privacy policy support for array of embeded subdocuments**39**

**Section 7. PROPOSE FRAMEWORK42**

7.1 Architecture**42**

7.2 Workflow**50**

**Section 8. APPLICATION DEMONSTRATION51**

**Section 9. EXPERIMENT57**

## Section 1. Introduction

* 1. **Thesis introduction:**

Nowadays, the quanity of data is increasing exponentially by the development of social media appications, sensor for data acquisitions and smart phone utilization. NoSQL databases is the most popular approach to handle those semi and unstructured data for a scalable application. Like relational database, security is highly considered in NoSQL database, especially when working with huge volume data. For the last decade, Discretionary Access Control (DAC), Mandatory Access Control (MAC), Role Based Access Control (RBAC) have been used almostly to handle security. However, because of the rapid development of large scale dynamic systems, those traditional access controls have gradually reveal their disadvantages, for example, applied for only closed system, role explosion and inflexibility in specifying dynamic policies and contextual conditions. To overcome those problems, Attribute Based Access Control models have been recently investigated and according to Gartner‘s prediction: “By 2020, 70% of all businesses will use attribute based access control as the dominant mechanism to protect critical assets.”[2]

Based on the Hibert and Lopez assessment of global information storage over time, Figure 1.1 shows the relative between the increasement of stored information from 1986 to 2007 and the trend of access control. Even when access control systems are successful in restricting unauthorized and unauthenticated users, they are ineffective as privacy protection for a large, decentralized system like the World Wide Web, where it is easy to copy or aggregate information. Therefore, in this thesis, we will strongly focus on access control and privacy protection.

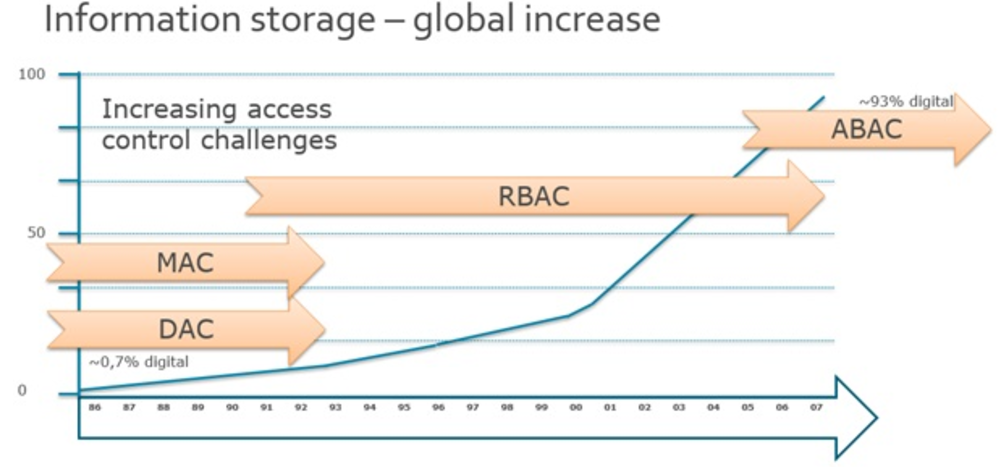


Figure 1.1 The increasement of information storage from 1986-2007

* 1. **Related work**

Our thesis is related to several topics in the area of access control and privacy protection for data management, namely policy specification, privacy-preserving data management systems. We now survey the most relevant approaches in these areas and point out the difference of our work with respect to these approaches.

Hua Wang, Lili Sun, and Vijay Varadharajan[3] have proposed a purpose-based framework for supporting privacy preserving access control policies and mechanisms. They have also developed algorithms to help a system to detect and analyze the conflicts when adding new policies. However they don’t mention much about how to model the conditional expression and the algorithms they proposed just focused only on simple attributes lacked of evaluating conditional expression.

Prosunjit Biswas, Ravi Sandhu, and Ram Krishman[4] have presented an attribute based protection model for JSON documents. Their approach is to add a new attibute called “security-label” to JSON elements and specify access control policies using these values. The advantage of the seperation of labeling and authorization policies is that they can be specified and administered independently possibly by different level of administrators. However, the number of label assignments can be very large because it is calculated by the exponential function. Therefore the space storage is considered to be a potential problem when the system is expanded.

IBM has proposed a formal language called Enterprise Privacy Authorization Language[5] called as EPAL for writing enterprise privacy policies to govern data handling practices in IT system. An EPAL policy defines lists of hierarchies of data-categories, user-categories and purpose. User-categories are the entities that use collected data, data-categories define different categories of collected data and purposes model the services for which data is intended to be used. An EPAL policy also defines sets of actions, obligations, and conditions. Actions model how the data is used.and obligations define actions that must be taken by the environment EPAL. Lastly, conditions are boolean expressions that evaluate the context. Privacy authorization rules are defined using these elements and each rule allows or denies actions on data-categories by user-categories for certain purposes under certain conditions while mandating certain obligations.

Ji-Won Byun, Ninghui Li[6] have presented a comprehensive approach for privacy preserving access control based on the notion of purpose. A key feature of their approach is that it allows multiple purposes to be associated with each data element and also supports explicit prohibitions, thus allowing privacy officers to specify that some data should not be used for certain purposes.

* 1. **Purpose and Scope**

In this thesis, our access control model is built on the principle of NIST Standard ABAC that an access decision is permitted only if the request satisfies conditions on attributes of subject, resource and environment specified in policies. Moreover, with the approach of attribute-based access control, we have proposed a flexible model struture for privacy protection called as attribute-based privacy protection so that it can be evaluated not only by access purpose but also by subject, resource, environment attributes and function defined by user. We use Polish notation for modeling conditional expressions so as to describe complex policies such as user, data, environment, driven policies. We also build a web application which interacts to our framework so that administrators can easily define and review policies. The experiment is carried out to illustrate the relationship between the processing time for access decision and the complexity of policies.

* 1. **Thesis Structure**

The thesis is organized as follows:

Section 1. INTRODUCTION presents about overview, purpose and scope of theis, also a brief survey of related works.

Section 2. NOSQL DATABASE and MONGODB presents about the definition and categories of NoSQL database. MongoDB is chosen as the representative database for document store.

Section 3. ACCESS CONTROL MODELS describes about the categories of access control system and emphasizes the importance of Attribute-Based Access Control model.

Section 4. PRIVACY PROTECTION describes about the theory, analyzes a high-light current research and presents our approach for protecting privacy.

Section 5. ABAC and PRIVACY PROTECTION MODEL presents the proposed model and introduces the main components.

Section 6. POLICY STRUCTURE presents how we model the access control and privacy policy, how we parse a conditional expression and solve the conflict between policies.

Section 7. PROPOSED FRAMEWORK describe how we design and implement a framework for access control and privacy protection.

Section 8. APPLICATION DEMONSTRATION demonstrates a web application which used our framework to manage access control and privacy policies.

Section 9 EXPERIMENT presents how we test the performance by evaluating the processing time.

## Section 2. NoSQL Database and MongoDB

* 1. **NoSQL Database:**

"NoSQL is a new way of designing Internet-scale database solutions. It is not a product or technology but a term that defines a set of database technologies that are not based on the traditional RDBMS principles."[7]

NoSQL is a term used to define non-relational databases. Therefore it encompasses majority of the data stores and databases which don’t follow the popular and well-establised RDBMS principles and often relate to large data sets accessed and manipulated on a Web scale. NoSQL databases evolved from the need of handling Big Data where the traditional RDBMS technologies did not provide adequate solutions.

NoSQL Databases are categorized on the basic of how the data is stored:

* **Key-Value database**: A HashTable is the simplest data structure that can hold a set of key/value pairs. Such data structures are extremely popular because they provide a very efficient, big O(1) average algorithm running time for accessing data. The key of a key/value pair is a unique value in the set and can be easily looked up to access the data. The value of a key/value pair can be any kind of data like an image, user preference life or document.

Common Key-Value databases: Redis, Berkeley DB, Couchbase Server,…

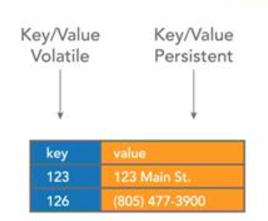


Figure 2.1 Example of Key-Value data

* **Column-oriented database**: A column-oriented database stores each column continuously on disk or in-memory each column will be stored in sequential blocks. It avoids consuming space when storing nulls by simply not storing a column when a value doesn’t exist that column. For analytical queries that perform aggregate operations over a small number of columns retrieving data in this format is extremely fast.

Common Column-oriented database: Cassandra, Riak, Voldemort,…

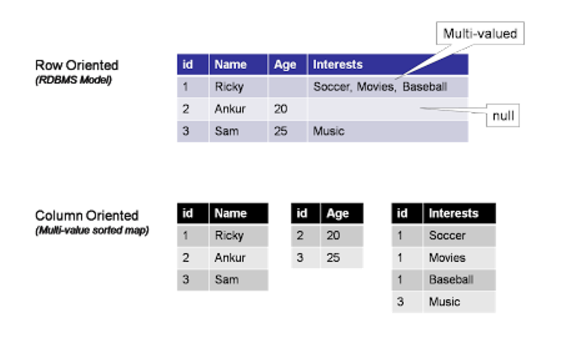


Figure 2.2 Example of Column-oriented data

* **Graph database**: Graph database is collection of nodes and edges which uses graph theory to store, map and query relationships. Each nodes represents an entity (also known as row in RDBMS) and each edge represents a connection or relationship between two nodes. Every node in a graph database is defined by a unique identifier, a set of outgoing edges and/or incoming edges and a collection of properties expressed as key/value pairs. Each edfe is defined by a unique identifier, a starting-place and/or ending-place node and a set of properties.

Common Graph databases: Neo4j, FlockDB,…

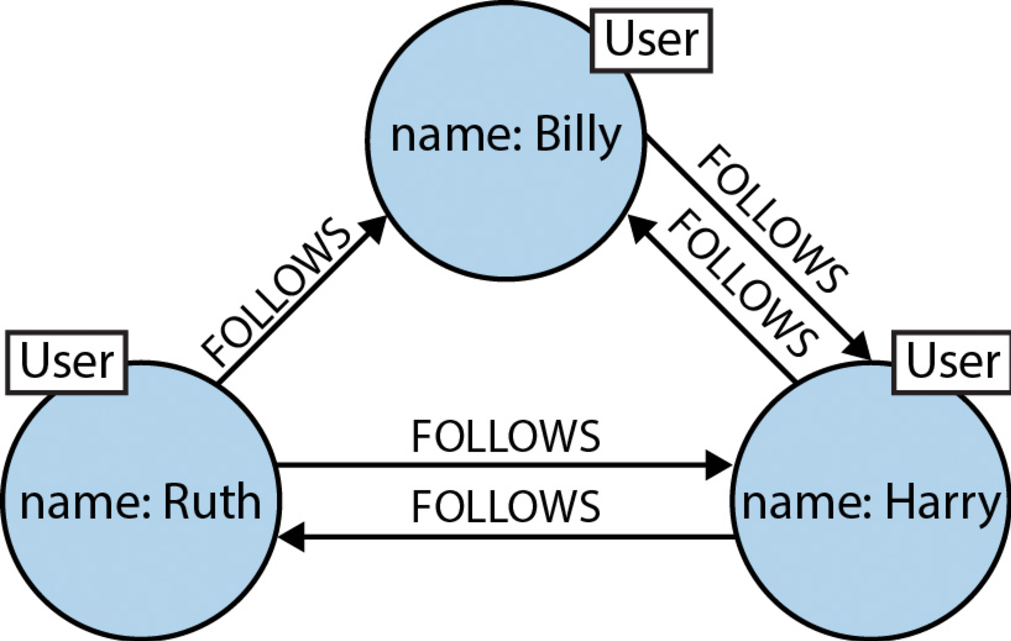


Figure 2.3 Example of graph data.

* Document Database: A document database is designed for storing, retrieving, and managing document-oriented, or semi structured data. The word document in document databases connotes loosely structured sets of key/value pairs in documents, typically JSON. Documents are grouped into "collections" which serve a similar purpose to a relational table. Document databases allow indexing of documents on the basic of not only its primary identifier but also its properties.



Figure 2.4 Example of document data.

* 1. **MongoDB Database:**
* Introduction:

MongoDB is one of the leading NoSQL document store platform which enables organizations to handle Big Data. It is written in C++ programming language which is one of the reasons for high performance in processing and retrieving data. MongoDB Drivers support for a lot of programming languages including C, C#, C++, Erlang, Haskell, etc… MongoDB uses a JSON based document store to store the data. This approach also makes it high performance by providing for grouping of relevant data together internally and making it easily searchable. Transactions and join-operators are not available in MongoDB.

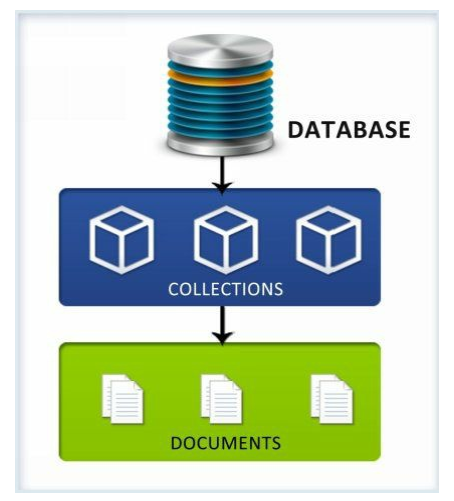


Figure 2.5 MongoDB Database Model.

*Database*: A database is a set of collections. A MongoDB deployment can have many databases.

*Collection*: A collection is a set of documents. It is like the term "table " in RDBMS but there is no relationships between two collections.

*Document*: A document is like a row in RDBMS but schema less. It is stored as Key-Value pairs and documents within the same collection can have same or different set of fields or structure, and even common fields can store different type of values across documents. This implies there’s no rigidness in the way data is stored in the documents of a collections.

* Basic commands in MongoDB:
* *Insert a new document*:

To insert a new document in MongoDB, we use *insert* API:

>db.COLLECTION\_NAME.insert(document)

"COLLECTION\_NAME" parameter is the name of collection which document will be inserted. "document" parameter is the data will be inserted.

>db.employee.insert({

"\_id": ObjectId(5983548781331adf45ec5),

"name": "John",

"title": "Developer",

"skills": ["Android", "iOS"],

"age": 25

})

* *Find a document*:

To find a document in MongoDB, we use *find* API:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Syntax** | **Example** | **Synonym in WHERE clause** |
| Equality | {<key>:<value>} | db.employee.  find({"title":"intern"}) | where titile = 'intern' |
| Less Than | {<key>:{$lt:<value>}} | db. employee.find({"age":{$lt:50}}) | where age < 50 |
| Less Than Equals | {<key>:{$lte:<value>}} | db. employee.find({"age":{$lte:50}}) | where age <= 50 |
| Greater Than | {<key>:{$gt:<value>}} | db. employee.find({"age":{$gt:50}}) | where age > 50 |
| Greater Than Equals | {<key>:{$gte:<value>}} | db. employee.find({"age ":{$gte:50}}) | where age >= 50 |
| Not Equals | {<key>:{$ne:<value>}} | db.mycol.find({"age":{$ne:50}}) | where age != 50 |

Table 2.1 Operators in MongoDB

* *Update a document*:

To update a document in MongoDB, we use *update* API:

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA, UPDATED\_DATA)

SELECTION\_CRITERIA: is the condition for which documents will be updated.

UPDATED\_DATA: is the data will be updated to documents.

>db.employee.update({"title": " Developer"},{$set: {"title": "doctor"}})  
>db.employee.find()

{

"\_id": ObjectId(5983548781331adf45ec5),

"name": "John",

"title": "doctor",

"skills": ["Android", "iOS"],

"age": 25

}

* *Delete a document*:

To delete a document in MongoDB, we use *remove* API

>db.COLLECTION\_NAME.remove(DELETION\_CRITERIA, UPDATED\_DATA)

SELECTION\_CRITERIA: is the condition for which documents will be updated.

* Access Control in MongoDB:

In MongoDB Authentication and Authorization is supported on per-database level. The users can access the database only if they login using the credentials. This disables anonymous access to the database. Authorization will be used after the authentication to ensure that the user has only the required rights for executing the tasks in hand. User exist in the context of a single logical database and are stored in the **system.users** collection within the database.

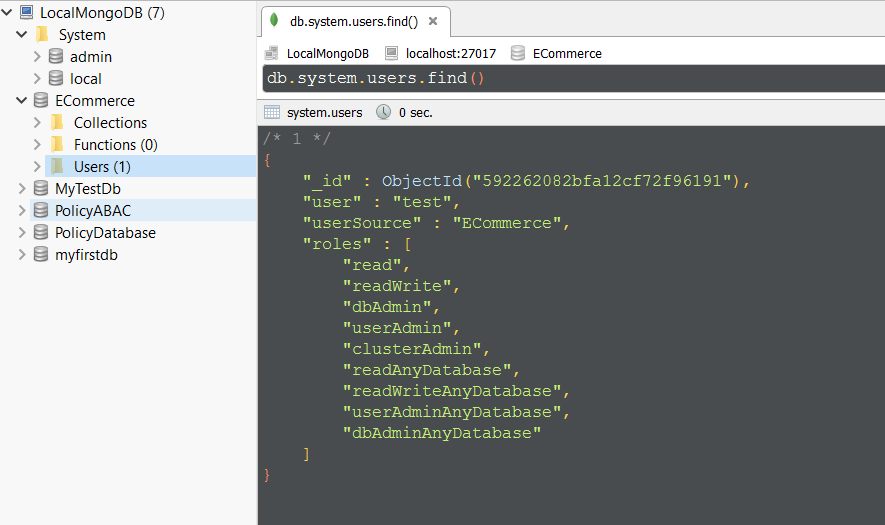


Figure 2.6. Example of RBAC in MongoDB

**system.users:** This collection stores information for authentication and authorization on that database. It stores the user’s credentials for authentication and users privileges information for authorization. MongoDB use a role-based approach for authorization. Therefore a user can have multiple roles and may have different roles on different databases. The available roles are:

*read* – This provides a read only access of all the collections for the specified database.

*readWrite* – This provides a read and write access to any collection within the specified database.

*dbAdmin* – This enables the users to perform administrative actions within the specified database such as index management using ensureIndex, dropIndexes, reIndex, renaming collections, create collections.

*userAdmin* – This enables the user to perform read and write operation on the system.users collection of the specified database, also enables them to modify permissions for existing users and create new users.

*clusterAdmin* – This role enables the user to grant access to administration operations which affects or present information about the whole system. clusterAdmin is applicable only on the admin database and does not confer any access to the local or config database.

*readAnyDatabase* – This role enables user to read from any database in the MongoDB environment.

*readWriteAnyDatabase* – This role is similar to readWrite except it is for all databases.

*userAdminAnyDatabase* – This role is similar as userAdmin role except it applies to all databases.

*dbAdminAnyDatabase* – This role is same as dbAdmin, except it applies to all databases.

## Section 3. Access Control Models

* 1. **Definition and Classical Access Control Models:**

Access Control is a way of selectively restrict access to a specific resource. For example, controlling access to physical locations such as a room or controlling access to secret formulas and financial secrets. Access control appears whenever people have the need to share their resources selectively. The actual process of obtaining the access to resource is known as authorization.

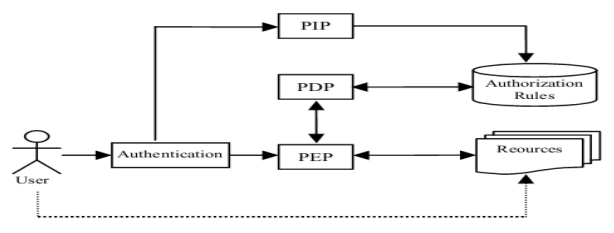


Figure 3.1 Access Control Mechanism Components.

**Authentication:** determines whether the user is anonymous or authenticated.

**PIP:** Policy Information Point which servers as the retrieval source of data required for policy evaluation to provide the information needed by the PDP to make decision.

**PDP**: Policy Decision Point which makes the access decisions by evaluating the applicable policies. The PDP uses Access Control Mechanism’s computational languages to implement the decision procedures.

**PEP**: Policy Enforcement Point enforces policies for authorization and policy decisions in response to a request from a subject. It is responsible for executing the appropriate access decisisons made by the PDP, which will return the data to subject if the decision is permit and a message when the decision is denied.

Over several decades, some models of access control systems have been developed as followings:

Discretionary Access Control (DAC) limits access to resource based on the identity of users or groups to which the users belong. DAC protects all system resources from unauthorized access down to a single user. A user who does not have permission to access a resource can be granted this permission by the resource’s owner.

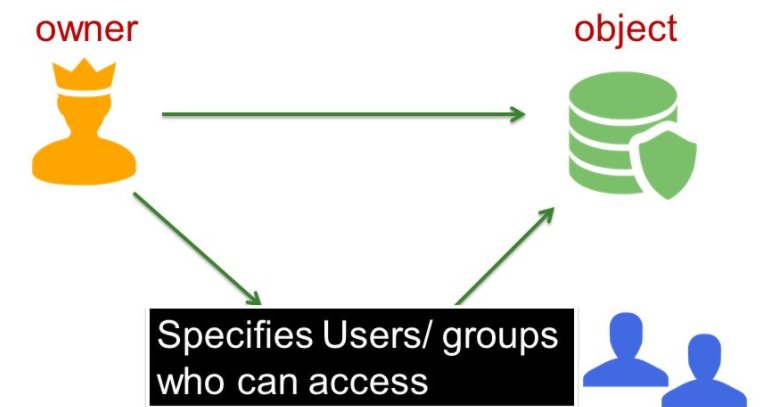


Figure 3.2 Discretionary Access Control Model.

Mandatory Access Control (MAC) limits access to resource based on label relationships. The system associates a sensitivity label with all processes that are created to execute programs. MAC policy use this label in access control decisions. In general, processes can not store information or communicate with other processes, unless the label of the destination is equal to the label of the process. MAC policy permits processes to read data from objects at the same label or from objects at a lower label. MAC policy permits processes to write data from objects at the same label or from objects at a higher label.

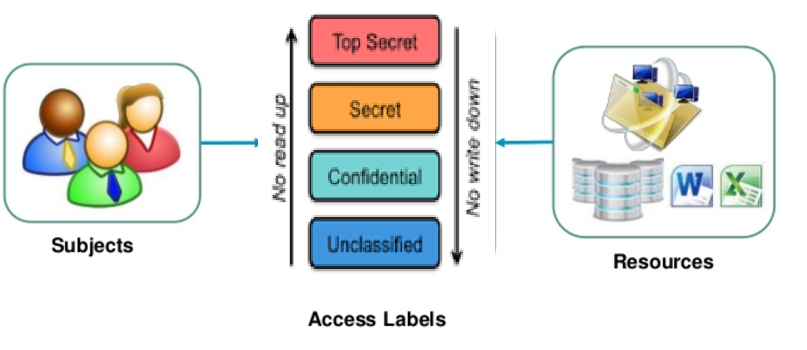


Figure 3.3 Mandatory Access Control Model.

Role Based Access Control (RBAC) limits access to resource based on user’s role within a business. All access is controlled through roles that users are given, which is a set of permissions. A user’s role determines what permission he or see is granted. There are parts to role-based access controls.

* The *permission*: The permission defines a specific operation (read, create, update or delete..) and the target entries which those operations apply. Permission are building blocks; they can be assigned to multiple privileges as needed.
* The *privileges*: A privilege is essentially a group of permissions. Permissions are not applied directly to a role. Permissions are added to a privilege so that the privilege creates a coherent and complete picture of a set of access control rules. For example, a permission can be created to edit, delete document online. Then that permission can be combined with another permission relating to managing financial services, and they can be used to create a single privilege that relates to managing financial document system.
* The *role*: A role represents a list of users who are able to perform the actions defined in the privileges.

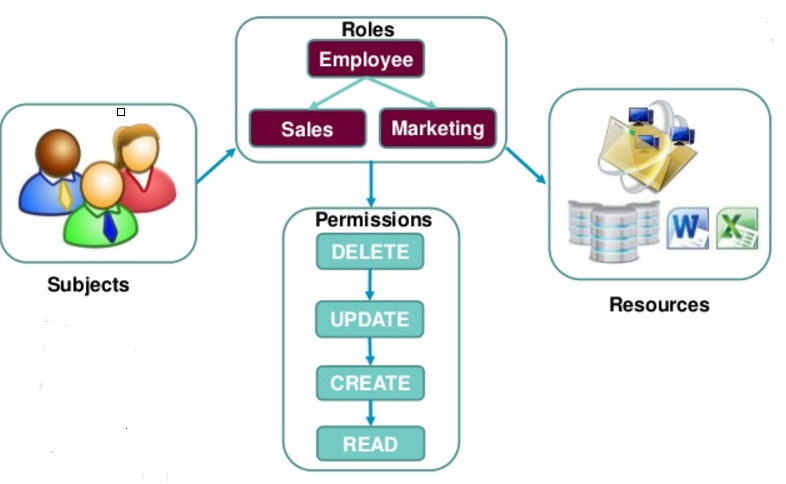


Figure 3.4 Role-Based Access Control Model.

* 1. **Attribute-Based Access Control:**
* Introduction:

Attribute-based access control is a logical access control methodology where authorization to perform a set of operations is determined by evaluating attributes associated with the subject, resource, requested operations, and, in some cases, environment conditions against policy, rules, or relationships that describe the allowable operations for a given set of attributes.

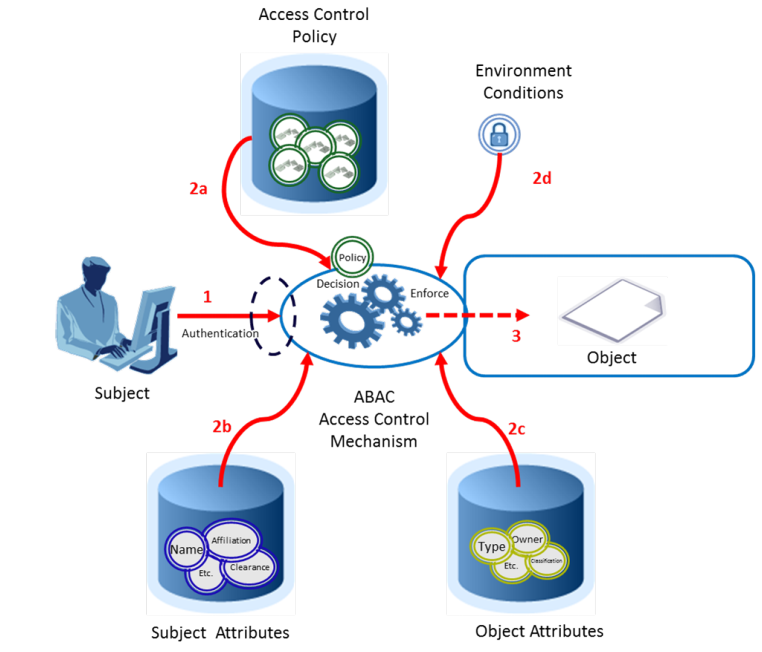


Figure 3.5 Attribute-Based Access Control Model

**Subject** is the entity (it can be the user, requestor, or mechanism acting on behalf of the user or requestor). Subjects often act on behalf of a specific human or organization. Subjects may be assigned attributes that describe their name, organization affiliation, citizenship.

**Resource** is a passive information system-related entity containing information. It can be the requested entity or anything upon which an operation may be performed by a subject including data, devices and networks.

**Action** is the operation to be carried on the resource (e.g.read, create, delete,..)

**Environment** is any information regarding the context of the access that might be used in making the access decisiion (eg.time, network, location,…).

**Policy** is the presentation of rules or relationships that defined the set of allowable operations a subject may perform upon an object in permitted environment conditions.

* 1. **Extensive Access Control Markup Language**
* Introduction:

eXtensive Access Control Markup Language also known as XACML defines a policy specification language and reference architecture for ABAC implementation. It is developed by OASIS (Organization for the Advancement of Structured Information Standards) and the newest version now is 3.0.

* Architecture:

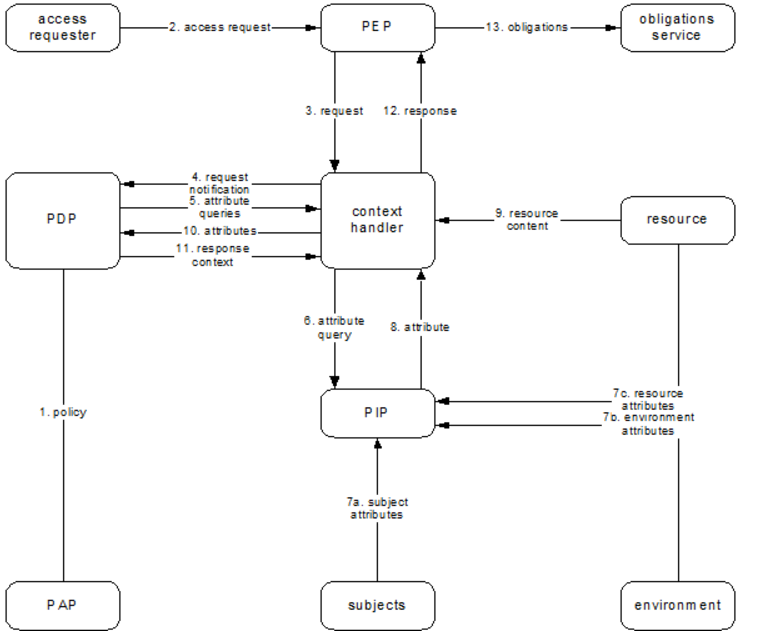


Figure 3.6 XACML Architecture

* Workflow process:

1. A user send a request which is intercepted by the PEP.
2. The PEP transfer the native request format to *context handler*.
3. *context handler* transforms it to an XACML request context and sends to PDP.
4. The PDP evaluates the request. If PDP needs any additional attributes, it will ask the *context handler* to get them from PIP.
5. The PDP returns the response context (including the authorization decision) to the context handler.
6. The *context handler* translates the response context to the native response format and returns it to PEP.
7. If access is permittedm then the PEP permits access to the resource; otherwise, it denies access.

* Policy Language Model:

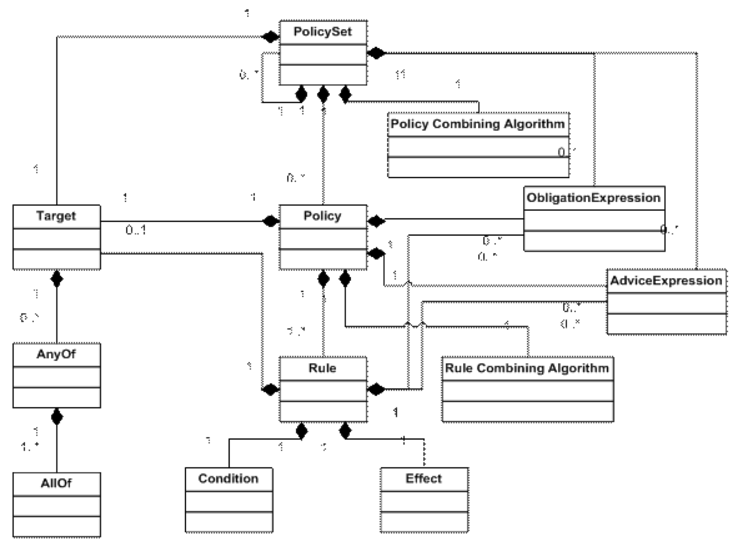


Figure 3.7 XAML Policy Model

XACML access policies are structured as PolicySets that are composed of Policies and optionally other PolicySets. Policy is composed of Rules and Targets. A Target defines a simple Boolean condition evaluated by Policy Decision Point (PDP). If no Target matches the request, the decision computed by the PDP is NotApplicable. A Rule includes a series of Boolean conditions that if evaluated True have an effect of either Permit or Deny.



Figure 3.8 Example of XACML policy

* Combining Algorithms:

Because a Policy may contain multiple Rules, and a PolicySet may contain multiple Policies or PolicySets, each Rule, Policy may evaluate to different decisions (Permit, Deny, NotApplicable). Therefore XAML provides a way of resolving conflict between these individual decisions. This approach is achieved through a collection of combining algorithms:

* *Deny-overrides*: if any decision evaluates to Deny, or no decision evaluates to Permit, then the result is Deny. If all decisions evaluate to Permit, the result is Permit.
* *Permit-overrides:* if any decision evaluates to Permit, then the result is Permit, otherwise the result is Deny.
* *First-applicable:* the result is the result of the first decision (either Permit, Deny, or Indeterminate) when evaluated in their listed order.
* *Only-one-applicable*: if only one decision applies, then the result is the result of the  
  decision, and if more than one decision applies, then the result is Indeterminate.

## Section 4. Privacy protection

Data privacy refers to the evolving relationship between technology and the legal right to, or public expectation of privacy in the collection and sharing of data about one’s self. Secure private information cannot be easily achieved by traditional access management systems because traditional access management systems focus on which user is performing what action on which data object and privacy policies are concerned with which data object is used for what purposes. Privacy concerns exist wherever uniquely identifiable data relating to a person or persons are collected and stored, in digital form or otherwise. In order to protect the privacy of individuals, a number of work has showed that the notion of purpose used for specifying privacy policies:

* Ji-Won Byun, Ninghui Li[6] have presented a comprehensive approach for privacy preserving access control based on the notion of purpose.

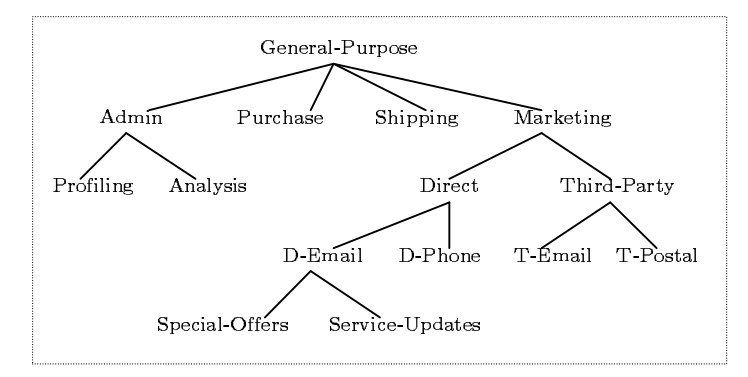


Figure 4.1 : Purpose tree

They organize a set of purposes in a tree structure. They specified an *intended purpose* which is a tuple <AIP, PIP> where AIP is called the set of allowed intended purpose, PIP is called the set of prohibited intended purpose.

For example: Suppose IP = <{Admin, Direct}, {D-Email}> is defined over the purpose tree given in figure 4.1. Using their proposed algorithm, we will have the set of purposes implied by IP which are Admin, Profiling, Analysis. They will assign IP to each data element. This approach allows multiple purposes to be associated with each data element and also supports explicit prohibitions, thus allowing privacy officers to specify that some data should not be used for certain purposes

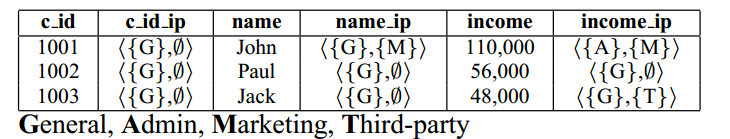


Figure 4.2 Example of applying purpose into data

However, there are two disadvantages in their approach. First, because they assign intended purpose to each data element, it result to the size of data storage will be double. Second, their approach allows a field to have only a privacy value, the value of the field should be related to purpose in practice. For example, suppose we have a SSN field, their approach can not solve the following rule: " If purpose is Marketing then SSN field will show only Area Numbers. If purpose is Third-party then SSN field will show Group Numbers and Serial Numbers".

Private or sensitive information can be preserved by restricting the intended purpose of data access. According to “Data protection principles” [8] based on Data Protection Act Organization , purpose is considered as the second principle: “Personal data shall be obtained only for one or more specified and lawful purposes, and shall not be further procecssed in any manner incompatible with that purpose or those purposes.” Extending those approaches, we have proposed an Attribute-Based Privacy Protection (ABPP) model which will use not only access purpose attribute but also user, resource, environment attributes to evaluate privacy policies so that we can specify privacy policies more dynamically and flexibly.

## Section 5. Attribute-Based Access Control and

## Privacy Protection Model

In this section, we describe the base theory of this thesis. When a subject access an object, the authorization process is carried out though two stages called as 2-stage authorization:

* First stage: access control policy authorization verifies that the request is legitimate with rights for the subject to access data.
* Second stage: request is transfer to this stage for checking privacy compliance based on privacy policies.

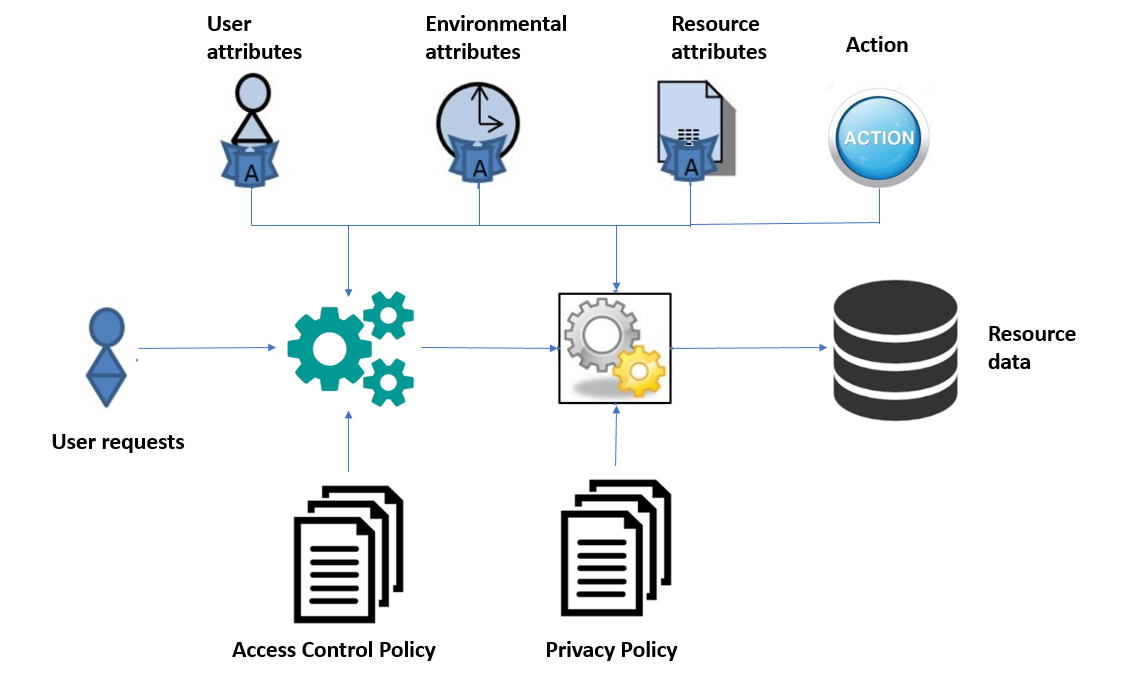


Figure 5.1 Attribute-Based Access Control and Privacy Protection Model.

**Access Control Policies**: contain policies which are used to determine whether a subject can access resources. The decision is made based on rules inside policies which are the boolean expressions evaluated by user’s defined function, subject, resource, environment attribute. Those policies are specified and managed by administrators.

**Privacy Policies**: contain policies which are used to determine whether some fields of a record in resources should be shown, hidden or blurred when a user access to it. The privacy protection is made based on rules inside policies which are the boolean expressions evaluated by user’s defined function, subject, resource, environment attribute. Those policies are specified and managed by administrators.

## Section 6. Policy Structure

* 1. **General Structure:**

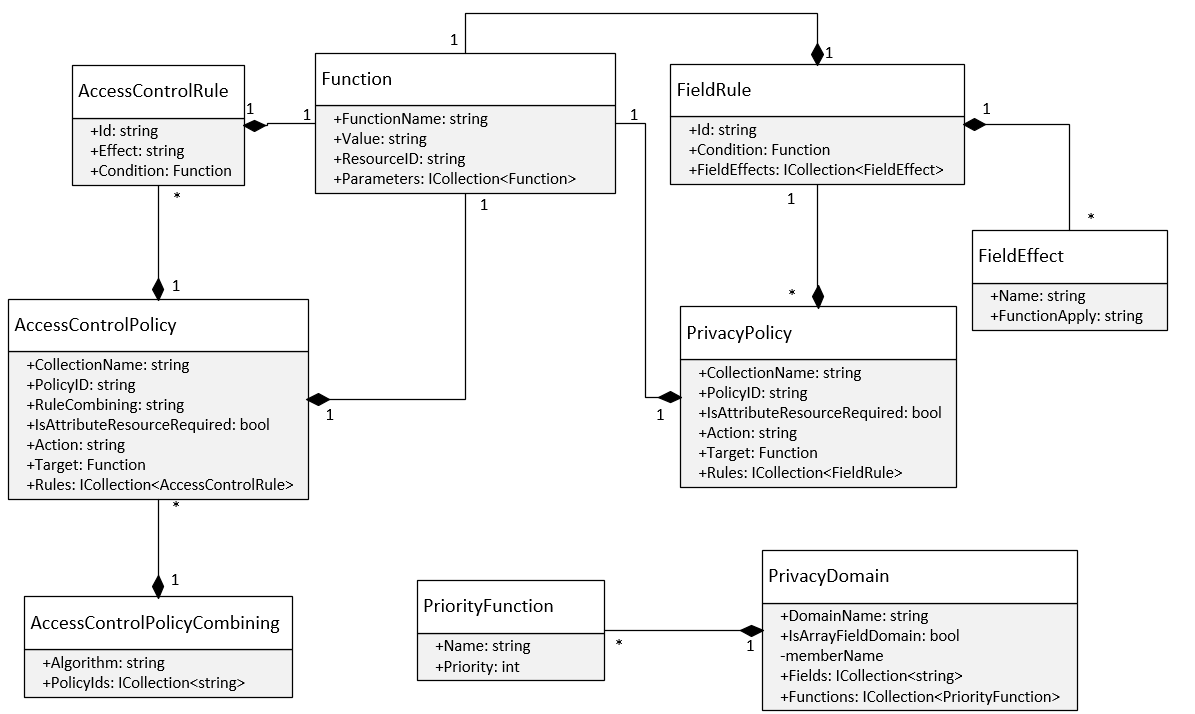


Figure 6.1 Privacy and Access Control Policy class diagram

We will explain more about how we model the conditional expression of policies in *Function* class. It is a recursive structure. If the value of “Value” field is defined so it means that this function represents for a constant value or the JSON path to get data from resource (in this case "ResourceID" field is also defined) . If the value of “Value” is null it means that this is the function and the value of "FunctionName" field must be defined.

|  |  |
| --- | --- |
| **Fields** | **Description** |
| FunctionName | The name of function. |
| Value | The constant value or a path json |
| ResourceID | The identifer or name of the resource |
| Parameters | Its value can be a constant value or another function |

Table 6.1 *Function* class description.

For example we have a conditional expression:

*Expression* = "Equal (Subject.role, intern) And GreaterThan (Subject.age, 18)"

Then the result will be:

{

"function\_name" : "And",

"parameters" : [

{

"function\_name" : "Equal",

"parameters" : [

{

"value" : "role",

"resource\_id" : "Subject"

},

{

"value" : "intern",

"resource\_id" : null

}

]

},

{

"function\_name" : "GreaterThan",

"parameters" : [

{

"value" : "age",

"resource\_id" : "Subject"

},

{

"value" : "18",

"resource\_id" : null

}

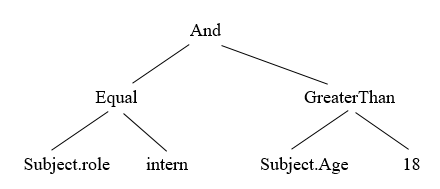
]

}

]

}

It can also be visualized in expression tree:



We also proposed an algorithm which converts the conditional expression in text format to the *Function* structure:

**Algorithm :** Algorithm for parsing conditional expression  
**Input**: *rawExpression*: string

**Output**: *function*: *Function* class

**1** Let *listToken*: List<string>**←** *getTokens*(*rawExpression*); /\* gets tokens from *rawExpression*  
**2** Let *stackToken:* Stack<string> /\* A stack stores names of functions \*/

**3** Let *queueToken:* Queue<string> /\* A queue stores tokens in Reverse Polish Notation form \*/

**4** Let *stackFunction:* Stack<Function> /\* A stack stores functions \*/

**5 begin**

**6 foreach** *token* **in** *listToken* **do**

**7 if** *IsFunctionName***(** *token )* **Or** *IsLogicalOperator***(** *token )* **Or** *token ==* "("**then**

**8** *stackToken*.**push** (*token*)

**9 else if** *token ==* ")"**then**

**10 while** *stackToken*.length > 0 **do**

**11 var** *temp* :*= stackToken*.pop()

**12 if** *temp ==* "("**then**

**13** *queueToken***.**enqueue**(***stackToken*.pop())

**14 *break***

**15 else** *queueToken***.**enqueue**(***temp*)

**16 end while**

**17 else** *queueToken.*enqueue (*token*)

**18 end foreach**

**19 while** *stackToken.*length > 0 **do** *queueToken.*enqueue (*stackToken.*pop())

**20 while** *queueToken.*length > 0 **do**

**21** *token* := *queueToken*.dequeue()

**22 if** *IsFunctionName***(** *token )* **do**

**23 var** *function* := Function.CreateFunction (token)

**24 for** i := 1 to *GetNumberParameters* (*function*) **do**

**25** *function.*Parameters.AddToFirstElement (*queueFunction.*dequeue())

**26 end for**

**27** *stackFunction*.push (function)

**28 else** *stackFunction*.push (Function.CreateConstantValue(*token*))

**29 end while**

**30 return** *stackFunction.*pop()

**31 end**

Example: *Expression* = "Equal (Subject.role, intern) And GreaterThan (Subject.age, 18)"

First we will split the expression into tokens:

*listToken* = {Equal, (, Subject.role, intern, ), And, GreaterThan, (, Subject.age, 18, ) }

|  |  |  |
| --- | --- | --- |
| **Token** | **Stack** | **Queue** |
| Equal | Equal |  |
| ( | Equal ( |  |
| Subject.role | Equal ( | Subject.role |
| intern | Equal ( | Subject.role intern |
| ) |  | Subject.role intern Equal |
| And | And | Subject.role intern Equal |
| GreaterThan | And GreaterThan | Subject.role intern Equal |
| ( | And GreaterThan ( | Subject.role intern Equal |
| Subject.age | And GreaterThan ( | Subject.role intern Equal Subject.age |
| 18 | And GreaterThan ( | Subject.role intern Equal Subject.age 18 |
| ) | And | Subject.role intern Equal Subject.age 18 GreaterThan |
|  |  | Subject.role intern Equal Subject.age 18 GreaterThan And |

Table 6.2 Example of running algorithm.

Then we will parse the token queue into *Function* structure, the process is like building an expression tree with bottom-up approach:

*queueFunction* = { }

Dequeue the token queue untill it is empty:

*- Subject.role* we will parse it to *Function* structure and enqueue it to *queueFunction*:

*queueFunction* = { {"value": "role", "resource\_id": "Subject"} }

*- intern* we will do the same as *Subject.role*

*queueFunction* = { {"value": "role", "resource\_id": "Subject"},

{"value": "intern", "resource\_id": null} }

- *Equal*  because it is a name of function so its *Function* structure will be

{

"function\_name" : "Equal",

"parameters" : [ ]

}

Then we will dequeue the *queueFunction* based on the number of parameters of this function and add those elements to "parameters" field. Next we will enqueue this new element to *queueFunction.*

{

"function\_name" : "Equal",

"parameters" : [

{

"value" : "role",

"resource\_id" : "Subject"

},

{

"value" : "intern",

"resource\_id" : null

}

]

}

Continue with the remaining elements, we will have the same as the following result:

{

"function\_name" : "And",

"parameters" : [

{

"function\_name" : "Equal",

"parameters" : [

{ "value" : "role", "resource\_id" : "Subject" },

{ "value" : "intern", "resource\_id" : null }

]

},

{

"function\_name" : "GreaterThan",

"parameters" : [

{ "value" : "age", "resource\_id" : "Subject" },

{ "value" : "18", "resource\_id" : null }

]

}

]

}

* 1. **Access Control Policy Structure:**

We specify access control policy structure as follows:

|  |  |
| --- | --- |
| **Fields** | **Description** |
| policy\_id | identifier of policy |
| collection\_name | name of collection or table containing resource data |
| action | the action performed by subject |
| rule\_combining | to solve the conflict of rules |
| is\_attribute\_resource\_required | a derived field which is used to determine that whether the policy need attribute resource to evaluate condition of target or rules. The necessary of this field will be mentioned in the next section. |
| target | The conditional expression specifies when the policy should be applied to. |
| rules | an array field with each element in it is a rule which contains “id” field, “effect” field (value of this field can only be “Permit” or “Deny”) and condition. |

Table 6.3 *AccessControlPolicy* class description

Example:

{

"policy\_id" : "Policy 1",

"collection\_name" : "Department",

"action" : "read",

"rule\_combining" : "permit-overrides",

"is\_attribute\_resource\_required" : true,

"target" : { //Equal (Subject.active, True)

"function\_name" : "Equal",

"parameters" : [

{

"value" : "active",

"resource\_id" : "Subject"

},

{

"value" : "True",

"resource\_id" : null

}

]

},

"rules" : [

{

"id" : "rule 1",

"effect" : "Permit",

"condition" : {

"function\_name" : "Equal",

"parameters" : [

{

"value" : "dept\_name",

"resource\_id" : "Resource"

},

{

"value" : "department",

"resource\_id" : "Subject"

}

]

}

}

]

}

* 1. **Privacy Policy Structure:**

We specify privacy policy structure as follows:

|  |  |
| --- | --- |
| **Fields** | **Description** |
| policy\_id | identifier of policy |
| collection\_name | name of collection or table containing resource data |
| is\_attribute\_resource\_required | a derived field which is used to determine that whether the policy need attribute resource to evaluate condition of target or rules. The necessary of this field will be mentioned in the next section. |
| Target | The conditional expression specifies when the policy should be applied to. |
| Rules | an array field with each element in it is a rule which contains “id” field, “field\_effects” field and condition. “field\_effects” field is an array field with each element specifies which privacy function will be used with the value of “name” field when condition is satisfied. |

Table 6.4 *PrivacyPolicy* class description

Example:

{

"collection\_name" : "Department",

"policy\_id" : "policy 2",

"action" : "read",

"is\_attribute\_resource\_required" : true,

"target" : {

"function\_name" : "Equal",

"parameters" : [

{

"value" : "role",

"resource\_id" : "Subject"

},

{

"value" : "intern",

"resource\_id" : null

}

]

},

"rules" : [

{

"rule\_id" : "rule 1",

"condition" : {

"function\_name" : "Equal",

"parameters" : [

{

"value" : "dept\_name",

"resource\_id" : "Resource"

},

{

"value" : "OPERATIONS",

"resource\_id" : null

}

]

},

"field\_effects" : [

{

"name" : "dept\_id",

"effect\_function" : "DefaultDomainPrivacy.Hide"

},

{

"name" : "dept\_no",

"effect\_function" : "DefaultDomainPrivacy.Show"

},

{

"name" : "dept\_name",

"effect\_function" : "DefaultDomainPrivacy.Show"

}

]

}

]

}

* 1. **Conflict resolving approach:**

Because an access control policy may contain multiple rules and we have many policies. Each rule, access control policy may evaluate to different decisions (Permit, Deny). Therefore our approach is to use combining rule algorithm inherited from XACML[9].

A new structure named “AccessControlPolicyCombiningConfiguration” is added to resolve conflict between multiple access control policies.

|  |  |
| --- | --- |
| **Fields** | **Description** |
| policies\_id | A list identifiers of policies |
| algorithm | The name of algorithm is used to solve conflict when multiple policies are contained in “policies\_id” field |

Table 6.5 *AccessControlPolicyCombiningConfiguration* class description

Example:

{

"\_id" : "58f24565de2b68f43464287a",

"policies\_id" : [

"Policy 1", "Policy 2"

],

"algorithm" : "permit-overrides"

}

In privacy policies, the conflict situation occurred when there are multiple rules in a privacy policy which are satisfied the condition. It results to that we have many privacy functions to be applied to one field of object. Therefore we have added a new structure named “PrivacyDomain” to solve conflict. We also specify a constraint that a field in resource can only belong to at most two domains, one is default domain which contains two basic privacy functions: show and hide, another domain is configured by administrator.

|  |  |
| --- | --- |
| **Fields** | **Description** |
| domain\_name | Name of domain. |
| fields | The names of fields in resource which are belong to this domain. |
| hierarchy | To configurate the priority for each privacy function or sub-privacy policy. |
| is\_sub\_policy | To check whether this is domain for privacy function or sub-privacy policy. |

Table 6.6 *PrivacyDomain* class description

Example:

{

"domain\_name" : "DefaultDomain ",

"fields" : [],

"is\_sub\_policy" : false,

"hierarchy" : [

{

"name" : "Hide",

"priority" : 1

},

{

"name" : "Show",

"priority" : 2

}

]

},

Let consider below example to see how conflict resolving process work:

Employee Resource:

{

"name": "John",

"personal\_info": {

"birth\_date": "15/01/1994",

"ssn": "457-55-5462"

}

}

Privacy policy:

{

"policy\_id": "policy 1",

….

"rules" : [

{

"rule\_id" : "rule 1",

"field\_effects" : [

{

"name" : "name",

"effect\_function" : "Optional"

},

{

"name" : "personal\_info.birth\_date",

"effect\_function" : "DateTimeDomain.ShowYear"

},

{

"name" : "personal\_info.ssn",

"effect\_function" : "SsnDomain.SerialNumber"

}

],

"condition" : {//assume that this condition is satisfied}

},

{

"rule\_id" : "rule 2",

"field\_effects" : [

{

"name" : "name",

"effect\_function" : "DefaultDomain.Show"

},

{

"name" : "personal\_info.birth\_date",

"effect\_function": "DateTimeDomain.ShowMonthAndYear"

},

{

"name" : "personal\_info.ssn",

"effect\_function" : "SsnDomain.AreaNumber"

}

],

"condition" : {//assume that this condition is satisfied }

},

{

"rule\_id" : "rule 3",

"field\_effects" : [

{

"name" : "name",

"effect\_function" : "DefaultDomain.Show"

},

{

"name" : "personal\_info.birth\_date",

"effect\_function" : "DefaultDomain.Show"

},

{

"name" : "personal\_info.ssn",

"effect\_function" : "Optional"

}

],

"condition" : {//assume that this condition is satisfied }

}

]

}

Privacy Domain:

{

"domain\_name" : "DateTimeDomain",

"fields" : ["Employee.personal\_info.birth\_date"],

"is\_sub\_policy" : false,

"hierarchy" : [

{

"name" : "ShowYear",

"priority" : 1

},

{

"name" : "ShowMonthAndYear",

"priority" : 2

}

]

},

{

"domain\_name" : "SsnDomain",

"fields" : ["Employee.personal\_info.ssn"],

"is\_sub\_policy" : false,

"hierarchy" : [

{

"name" : "AreaNumber",

"priority" : 1

},

{

"name" : "GroupNumber",

"priority" : 2

},

{

"name" : "SerialNumber",

"priority" : 3

}

]

}

First we will explain more detail about the “field\_effects” field in privacy policy structure. It is an array field and the number of elements in this field is equal to the number of single value field in resource. Each element has the following structure:

"name": is the path to the single value field.

"effect\_function": This field has only 2 value patterns. First is "Optional" value, second is "X.Y" value where X is privacy domain, and Y is name of privacy function in that domain.

Back to the example, we have the conflict privacy table.

|  |  |
| --- | --- |
| **Fields** | **Conflict Privacy Functions** |
| name | Optional, DefaultDomain.Show |
| personal\_info.birth\_date | DateTimeDomain.ShowMonthAndYear,  DateTimeDomain.ShowYear,  DefaultDomainPrivacy.Show |
| personal\_info.ssn | SsnDomain.AreaNumber, SsnDomain.SerialNumber, Optional |

Table 6.7 Example of Conflict Privacy Functions

The privacy function will be chosen using the following rule:

P(“Optional”) < P(“DefaultDomain.Show”) < P(X.Y1) < … < P(X.Yn) < P (“DefaultDomain.Hide”)

where P(X.Y) stands for priority of privacy function Y in domain X. The priority is configured by admininstrator in “PrivacyDomain” structure.

Applying this rule to above conflict table, we will have the following result:

|  |  |
| --- | --- |
| **Fields** | **Privacy Function Chosen** |
| name | DefaultDomain.Show |
| personal\_info.birth\_date | DateTimeDomain.ShowYear |
| personal\_info.ssn | SsnDomain.AreaNumber. |

Table 6.8 Result of solving conflict between privacy functions

Applying those privacy function chosen, the result data will be:

{

"name": "John",

"personal\_info": {

"birth\_date": "1994",

"ssn": "457"

}

}

* 1. **Advanced privacy policy support for array of embedded subdocuments:**

JSON object can contain not only single-value field (string, number, boolean value) but also contain an array of objects field (it’s also known as array of embedded subdocuments in mongoDB) which makes the JSON object more complex. To support privacy for array of objects field, we extend the value of "effect\_function" field in Privacy Policy structure so that its value can be an identifer of another privacy policy which target to this array field.

For example:

A complex JSON Object: "Project Bonus" Resource:

{

"project\_name" : "E-learning system",

"bonuses": [ //this is array of embedded subdocuments

{ "name": "John", "bonus": 600 },

{ "name": "Bob", "bonus": 90 }

]

}

We have a privacy rule: "Employee can only view his/her bonus for each project. "The equivalent sub-privacy policy will be specified as followings:

{

"policy\_id": "Policy 3",

**"collection\_name" : "ProjectBonus.bonuses",**

….

"rules" : [

{

"rule\_id" : "rule 1",

"field\_effects" : [

{

"name" : "name",

"effect\_function" : "DefaultDomain.Show"

},

{

"name" : "bonus",

"effect\_function" : " DefaultDomain.Hide"

},

],

"condition" : {

"function\_name" : "Equal",

"parameters" : [

{

"value" : "name",

"resource\_id" : "Subject"

},

{

"value" : "name",

"resource\_id" : "Resource"

}

]

}

},

]}

Then the main privacy policy for "ProjectBonus" collection will use the value of "policy\_id" of this sub-privacy policy for the "effect\_function" field:

{

"policy\_id": "Policy 1",

**"collection\_name" : "ProjectBonus",**

….

"rules" : [

{

"rule\_id" : "rule 1",

"field\_effects" : [

{

"name" : " project\_name",

"effect\_function" : "DefaultDomain.Show"

},

{

**"name" : "bonuses",**

**"effect\_function" : " Policy 3"**

},

],

"condition" : {//assume that this condition is satisfied }

},

]

}

Now, assuming that John want to view "ProjectBonus" collection, the result will be:

{

"project\_name" : "E-learning system",

"bonuses": [

{

"name": "John",

"bonus": 600

},

{

"name": "Bob",

"bonus": ""

}

]

}

Bonus for Bob has been hidden due to sub-privacy policy.

## Section 7. Propose Framework

* 1. **Architecture:**

Framework architecture is divided into sub-components as followings:

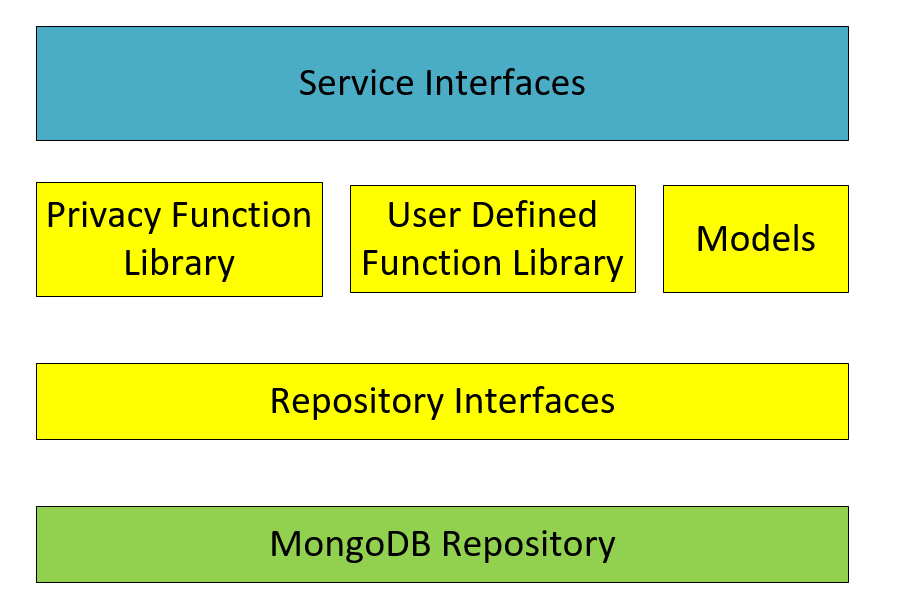


Figure 7.1 Framework Architecture

* Models

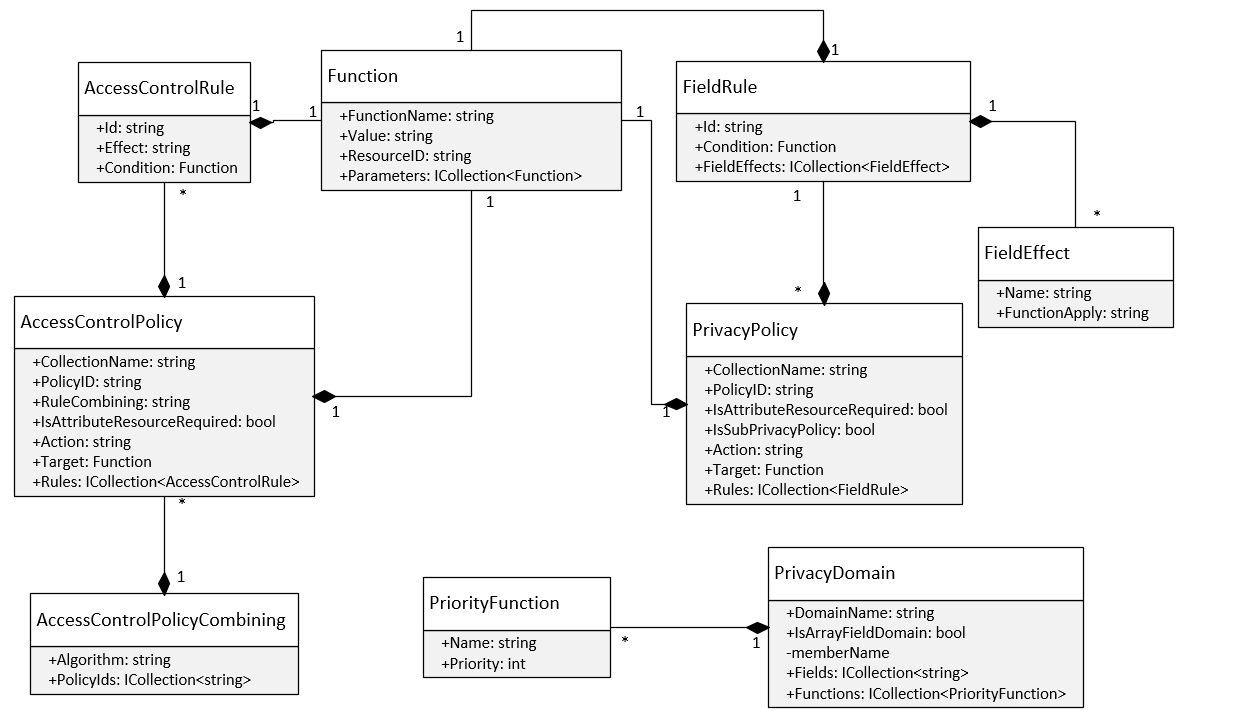


Figure 7.2 Class Diagram of *Model* Component

Model component contains entity classes which are lightweight persistence domain objects. In relational database, an entity class represents a table and each instance corresponds to a row in that table. When migrating to NoSQL database, it represents the structure of a JSON object and each instance corresponds to the value of JSON object.

* Repository Interfaces:

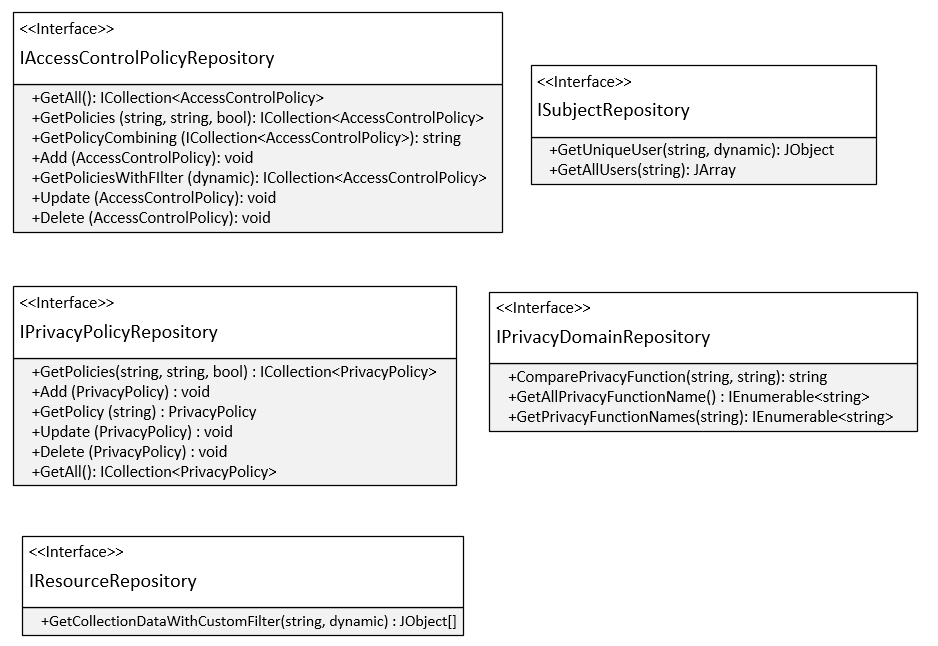


Figure 7.2 Class Diagram of *Repository Interfaces* Component

This component is designed following the Repository Pattern. Basically, it provides an abstraction of data, so that our framework can work with a simple abstraction that has an interface approximating that of a collection. Tight coupling of the database logic in the service make framework tough to test and extend further. Direct access of the data in the service logic may cause problems such as:

+ Difficulty applying Unit Test to the service logic.

+ Business logic cannot be tested without the dependencies of external systems like database.

+ Duplicate data access code throughout the business layer.

This component separates the data access logic and maps it to the models in the service component. It hides the details of data access from the service component. In other words, service component can access the data object without having the knowledge of the underlying data access architecture. In the future, underlying data sources or architecture can be changed without affecting the service component.

There are various advantages of the Repository Pattern including:

+ Service logic can be tested without need for an external source.

+ Database access logic can be tested separately.

+ No duplicate of code.

+ Caching strategy for the datasource can be centralized.

+ Centralizing the data access logic, so code maintainability is easier.

* MongoDB Repository (implemented from Repository Interfaces)

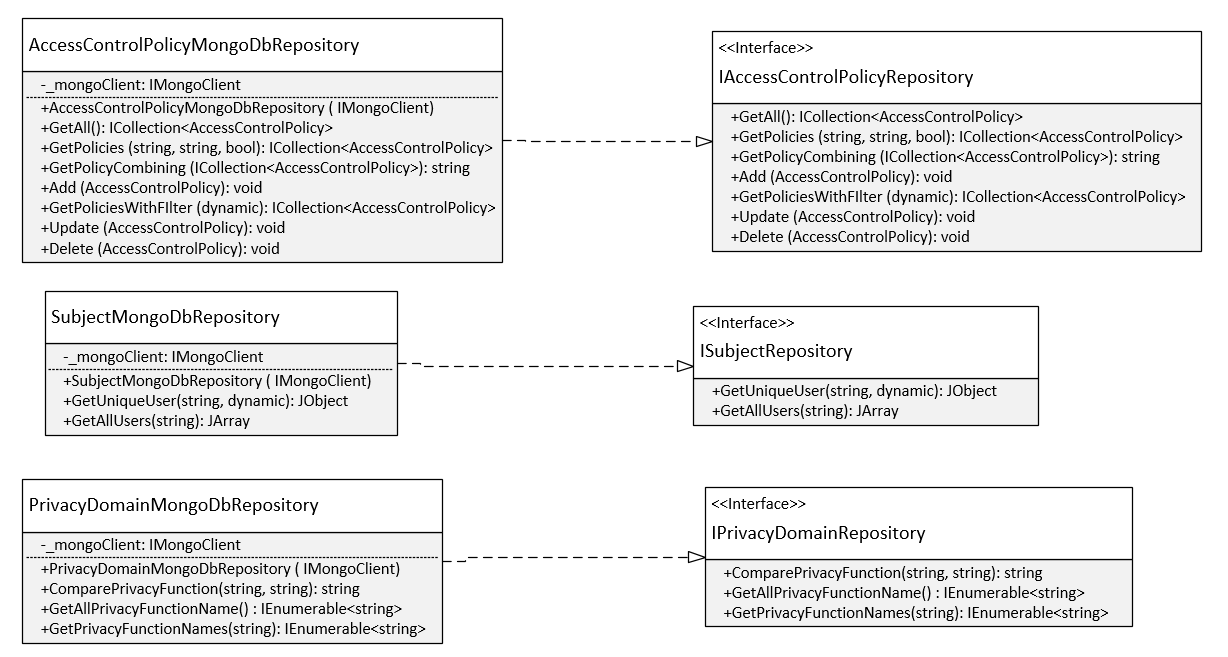




Figure 7.3 Class Diagram of *MongoDB Repository* Component

This component implements how to connect and perform CRUD operations to MongoDB. It uses another library named MongoDB.Driver which is the official MongoDB C# / .NET Driver providing asynchronous interaction with MongoDB.

* Privacy Function Library:

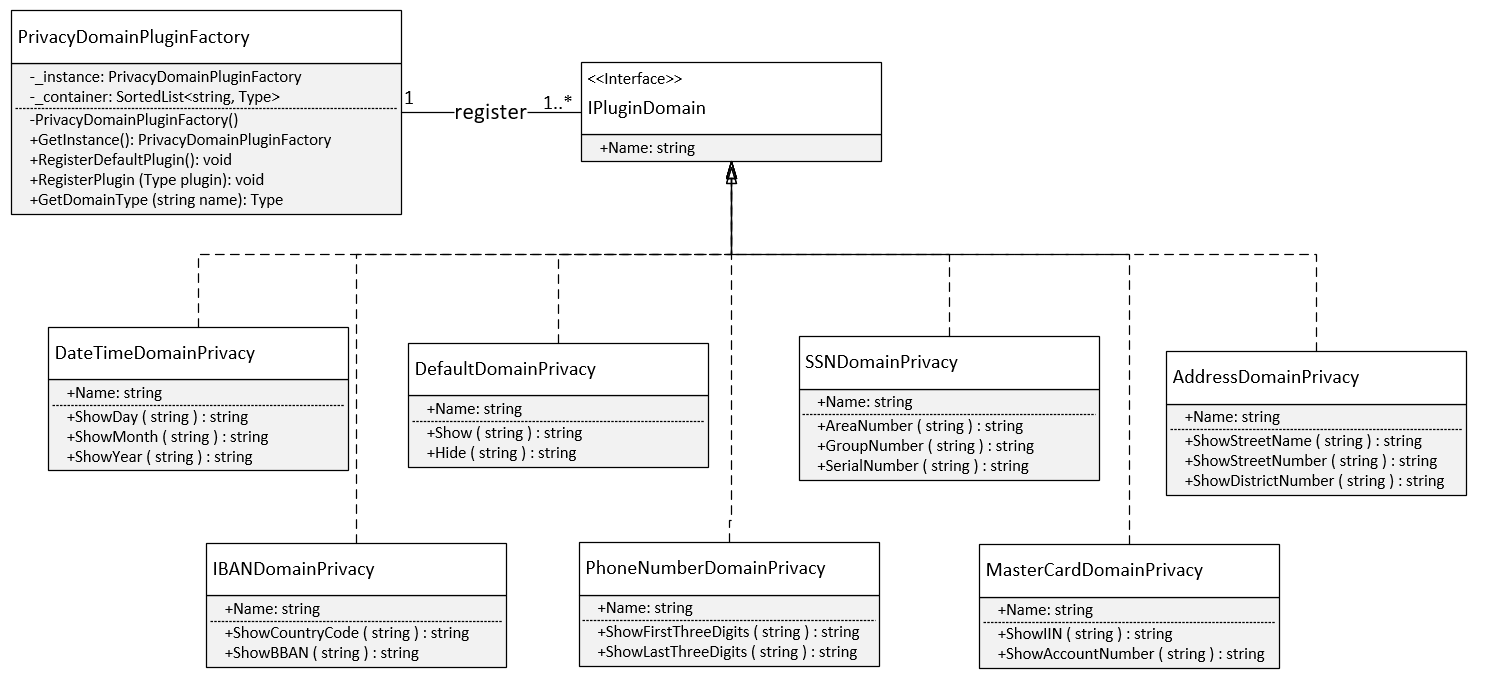


Figure 7.4 Class Diagram of *Privacy Function Library* Component

Privacy Function Library is designed based on these patterns: Singleton, Abstract Factory, Plugin Pattern. Plug-in is an extrnal piece of functionality that may be add to an existing system by abiding by a contract pre-defined by that system. The main reasons we apply this pattern in the design of Privacy Function Library:

* Easily adding new features.
* Reducing the size of library.
* Other systems can extend this library in our framework without modifying the existing source code.

*PrivacyDomainPluginFactory* class provides a level of indirection that abstracts the creation of plugins without directly specifying their concrete classes. The *PrivacyDomainPluginFactory* object has the responsibility for providing creation of registered plugins for the entire framework. The other classes never create instance of plugin directly, they ask the factory to do that for them. This mechanism makes exchanging registered plugins easy because the specific class of the factory object appears only once in the framework - where it is instantiated. Because the registered plugin provided by the factory object is so pervasive, we have implemented it with Singleton pattern. It ensures that only one instance of *PrivacyDomainPluginFactory* class is created and provides a global point of access to the object.

* User Defined Function Library:

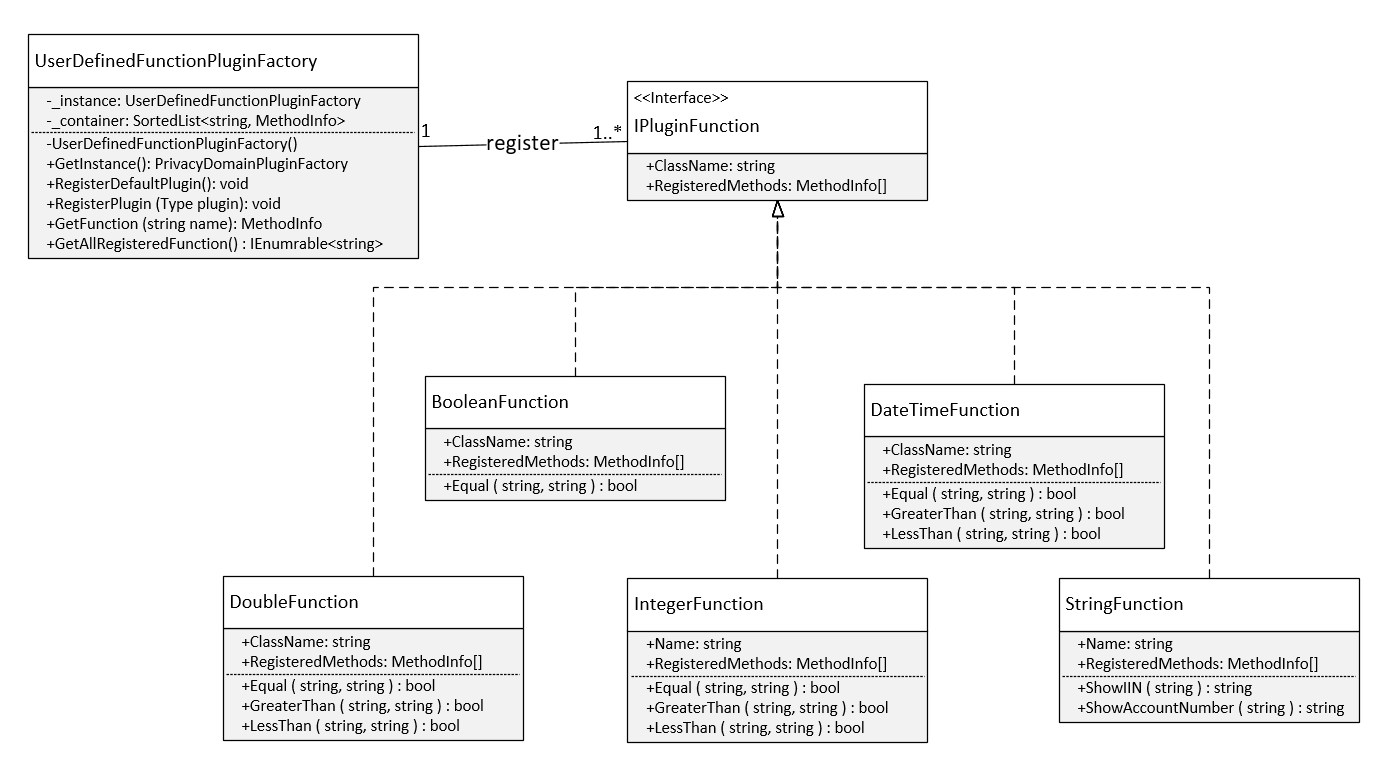


Figure 7.5 Class Diagram of *User Defiend Function Library* Component

Like Privacy Function Library, this component is designed based on Singleton, Abstract Factory, Plugin Pattern. It helps the other developers who use our framework can easily add their own function by register them in *UserDefinedFunctionPluginFactory* class. Our framework will automatically invoke registered functions when evaluating a conditional expression in a policy.

For example: A value of "target " field in a policy:

{

"function\_name" : "ExternalExampleFunction",

"parameters" : [

{

"value" : "salary",

"resource\_id" : "Subject"

},

{

"value" : "bonus",

"resource\_id" : " Subject"

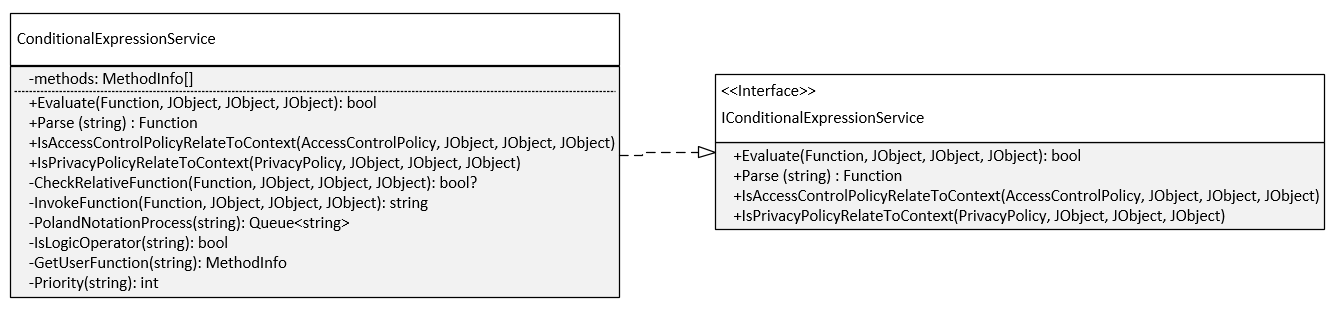
}

]

}

*UserDefinedFunctionPluginFactory* class will base on the name ("ExternalExampleFunction") to get the information of function such as: assembly name, parameter information, type returning,…then automatically invoke it with values of "salary" and "bonus" fields. To load and work with external assembly, we use System.Reflection namespace[10] which contains classes that allow developer to obtain information about the application and to dynamically add types, values, and objects to application. It allows view attribute information at run time, examming various types in an assembly and instantiate these type.

* Service Interfaces:



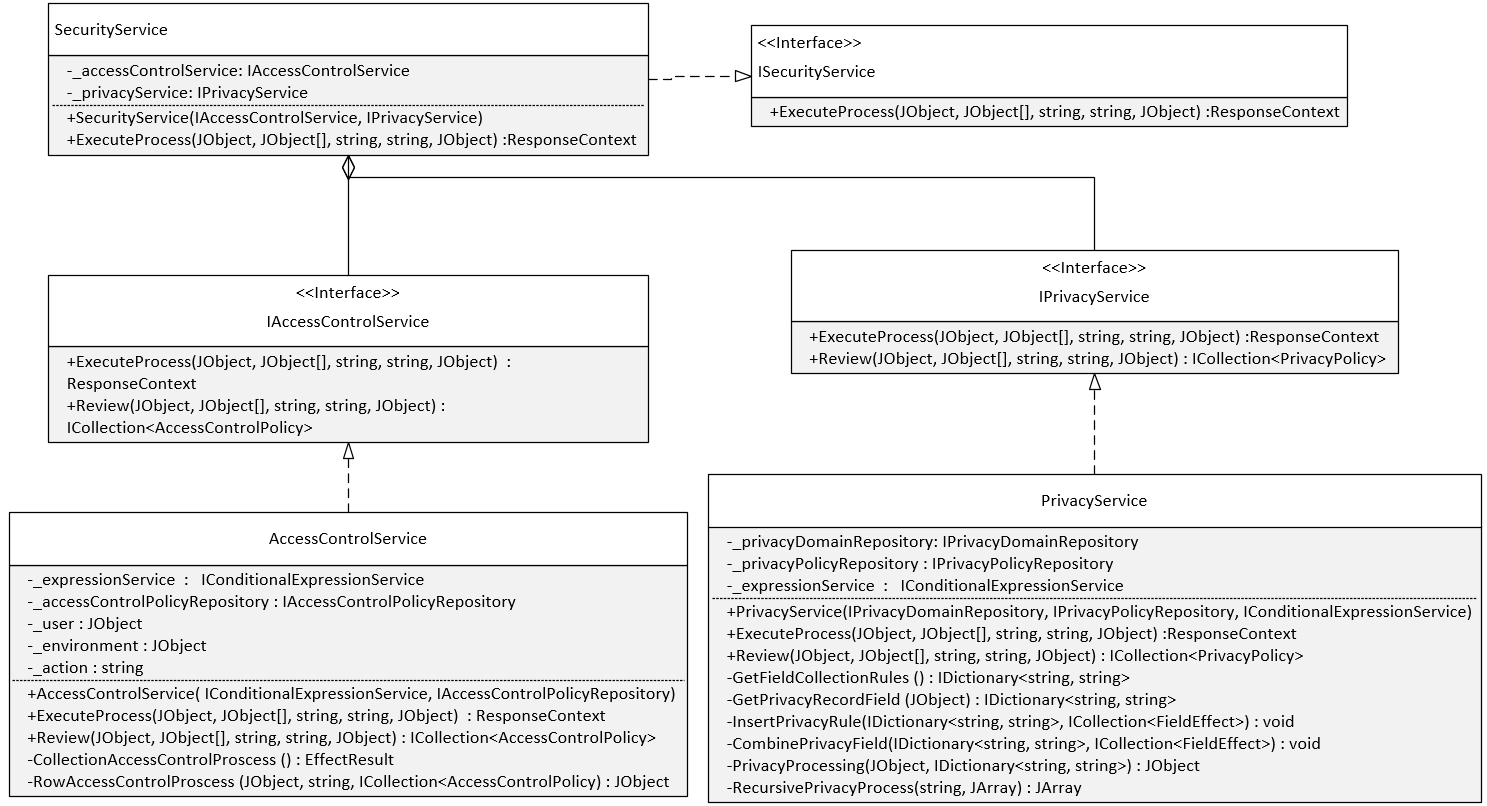


Figure 7.6 Class Diagram of *Service Interfaces* Component

This component defines our framework’s boundary that establishes a set of available operations and coordinates the framework’s response in each operation.

* *ConditionalExpressionService* class is responsible for parsing an expression in text format into an expression tree and evaluating the conditional expression.
* *AccessControlService* class is responsible for executing access control process which determines whether subject has right to access resources.
* *PrivacyService* class is responsible for protecting data privacy.
* *SecurityService* class integrates IAccessControlService interface and IPrivacyService interface to perform a 2-stage authorization. First stage verifies that the request is legitimate with rights for the subject to access data. Second stage checks privacy compliance based on privacy policies.

* 1. **Workflow:**

The main of process of our framework is described as follows:

1. *SecurityService* class receives subject, resource, action, environment attributes from the access request.
2. *SecurityService* class will invoke *IAccessControlService* interface to execute Access Control Process.
3. Access Control Process: First it will find access control policies which don’t need to use resource attributes by invoking method *GetPolicies* in *IAccessControlPolicyRepository* interface. The "is\_attribute\_resource\_required" field is used to check whether the access control policy uses resource attributes. The policies returned must be related to current context for minimizing the effort of evaluating policies. A policy is related to current context if its "target" field is evaluated to true by *IConditionalExpressionService* interface.
4. Access Control Process: After finding approriate policies, it will get the policy combining by invoke method *GetPolicyCombining* in *IAccessControlPolicyRepository* interface. Next *IConditionalExpressionService* interface will evaluate each rule in each policy to make a decision. If the decision is permit or deny, the access control process will finish and the result will be returned directly to *SecurityService* class.
5. Access Control Process: If the decision is NotApplicable, the access control process will continue to find access control policies which have to use resource attributes by invoking method *GetPolicies* in *IAccessControlPolicyRepository* interface again. Next *IConditionalExpressionService* interface will evaluate each rule in each policy to make a decision then returned it to *SecurityService* class.
6. If the decision returned from Access Control Process is permit, *SecurityService* class will continue by invoking *IPrivacyService* interface to execute Privacy Process.
7. Privacy Process: Like Access Control Process, it will find privacy policies which don’t need to use resource attributes by invoking method *GetPolicies* in *IPrivacyPolicyRepository* interface. This means that we are finding privacy policies to apply for column level. *IConditionalExpressionService* interface will evaluate each rule in each policy to determine privacy function for each field in resource.
8. Privacy Process: Next, it will find privacy policies to apply for cell level. *IConditionalExpressionService* interface will evaluate each rule in each policy to determine privacy function for each field in resource. The conflict between privacy functions of column and cell level will be solved as we mention in section 5.4. Raw resource will be processed with those privacy functions and *IPrivacyService* will return the privacy resource to *SecurityService* class.
9. *SecurityService* class creates a *ReponseContext* from the privacy resources processed by *IPrivacyService* interface and returns it.

## Section 8. Application Demonstration:

* 1. **Architecture:**

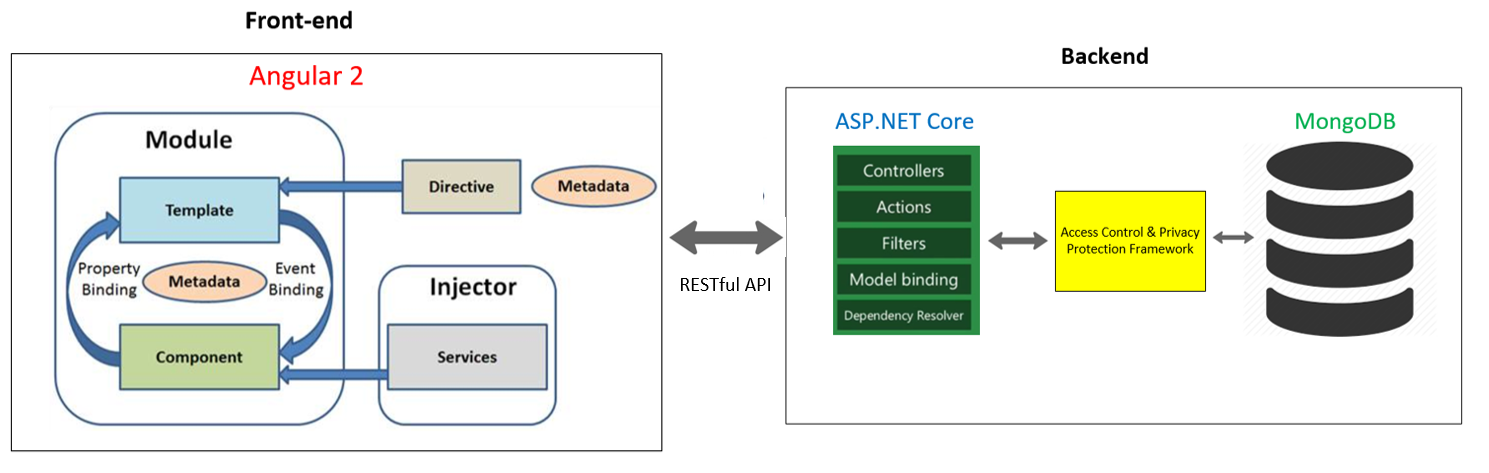


Figure 7.1 Application Architecture

* Front-end: We use Angular 2 which is a TypeScript-based open source front-end web application platform developed by Angular team at Google. Angular 2 supports single-page application which provides user-experience on our web application similar to a desktop application.
* Backend: We use ASP.NET Core which is a new open source and cross-platform framework for building modern cloud based internet connected applications, such as web apps, IoT apps and mobile backend. It was architected to provide an optimized development framework for apps that are deployed to the cloud or run on-premises. It can be developed and run cross-platform on Windows, Mac and Linux. Our framework is placed between ASP.NET Core and MongoDB database to ensure access control and privacy protection to data.
  1. **Prototype:**



Figure 8.2 Access Control Policy Form

Administrator can fill in the basic information for an access control policy like policy identifier, description, collection name,… In the *Utility* panel, administrator can choose a function name, resource field (this is the fields from the document of selected collection), subject field,… to build a conditional expression in *Target Condition* or *Current Rule.* When a rule is added, administrator can review and edit by clicking it in the table of rules below *Add Current Rule* button.

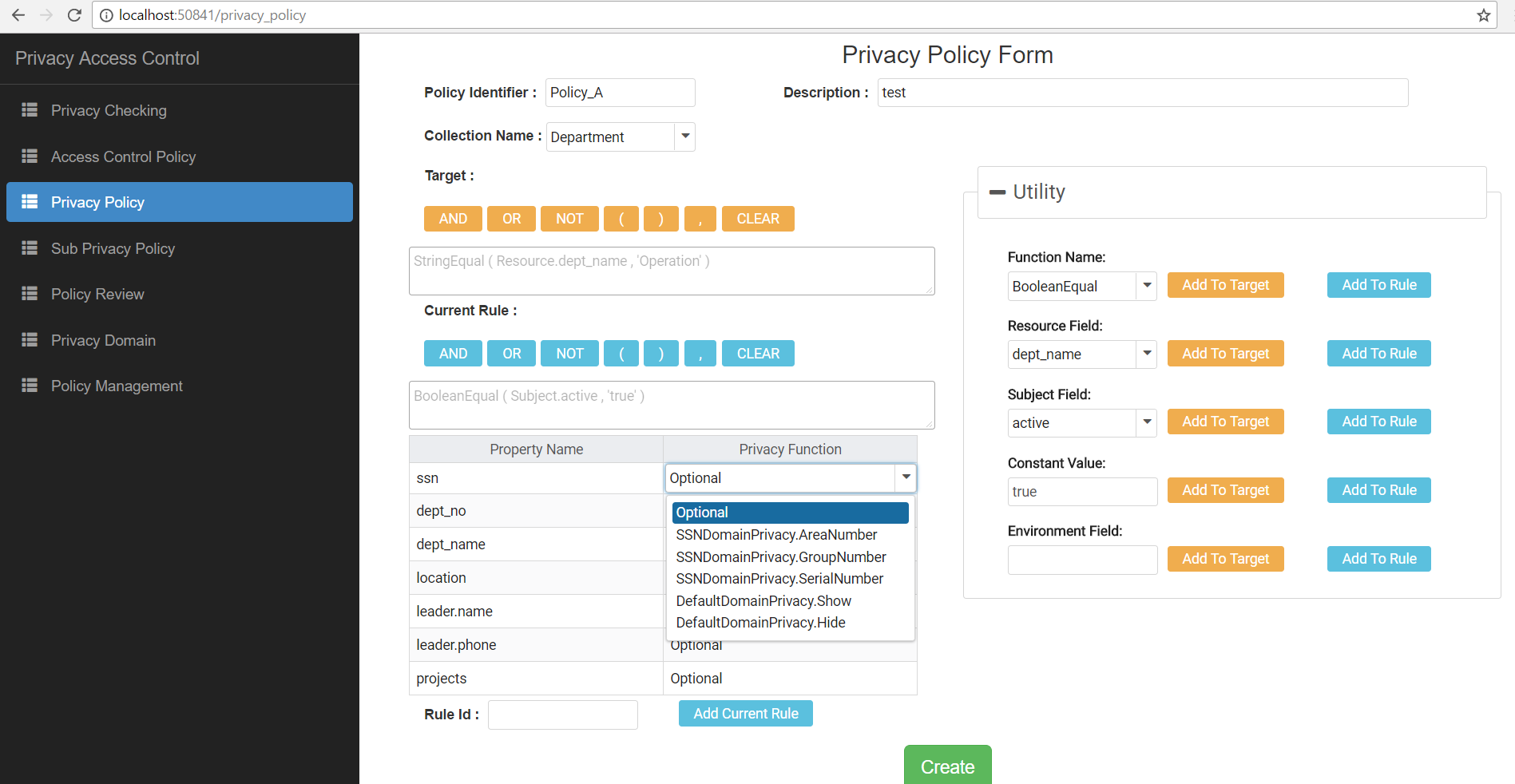


Figure 8.3 Privacy Policy Form

Administrator can fill in the basic information for a privacy policy like policy identifier, description, collection name,… In the *Utility* panel, administrator can choose a function name, resource field (this is the fields from the document of selected collection), subject field,… to build a conditional expression in *Target Condition* or *Current Rule.* Administrator can also select privacy function for each field of a document.

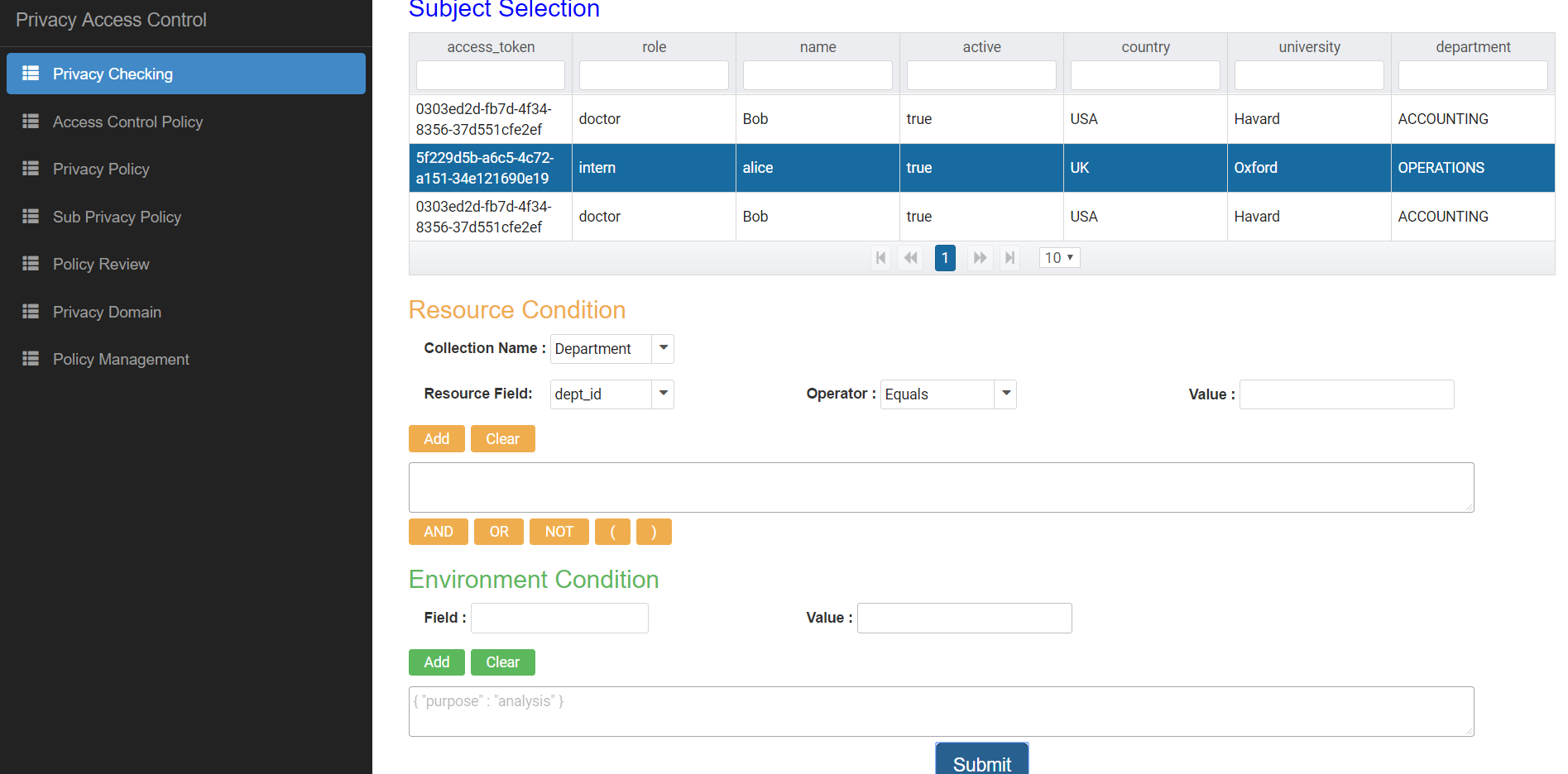


Figure 8.4 Access Control & Privacy Policy Checking Page

This page allows administrator can check what data will be returned with a subject and a collection are chosen and environment attributes are specified by administrator. The data will be shown below *Submit* button.

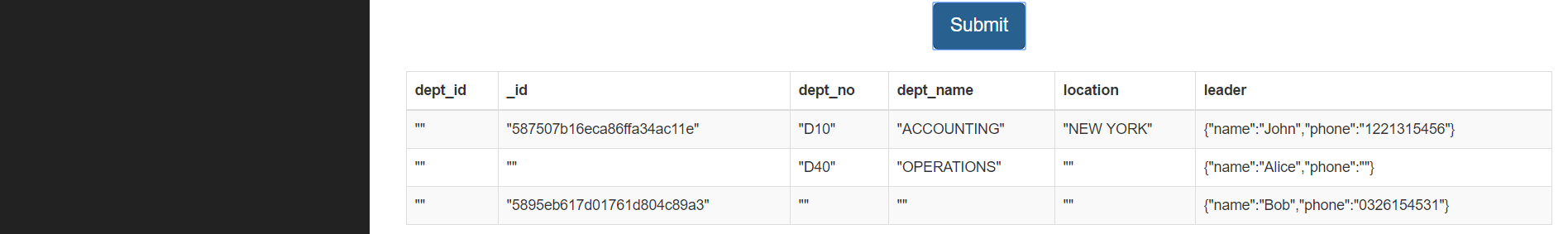


Figure 8.5 Data returned to administrator.



Figure 8.5 Sub Privacy Policy Form page.

This page allows administrator to specify privacy policy for array of embeded documents field. The *dropdown-list* in *Field Name* lists all the JSON paths to the field which is an array of embeded documents field. Administrator has to select the domain name which this sub-privacy policy belongs to. In the *Utility* panel, administrator can choose a function name, resource field, subject field,… to build a conditional expression in *Current Rule.* Administrator can also select privacy function for each field of a document.

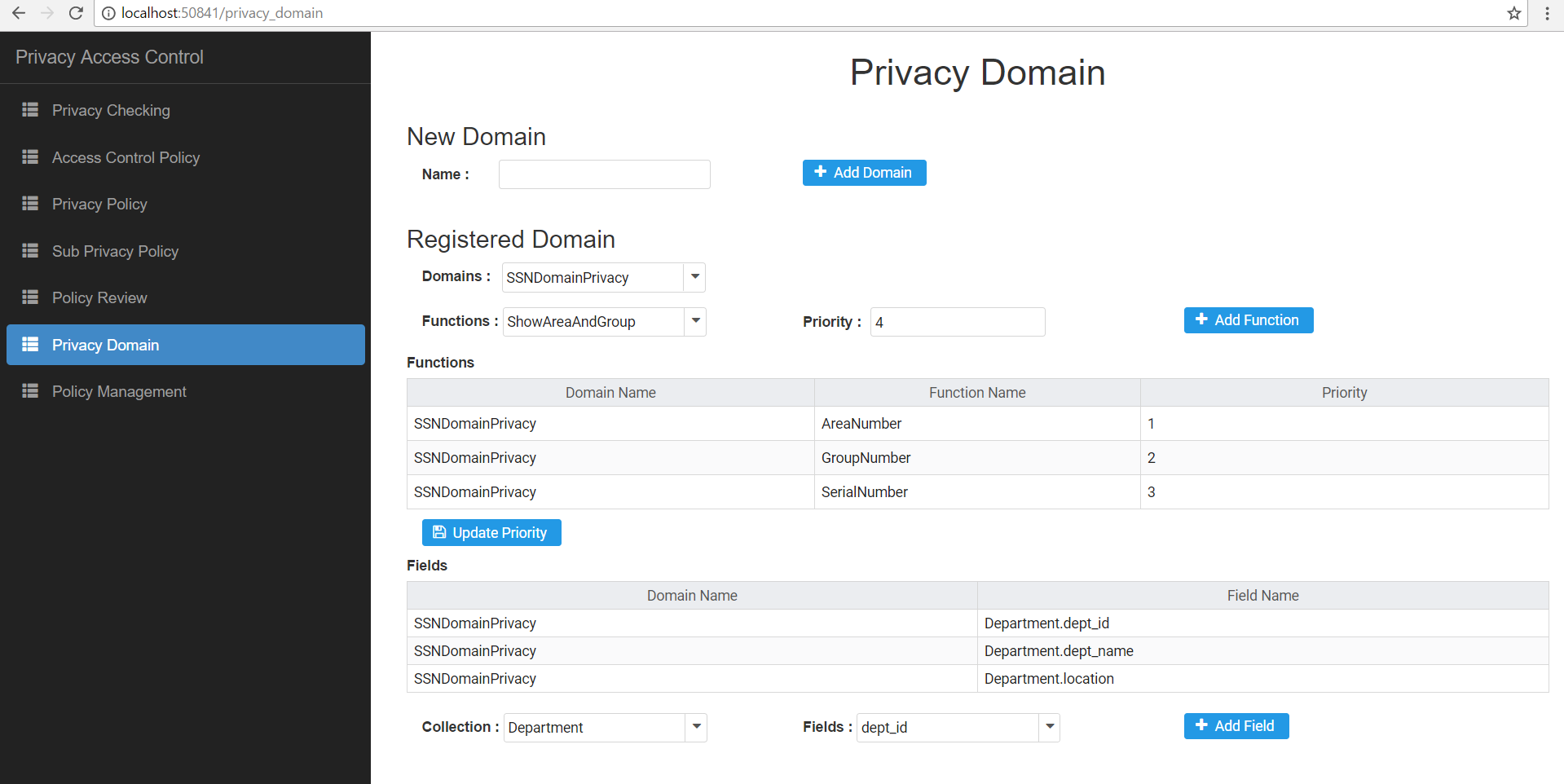


Figure 8.6 Privacy Domain page.

This pages allow administrator to add a new domain name and configurate which field is belong to which domain. Administrator can also update priority of each privacy function in a domain.

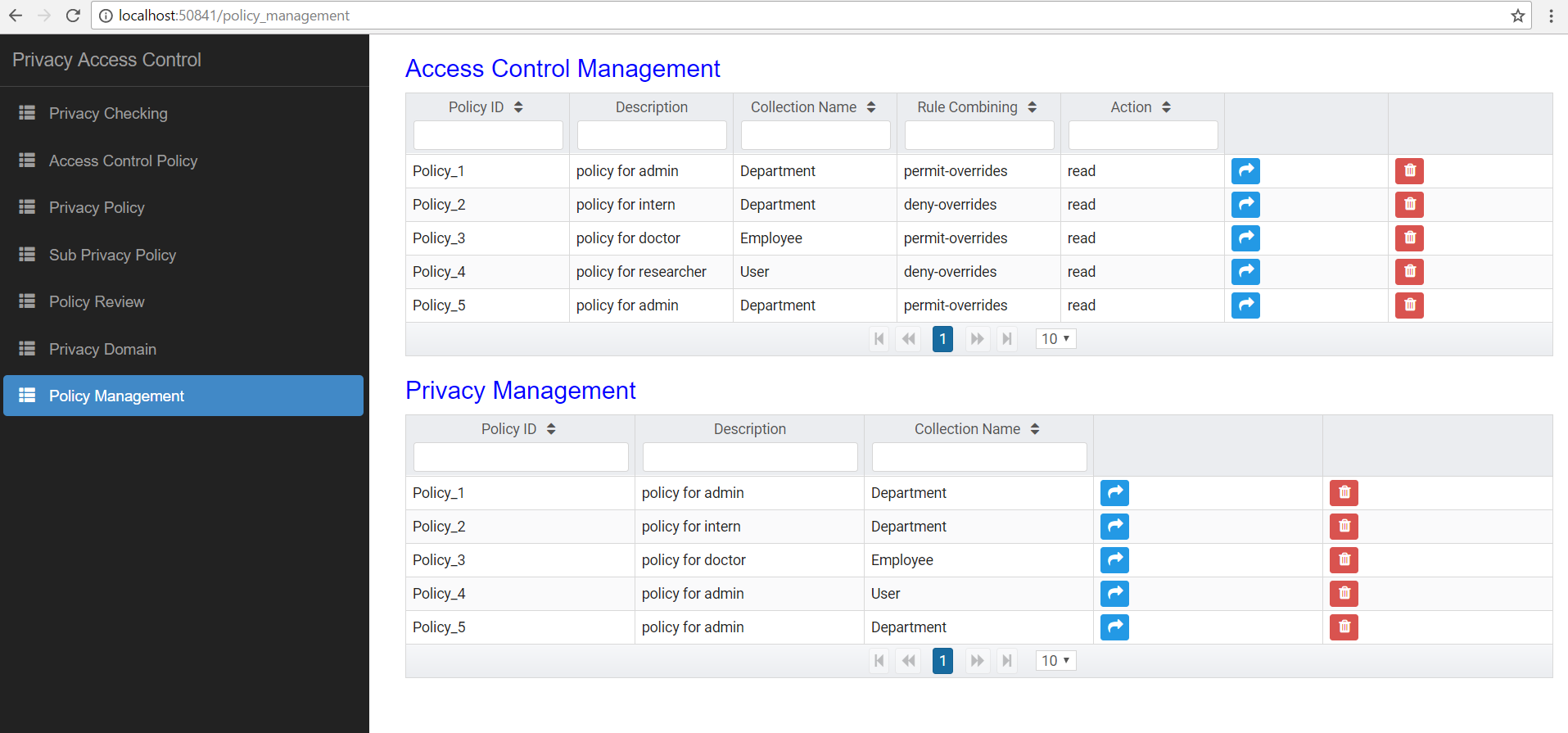


Figure 8.6 Policy Management page.

This page allows administrator to manage access control and privacy policies. Administrator can filter, delete and go to detail page for updating policy.

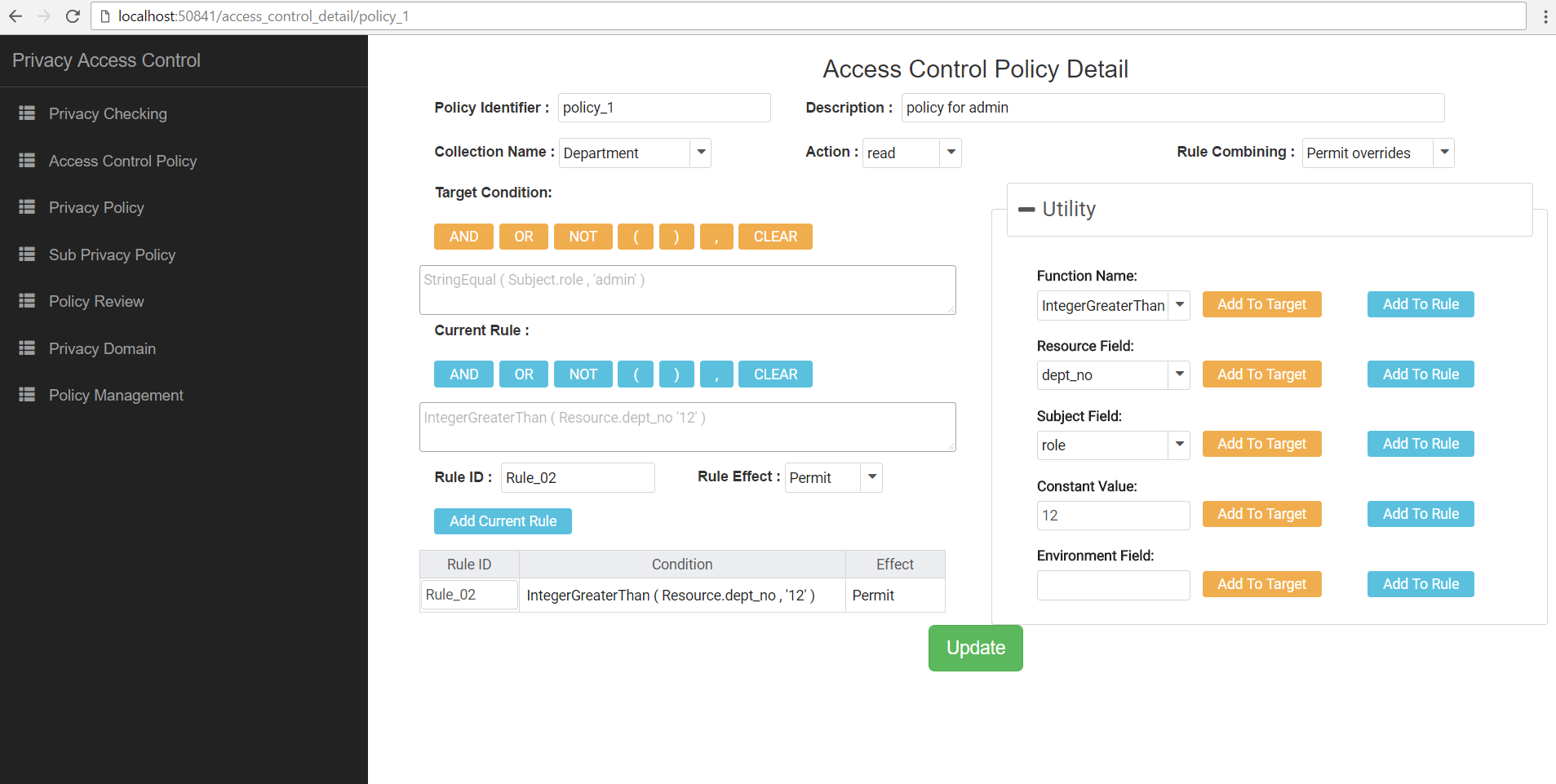


Figure 8.7 Access Control Policy Detail page.

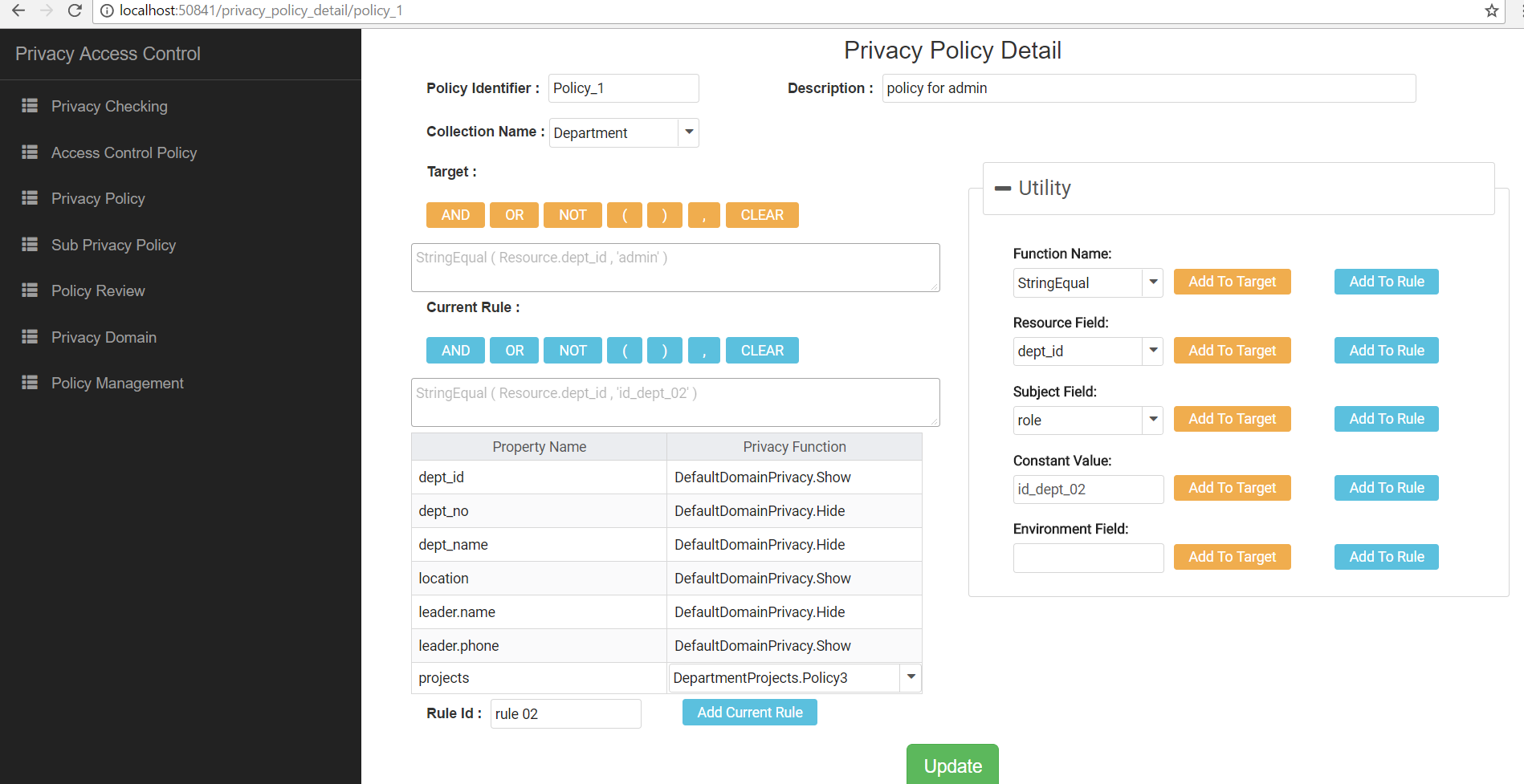


Figure 8.8 Privacy Policy Detail page.

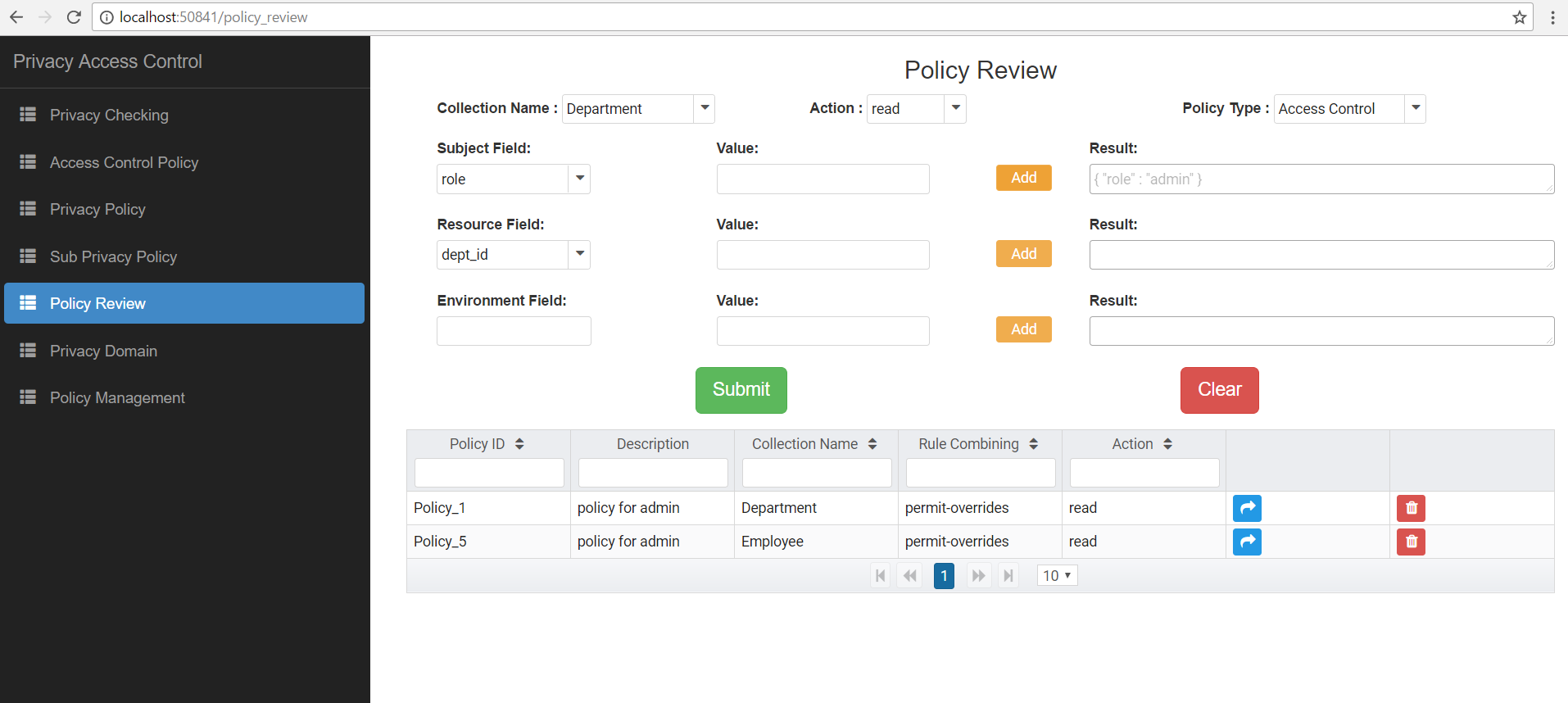


Figure 8.9 Policy Review page.

This page allows administrator to find the policy which has target or rule related to the specified condition. For example, at the above figure, administrator enter subject’s value: { "role": "admin "}and click *Submit* button, the system will find the policies which is related to this attribute and display them on the UI web.

## Section 9. Experiment

We carried out experiment about the relevance between processing time of authourization and complexity of documents. The system configuration for the experiments is Intel core i5-3230 M 2.60 GHz, 8 GB RAM. The prototype is implemented by C#, .NET Core and MongoDB for storing policies and data. All experiment include five access control polices and five privacy policies, each policy contains three rules and each rules required three attributes to evaluate. Each record in resource has to carry out thourh 2-stage authorization.

The structure of subject data we used in experiment is described as follows:

{

"first\_name" : "Raina",

"last\_name" : "Gillett",

"email" : "rgillett0@friendfeed.com",

"gender" : "Female",

"age" : 35,

"active" : true,

"job\_title" : "VP Product Management",

"department\_name" : "Business Development",

"location" : "Kenya",

"role" : "admin"

}

* First experiment: The struture of resource we used in first experiment is described as follows:

{

"date\_created" : "10/18/2014",

"is\_deleted" : false,

"tax" : 89.4,

"number\_developers" : 43,

"leader" : {

"name" : "Felicle",

"info" : {

"ssn" : "341-11-9027",

"date\_of\_birth" : "10/30/1998",

"salary" : 542.3,

"phone" : "355-(325)701-2694"

}

},

"dept\_no" : 706,

"location" : "Albania",

"name" : "Research and Development",

"long\_location" : "19.71611",

"lat\_location" : "41.06306",

"level\_domain" : "info"

}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Number records | T1(ms) | T2(ms) | T3(ms) | T4(ms) | T5(ms) | Average(ms) |
| 1 | 2000 | 2157 | 2274 | 2222 | 2189 | 2302 | **2228** |
| 2 | 4000 | 4501 | 4642 | 4644 | 4325 | 4494 | **4521** |
| 3 | 6000 | 6863 | 6690 | 6720 | 6761 | 6458 | **6698** |
| 4 | 8000 | 9022 | 8929 | 8908 | 9035 | 9010 | **8980** |
| 5 | 10000 | 11762 | 11619 | 11547 | 11273 | 11465 | **11533** |
| 6 | 12000 | 14290 | 13430 | 13643 | 13646 | 13980 | **13797** |

Table 9.1 Processing time of first experiment.

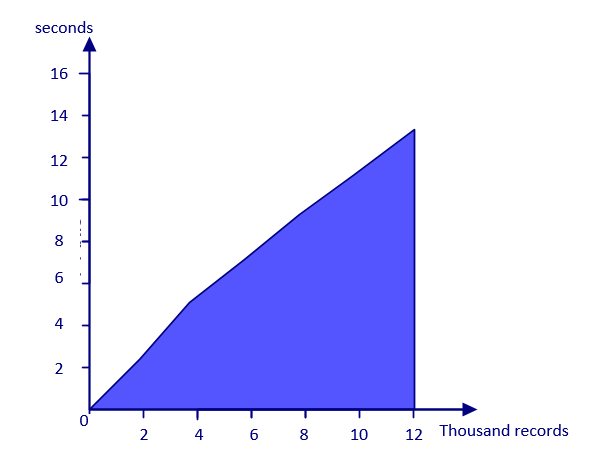


Figure 9.1 Chart of processing time in first experiment

* Second experiment: The struture of resource we used in second experiment is more complex than in first experiment which contains an array of embeded documents. Each record has an array of embeded documents field which contains at most five elements inside it.

{

"date\_created" : "3/3/2017",

"is\_deleted" : false,

"tax" : 8.52,

"number\_developers" : 51,

"leader" : {

"name" : "Kirby",

"info" : {

"ssn" : "307-18-0580",

"date\_of\_birth" : "10/13/1996",

"salary" : 709.6,

"phone" : "386-(522)612-8742"

}

},

"dept\_no" : 1269,

"location" : "Slovenia",

"name" : "Human Resources",

"long\_location" : "15.83139",

"lat\_location" : "46.57611",

"level\_domain" : "name",

"developers" : [

{

"name" : "Katti",

"info" : {

"ssn" : "605-19-2690",

"salary" : 2175,

"phone" : "1-(752)629-0495"

}

}

]

}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Number records | T1(ms) | T2(ms) | T3(ms) | T4(ms) | T5(ms) | Average(ms) |
| 1 | 2000 | 2641 | 2775 | 2708 | 2611 | 2507 | **2648** |
| 2 | 4000 | 5141 | 5270 | 5260 | 5265 | 5253 | **5237** |
| 3 | 6000 | 7879 | 7931 | 7692 | 8141 | 8176 | **7963** |
| 4 | 8000 | 10946 | 11226 | 10712 | 10768 | 10830 | **10896** |
| 5 | 10000 | 13700 | 13344 | 13581 | 13716 | 13526 | **13573** |
| 6 | 12000 | 16269 | 16483 | 16085 | 16219 | 16111 | **16233** |

Table 9.1 Processing time of second experiment.

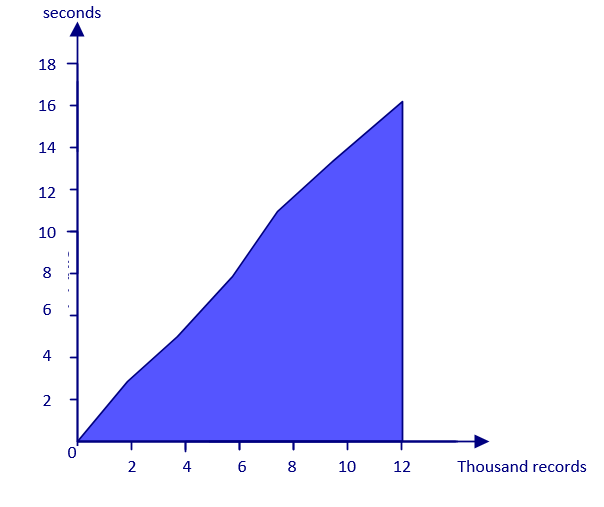


Figure 9.2 Chart of processing time in second experiment

From the result of two experiment, it indicates that the processing time increases with the complexity of data. In the first experiment, the average processing time for one record is approximately 1.1ms and in the second experiment which contains array of embeded documents is about 1.3ms per record.

REFERENCE