SAPL - Structure and Agency Policy Language

Introduction

SAPL (Structure and Agency Policy Language) describes both a **domain specific language (DSL)** for expressing access control policies and a **publish/subscribe protocol** based on JSON. Policies expressed in the SAPL policy language describe conditions for access control in applications and distributed systems. The underlying policy engine implements a variant of Attribute-based Access control (ABAC) which is centered around data streams and reactive programming patterns. Namely, the SAPL policy engine implements Attribute Stream-based Access Control (ASBAC).

A typical scenario for the application of SAPL would be a subject (e.g., a user or system) attempting to take an action (e.g., read or cancel an order) on a protected resource (e.g., a domain object of an application or a file). The subject makes a subscription request to the system (e.g., an application) for executing the action on the resource. The system implements a **policy enforcement point (PEP)** protecting the resources. The PEP collects information about the subject, action, resource, and potential other relevant data in an authorization subscription request and sends it to a **policy decision point (PDP)** which checks SAPL policies in order to decide whether access to the resource should be permitted as requested. This decision is packed in an authorization decision object and sent back to the PEP which accordingly either grants access according to the decision or prevents access to the resource. All data sources for the decision are subscribed to by the PDP and new decisions are sent to the PEP whenever indicated by the policies and data sources.

There exist several proprietary platform dependent or standardized languages, such as XACML for expressing policies. SAPL brings a number of advantages over these solutions:

- **Universality**. SAPL offers a generic, platform independent language for expressing policies. Even in complex heterogeneous environments, policies can be expressed in a common language.
- **Separation of Concerns**. Applying SAPL your domain model is relieved from modeling many aspects of access control. SAPL favors configuration at run-time over implementation and re-deployment of your applications.
- **Modularity and Distribution**. SAPL allows to manage policies in a modular fashion allowing the distribution of authoring responsibilities across teams.
- **Expressiveness**. SAPL provides access control schemata beyond the capabilities of most other practical languages. It allows for attribute-based access control (ABAC), role-based access control (RBAC), forms of entity-based access control (EBAC) and (in the near future) relation-based access control (ReBAC).
- **Human Readability**. The SAPL syntax is designed from the ground up to be easily readable by humans. Basic SAPL is easy to pick-up for getting started but offers enough expressiveness to address complex access control scenarios.
- **Transformation and Filtering**. SAPL allows transforming resources and filtering data from

resources (e.g., blacken the first digits of a credit card number, or hiding of birth dates by assigning individuals into age groups).

- **Designed for a session and data stream-based applications**. SAPL is designed to offer low-latency authorization for interactive applications and data streams.
- **Designed for a RESTful World with JSON**. SAPL is designed to be easily integrated with modern JSON-based APIs. The core data model of SAPL is JSON offering easy reasoning over such data and simple access to external attributes from RESTful JSON APIs.
- **Supports Multi-Subscriptions**. SAPL allows to bundle multiple authorization subscriptions into one multi-subscription thus further reducing connection time and latency.

The following sections will explain the basic concepts of SAPL policies and show how to easily integrate SAPL in a Java application. Afterwards the different parts of SAPL are explained in more detail.

Authorization Subscriptions

A SAPL authorization subscription is a JSON object, i.e., a set of name/value pairs or *attributes*. It contains attributes with the names `subject`, `action`, `resource` and `environment`. The values of these attributes may be any arbitrary JSON value, e.g.:

Introduction - Sample Authorization Subscription

```
{
  "subject"     : {
                    "username"    : "alice",
                    "tracking_id" : 1234321,
                    "nda_signed"  : true
                  },
  "action"      : "HTTP:GET",
  "resource"    : "https://medical.org/api/patients/123",
  "environment" : null
}
```

This authorization subscription expresses the intent of the user `alice`, with the given attributes, to `HTTP:GET` the resource at `https://medical.org/api/patients/123`. Such a SAPL authorization subscription may be issued in a RESTful API by a PEP in front of the request handlers of the API.

Structure of a SAPL Policy

A SAPL policy document generally consists of:

- the keyword `policy`, declaring that the document contains a policy (opposed to a policy set; more on policy sets see below)
- a unique (for the PDP) policy name
- the entitlement, which is the decision result to be returned upon successful evaluation of the policy, i.e., `permit` or `deny`

- an optional target expression for indexing and policy selection
- an optional `where` clause containing the conditions under which the entitlement (`permit` or `deny` as defined above) applies
- optional `advice` and `obligation` clauses to inform the PEP about optional and mandatory requirements for granting access to the resource
- an optional `transformation` clause for defining a transformed resource to be used instead of the original resource

A simple SAPL policy which allows `alice` to `HTTP:GET` the resource `https://medical.org/api/patients/123` would look as follows (in a real world this policy is too specific):

Introduction - Sample Policy 1

```
policy "permit_alice_get_patient123" ①
permit resource =~ "^https://medical.org/api/patients.*" ②
where ③
  subject.username == "alice"; ④
  action == "HTTP:GET";
  resource == "https://medical.org/api/patients/123";
```

① declares the policy with the name `permit_alice_get_patient123`. The JSON values of the authorization subscription object are bound to the variables `subject`, `action`, `resource` and `environment` and can be directly accessed in the policy. The syntax `.name` accesses attributes of a nested JSON object.

② declares that if the resource is a string starting with `https://medical.org/api/patients` (using the regular expression operator `=~`) and the conditions of the `where` clause applies, the subject will be permitted access to the resource. Note, that the `where` clause is only evaluated if the condition of the target expression evaluates to `true`.

③ starts the `where` clause (policy body) consisting of a list of statements. The policy body evaluates to `true` if all statements evaluate to `true`.

Authorization Decisions

The SAPL authorization decision to the authorization subscription is a JSON object as well. It contains the attribute `decision` as well as the optional attributes `resource`, `obligation` and `advice`. For the introductory sample authorization subscription with the preceding policy a SAPL authorization decision would look as follows:

Introduction - Sample Authorization Decision

```
{
  "decision"   : "PERMIT"
}
```

This authorization decision is evaluated by the PEP to grant or deny access.

Accessing Attributes

In many use-cases, all required information for making a decision may be contained in the authorization subscription object. However, the PEP is usually not aware of the specifics of the access policies and may not have access to all information required for making the decision. In this case, the PDP is able to access external attributes. The following example shows how accessing attributes is expressed in SAPL.

Extending the example above, in a real world application there will be multiple patients and multiple users. Thus policies need to be worded in a more abstract way. In a natural language, a suitable policy could be *Permit doctors to HTTP:GET data from any patient*. The policy addresses the profile attribute of the subject, stored externally. SAPL allows to express this policy as follows:

Introduction - Sample Policy 2

```
policy "doctors_get_patient"
permit
  action == "HTTP:GET" &
  resource =~ "^https://medical\.org/api/patients/\d*$"
where
  subject.username.<user.profile>.function == "doctor";
```

In *line 4* a regular expression is used for identifying a request to any patient's data (operator `=~`). The authorization subscription resource must match this pattern for the policy to apply.

The policy assumes that the user's function is not provided in the authorization subscription but stored in the user's profile. Accordingly *line 6* accesses the attribute `user.profile` (using an attribute finder step `.<finder.name>`) to retrieve the profile for the user with the username provided in `subject.username`. The fetched profile is a JSON object with a property named `function` which can be compared to `"doctor"`.

Line 6 is placed in the policy body (starting with `where`) instead of the target expression. The reason for this location is that the target expression block is also used for indexing policies efficiently and therefore needs to be evaluated quickly. Hence it is not allowed to include conditions which may need to call an external service.

This should give you a basic understanding of the operation principles applied in SAPL.

Getting Started

SAPL provides an embedded PDP including an embedded PRP with a file system policy store which can be easily integrated in a Java application:

1. Add SAPL to your application. When using Maven you can add the following dependencies to your project's `pom.xml`:

   pom.xml

   ```
   <dependency>
       <groupId>io.sapl</groupId>
       <artifactId>sapl-pdp-embedded</artifactId>
       <version>2.0.0-SNAPSHOT</version>
   </dependency>
   ```

2. In your application, create a new `EmbeddedPolicyDecisionPoint` (the builder methods declare multiple Exceptions which must be handled). The arguments `"C:/path/to/config"` and `"C:/path/to/policies"` are strings specifying the directories which contain the configuration file `pdp.json` and all policies respectively.

   ```
   EmbeddedPolicyDecisionPoint pdp = EmbeddedPolicyDecisionPoint.builder()
       .withFilesystemPDPConfigurationProvider("C:/path/to/config")
       .withFilesystemPolicyRetrievalPoint("C:/path/to/policies", IndexType.FAST)
       .build();
   ```

3. Add a `pdp.json` with the following content to the directory `C:/path/to/config`:

   ```
   {
       "algorithm": "DENY_UNLESS_PERMIT",
       "variables": {},
       "attributeFinders": [],
       "libraries": []
   }
   ```

4. Add some policy sets or policies to `C:/path/to/policies`. Both policy sets and policies are files with the extension `.sapl`. E.g., you could add the following policy:

   test.sapl

   ```
   policy "test_policy"
   permit subject == "admin"
   ```

5. Obtain a decision using the PDP's `decide` method.

   The whole source code to make use of the `test.sapl` policy could look as follows:

```
import java.io.IOException;
import java.net.URISyntaxException;

import com.fasterxml.jackson.databind.node.JsonNodeFactory;

import io.sapl.api.functions.FunctionException;
import io.sapl.api.interpreter.PolicyEvaluationException;
import io.sapl.api.pdp.PDPConfigurationException;
import io.sapl.api.pdp.PolicyDecisionPoint;
import io.sapl.api.pdp.AuthorizationSubscription;
import io.sapl.api.pdp.AuthorizationDecision;
import io.sapl.api.pip.AttributeException;
import io.sapl.pdp.embedded.EmbeddedPolicyDecisionPoint.Builder.IndexType;
import reactor.core.publisher.Flux;

PolicyDecisionPoint pdp = null;
try {
    pdp = EmbeddedPolicyDecisionPoint.builder()
            .withFilesystemPDPConfigurationProvider("C:/path/to/config")
            .withFilesystemPolicyRetrievalPoint("C:/path/to/policies", IndexType
.FAST)
            .build();
}
catch (IOException | URISyntaxException | PolicyEvaluationException |
       PDPConfigurationException | FunctionException | AttributeException e) {
    e.printStackTrace();
}

JsonNodeFactory JSON = JsonNodeFactory.instance;
AuthorizationSubscription authzSubscription = new AuthorizationSubscription(JSON
.textNode("admin"),
    JSON.textNode("an_action"), JSON.textNode("a_resource"), null);

Flux<AuthorizationDecision> authzDecisions = pdp.decide(authzSubscription);
authzDecisions.subscribe(authzDecision -> System.out.println(authzDecision
.getDecision()));
```

The console output should be `PERMIT`. With subject set to `"alice"` instead of `"admin"`, the output should be `DENY`.

Reference Architecture

The architecture of the SAPL policy engine is in accordance with the terminology defined by RFC2904 "AAA Authorization Framework".

[SAPL Architecture] | *SAPL_Architecture.svg*

Policy Enforcement Point (PEP)

The PEP is a software entity which intercepts actions taken by users within an application. Its task is to obtain a decision whether the requested action should be allowed and accordingly either let the application process the action or deny access. For this purpose the PEP includes data describing the subscription context (like the subject, the resource, the action and other environment information) in an authorization subscription object which the PEP hands over to a PDP. The PEP subsequently receives an authorization decision object containing a decision as well as optionally a resource, obligations and advice.

The PEP must let the application process the action if and only if the decision is `PERMIT`. If the authorization decision object also contains an `obligation` the PEP must fulfill this obligation. Proper fulfillment is an additional requirement for granting access. In case the decision is not `PERMIT` or the obligation cannot be fulfilled, access has to be denied. Since policies may contain instructions to alter the resource (like blackening certain information, e.g., credit card numbers) the PEP should ensure that the application only reveals the resource contained in the authorization decision object if one is returned.

NOTE	A PEP strongly depends on the application domain. SAPL comes with a default PEP implementation using a passed in constraint handler service to handle obligations and advices contained in an authorization decision. In the future, more generic PEPs integrating with various platforms (e.g., spring security) will be provided.

Policy Decision Point (PDP)

The PDP has to make an authorization decision based on an authorization subscription object and the access policies which it receives from a **Policy Retrieval Point (PRP)** connected to a policy store. Beginning with the authorization subscription object the PDP fetches policy sets and policies matching the authorization subscription, evaluates them and combines the results to create and return an authorization decision object. As there may be multiple matching policies which might evaluate to different results the PDP needs to be configured with a **combining algorithm** (e.g., `permit-overrides` stating that the decision will be `permit` if any applicable policy evaluates to `permit`).

A policy may refer to attributes which are not included in the authorization subscription object and have to be obtained from external **Policy Information Points (PIP)**. The PDP fetches those attributes while evaluating the policy. To be able to access external PIPs the PDP can be extended by custom attribute finders. Policies might also contain functions not included in the default SAPL implementation. Such custom functions can be added through **Policy Function Providers (PFP)**.

SAPL provides two simple PDP implementations: An **embedded PDP** with an embedded PRP which can be integrated easily into a Java application and a **remote PDP client** which obtains decisions through a RESTful interface.

Policy Administration Point (PAP)

The PAP is an entity which allows managing policies contained in the policy store. In the embedded PDP with the Resources PRP the policy store can be a simple folder within the local file system

containing `.sapl` files. Therefore any access to files in this folder (e.g., FTP or SSH) can be seen as a very simple PAP. The PAP may be a separate application or be included in an existing administration panel.

Publish / Subscribe Protocol

The PDP receives an authorization subscription from a PEP and sends an authorization decision. Both subscription and decision are JSON objects consisting of name/value pairs (also called attributes) with predefined names. A PEP must be able to create an authorization subscription and to process an authorization decision object.

SAPL Authorization Subscription

A SAPL authorization subscription contains attributes with the names `subject`, `resource`, `action` and `environment`. Each attribute value can be any JSON value (i.e., an object, an array, a number, a string, `true`, `false` or `null`).

SAPL Authorization Decision

The SAPL authorization decision contains the attributes `decision`, `resource`, `obligation` and `advice`.

Decision

The `decision` tells the PEP whether to grant or deny access. Access should be granted only if the decision is `"PERMIT"`. The `decision` attribute can be one of the following string values with the described meanings:

- `"PERMIT"`: Access must be granted.
- `"DENY"`: Access must be denied.
- `"NOT_APPLICABLE"`: A decision could not be made because no policy is applicable to the authorization subscription.
- `"INDETERMINATE"`: A decision could not be made because an error occurred.

Resource

The PEP knows which resource it requested access for. Thus there usually is no need for returning this resource in the authorization decision object. However SAPL policies may contain a `transform` statement describing how the resource needs to be altered before it is returned to the subject seeking permission. This can be used to remove or blacken certain parts of the resource document (e.g., a policy could allow doctors to view patient data but remove any bank account details as they can only be accessed by the accounting department). If a policy which evaluates to `PERMIT` contains a `transform` statement, the authorization decision attribute `resource` contains the transformed resource. Otherwise there will not be a `resource` attribute in the authorization decision object.

Obligation

The value of `obligation` contains assignments which the PEP must fulfill before granting or denying access. As there can be multiple policies applicable to the authorization subscription with different obligations, the `obligation` value in the authorization decision object is an array containing a list of tasks. If the PEP is not able to fulfill these tasks access must not be granted. The array items can be any JSON value (e.g., a string or an object). Consequently the PEP must know how to identify and process the obligations contained in the policies. An `obligation` attribute is only included in the authorization decision object if there is at least one obligation.

A authorization decision could, for example, contain the obligation to create a log entry.

Advice

The value of `advice` is an array with assignments for the PEP as well and works similar to obligations except for one difference: The fulfillment of the tasks is no requirement for granting access. I.e., in case the `decision` is `PERMIT`, the PEP should also grant access if it can not fulfill the tasks contained in `advice`. An `advice` attribute is only included in the authorization decision object if there is at least one advice.

In addition to the obligation to create a log entry, a policy could specify the advice to inform the system administrator via email about the access.

Policy Evaluation

To come to the final decision included in the authorization decision object, the PDP evaluates all existing policy sets and top level policies (i.e., policies which are not part of a policy set) against the authorization subscription and combines the results. Each individual policy set and policy evaluates to `PERMIT`, `DENY`, `NOT_APPLICABLE` or `INDETERMINATE` (see below). The PDP can be configured with a **combining algorithm** which determines how to deal with multiple results. E.g., if access should only be granted if at least one policy evaluates to `PERMIT` and should be denied otherwise, the algorithm `deny-unless-permit` could be used.

Available combining algorithms for the PDP are:

- `deny-unless-permit`
- `permit-unless-deny`
- `only-one-applicable`
- `deny-overrides`
- `permit-overrides`

The algorithm `first-applicable` is not available for the PDP since the PDP's collection of policy sets and policies is an unordered set.

The combining algorithms are described in more detail later.

Multi-Subscriptions

SAPL allows for bundling multiple authorization subscriptions into one multi-subscription. A multi-subscription is a JSON object with the following structure:

Multi-Subscriptions - JSON Structure

```
{
  "subjects"                  : ["bs@simpsons.com", "ms@simpsons.com"],
  "actions"                   : ["read"],
  "resources"                 : ["file://example/med/record/patient/BartSimpson",
                                 "file://example/med/record/patient/MaggieSimpson"],
  "environments"              : [],

  "authorizationSubscriptions" : {
                                   "id-1" : { "subjectId": 0, "actionId": 0,
"resourceId": 0 },
                                   "id-2" : { "subjectId": 1, "actionId": 0,
"resourceId": 1 }
                                 }
}
```

It contains distinct lists of all subjects, actions, resources and environments referenced by the single authorization subscriptions being part of the multi-subscription. The authorization subscriptions themselves are stored in a map of subscription IDs pointing to an object defining an authorization subscription by providing indexes into the four lists mentioned before.

The multi-subscription shown in the example above contains two authorization subscriptions. The user `bs@simpsons.com` wants to `read` the file `file://example/med/record/patient/BartSimpson` and the user `ms@simpsons.com` wants to `read` the file `file://example/med/record/patient/MaggieSimpson`.

The SAPL PDP processes all individual authorization subscriptions contained in the multi-subscription in parallel and either returns the related authorization decisions as soon as they are available or it collects all the authorization decisions of the individual authorization subscriptions and returns them as a multi-decision. In both cases the authorization decisions are associated with the subscription IDs of the related authorization subscription. The following listings show the JSON structures of the two authorization decision types:

Single Authorization Decision with Associated Subscription ID - JSON Structure

```
{
  "authorizationSubscriptionId" : "id-1",
  "authorizationDecision"       : {
                                    "decision" : "PERMIT",
                                    "resource" : { ... }
                                  }
}
```

Multi-Decision - JSON Structure

```
{
  "authorizationDecisions" : {
                               "id-1" : {
                                          "decision" : "PERMIT",
                                          "resource" : { ... }
                                        },
                               "id-2" : {
                                          "decision" : "DENY"
                                        }
                             }
}
```

The SAPL Policy Language

SAPL defines a feature-rich domain specific language (DSL) for creating access policies.

Those access policies describe when access requests will be granted and when access will be denied. The underlying concept to describe these permissions is an attribute-based access control model (ABAC): A SAPL authorization subscription is a JSON object with the attributes `subject`, `action`, `resource` and `environment` each with an assigned JSON value. Each of these values may be a JSON object itself containing multiple attributes. Policies can make use of boolean conditions referring to those attributes (e.g., `subject.username == "admin"`).

However a role based access control (RBAC) system in which permissions are assigned to a certain role and roles can be assigned to users can be created with SAPL as well.

Overview

SAPL knows two types of documents: Policy sets and policies. The decisions of the PDP are based on all documents published in the policy store of the PDP. A policy set contains a number of connected policies.

Policy Structure

A SAPL policy consists of optional **imports**, a **name**, an **entitlement** specification, an optional **target expression**, an optional **body** with one or more statements, and optional sections for **obligation**, **advice** and **transformation**. An example of a simple policy is:

Sample SAPL Policy

```
import filter as filter ①

policy "test_policy" ②
permit ③
    subject.id == "anId" | action == "anAction" ④
where
    var variable = "anAttribute";
    subject.attribute == variable; ⑤
obligation
    "logging:log_access" ⑥
advice
    "logging:inform_admin" ⑦
transform
    resource.content |- filter.blacken ⑧
```

① Imports (optional)

② Name

③ Entitlement

④ Target Expression (optional)

⑤ Body (optional)

⑥ Obligation (optional)

⑦ Advice (optional)

⑧ Transformation (optional)

Policy Set Structure

A SAPL policy set contains optional **imports**, a **name**, a **combining algorithm**, an optional **target expression**, optional **variable definitions** and a list of **policies**. The following example shows a simple policy set with two policies:

Sample SAPL Policy Set

```
import filter.* ①

set "test_policy_set" ②
deny-unless-permit ③
for resource.type == "aType" ④
var dbUser = "admin";⑤

    policy "test_permit_admin" ⑥
    permit subject.function == "admin"

    policy "test_permit_read" ⑦
    permit action == "read"
    transform resource |- blacken
```

① Imports (optional)

② Name

③ Combining Algorithm

④ Target Expression (optional)

⑤ Variable Assignments (optional)

⑥ Policy 1

⑦ Policy 2

Imports

SAPL provides access to functions or attribute finders stored in libraries. The names of those libraries usually consist of different parts separated by periods (e.g. `sapl.pip.http` - a library containing functions to obtain attributes through HTTP requests). In policy documents, the functions and finders can be accessed by their fully qualified name, i.e. the name of the library followed by a period (`.`) and the function or finder name, e.g. `sapl.pip.http.get`.

For any SAPL top level document (i.e. a policy set or a policy which is not part of a policy set) any number of imports can be specified. Imports allow using a shorter name instead of the fully qualified name for a function or an attribute finder within a SAPL document. Thus imports can make policy sets and policies easier to read and write.

Each import statement starts with the keyword `import`.

- **Basic Import**: A function or an attribute finder can be imported by providing its fully qualified name (e.g. `import sapl.pip.http.get`). It will be available under its simple name (in the example: `get`) in the whole SAPL document.
- **Wildcard Import**: All functions or attribute finders from a library can be imported by providing an asterisk instead of a function or finder name (e.g. `import sapl.pip.http.*`). All functions or finders from the library will be available under their simple names (in the example: `get`).

- **Library Alias Import**: All functions or attribute finders from a library can be imported by providing the library name followed by `as` and an alias, e.g. `import sapl.pip.http as rest`.

The SAPL document can contain any number of imports, e.g.

Sample Imports

```
import sapl.pip.http.*
import filter.blacken
import simple.append

policy "sample"
...
```

SAPL Policy

This section describes the elements of a SAPL policy in more detail. A policy contains an entitlement (`permit` or `deny`) and can be evaluated against an authorization subscription. If the conditions in the target expression and in the body are fulfilled, the policy evaluates to its entitlement. Otherwise it evaluates to `NOT_APPLICABLE` (if one of the conditions is not satisfied) or `INDETERMINATE` (if an error occurred).

A SAPL policy starts with the keyword `policy`.

Name

The keyword `policy` is followed by the policy name. The name is a string *identifying* the policy, thus it has to be unique. Accordingly in systems with many policy sets and policies it is recommended to use a schema to create names (e.g., `"policy:patientdata:permit-doctors-read"`).

Entitlement

SAPL expects an entitlement specification. This can either be `permit` or `deny`. The entitlement is the value which the policy evaluates to if the policy is applicable to the authorization subscription, i.e., if both the conditions in the policy's target expression and in the policy body are satisfied.

NOTE	Since multiple policies can be applicable and the combining algorithm can be chosen, it might make a difference whether there is an explicit `deny`-policy or whether there is just no permitting policy for a certain situation.

Target Expression

Subsequent to the entitlement, an **optional** target expression can be specified. This is a condition for applying the policy, hence an expression which must evaluate to either `true` or `false`. Which elements are allowed in SAPL expressions is described below.

If the target expression evaluates to `true` for a certain authorization subscription, the policy *matches* this subscription. A missing target expression makes the policy match any subscription.

A matching policy whose conditions in the body evaluate to `true` is called *applicable* to an authorization subscription and returns its entitlement. Both target expression and body define conditions which must be satisfied for the policy to be applicable. Although they seem to serve a similar purpose there is an important difference: For an authorization subscription the target expression of each top level document is checked in order to select policies matching the subscription from a possibly large set of policy documents. Indexing mechanisms may be used to fulfill this task efficiently.

Accordingly, there are two limitations regarding the elements allowed in the target:

- As lazy evaluation deviates from boolean logic and prevents effective indexing, the logical operators `&&` and `||` may not be used. Instead, the target needs to make use of the operators `&` and `|`, for which eager evaluation is applied.
- Attribute finder steps which have access to environment variables and may contact external PIPs are not allowed in the target. Yet functions may be used because their output only depends on the arguments passed.

Body

The policy body is **optional** and starts with the keyword `where`. It contains one or more statements each of which must evaluate to `true` for the policy to apply to a certain authorization subscription. Accordingly the body extends the condition in the target expression and further limits the policy's applicability.

A statement within the body can either be a variable assignment which makes a variable available under a certain name (and always evaluates to `true`)

Sample Variable Assignment

```
var a_name = expression;
```

or a condition, i.e., an expression that evaluates to `true` or `false`.

Sample Condition

```
a_name == "a_string";
```

Each statement is concluded with a semicolon `;`.

There are no restrictions on the syntax elements allowed in the policy body. Lazy evaluation is used for the conjunction of the statements - i.e., if one statement evaluates to `false`, the policy returns the deicision `NOT_APPLICABLE`, even if future statements would cause an error.

If the body is missing (or does not contain any condition statement) the policy is applicable to any authorization subscription which the policy matches (i.e., for which the target expression evaluates to `true`).

Variable Assignment

A variable assignment starts with the keyword `var`, followed by an identifier under which the assigned value should be available, followed by `=` and an expression.

After a variable assignment, the result of evaluating the expression can be used in later conditions within the same policy under the specified name. This is useful because it allows to execute time consuming calculations or requests to external attribute stores only once although the result can be used in multiple expressions. Besides it can make policies shorter and more readable.

The expression can make use of any element of the SAPL expression language, especially of attribute finder steps which are not allowed in the target expression.

The value assignment statement always evaluates to `true`.

Condition

A condition statement simply consists of an expression that has to evaluate to `true` or `false`.

The expression can make use of any element of the SAPL expression language, especially of attribute finder steps which are not allowed in the target expression. Conditions in the policy body are used to further limit the applicability of a policy.

Obligation

An **optional** obligation expression contains a task which the PEP must fulfill before granting or denying access. It consists of the keyword `obligation` followed by an expression.

A common situation in which obligations are useful are *Break the Glass Scenarios*. Assuming in case of an emergency a doctor should also have access to medical records that she normally cannot read. However this emergency access has to be logged in order to prevent abuse. In this situation, logging is a requirement for granting access and therefore must be commanded in an obligation.

Obligations are only returned in the authorization decision if the decision is `PERMIT` or `DENY`. The PDP simply collects all obligations from policies evaluating to one of these entitlements. Depending on the final decision, the obligations and advice which belong to this decision are included in the authorization decision object. It does not matter if the obligation is described with a string (like `"create_emergency_access_log"`) or an object (like `{ "task" : "create_log", "content" : "emergency_access" }`) or another JSON value - only the PEP must be implemented in a way that it knows how to process these obligations.

Advice

An **optional** advice expression is treated similarly to an obligation expression. Unlike obligations, fulfilling the described tasks in advice is not a requirement for granting or denying access. The advice expression consists of the keyword `advice` followed by any expression.

If the final decision is `PERMIT` or `DENY`, advice from all policies evaluating to this decision is included in the authorization decision object by the PDP.

Transformation

An **optional** transformation statement is preluded with the keyword `transform` and followed by an expression. If a transformation statement is supplied and the policy evaluates to `permit`, the result of evaluating the expression will be returned as `resource` in the authorization decision object.

Accordingly, a transformation statement might be used to hide certain information (e.g., *a doctor can access patient data but should not see bank account details*). This can be reached by applying a filter to the original resource which removes or blackens certain attributes. Thus SAPL allows for **fine grained** or **field level** access control without the need to treat each attribute as a resource and write an own policy for it.

The original resource is accessible via the identifier `resource` and can be filtered as follows:

Transformation Example

```
transform
    resource |- {
        @.someValue : remove,
        @.anotherValue : filter.blacken
    }
```

The example would remove the attribute `someValue` and blacken the value of the attribute `anotherValue`. The filtering functions are described in more detail below.

It is not possible to combine multiple transformation statements through multiple policies. Each combining algorithm in SAPL will not return the decision `PERMIT` if there are more than one policies evaluating to `PERMIT` and at least one of them contains a transformation statement (this is called *transformation uncertainty*). For more details, see below.

SAPL Policy Set

While a policy can either be a top level SAPL document or be contained in a policy set, policy sets are always top level documents. I.e., for evaluating an authorization subscription, the PDP evaluates any existing policy set. Policy sets are evaluated against an authorization subscription by checking their target expression, if applicable evaluating their policies and if necessary combining multiple decision according to a combining algorithm specified in the policy set. Finally, similar to policies, policy sets evaluate to either `PERMIT`, `DENY`, `NOT_APPLICABLE` or `INDETERMINATE`.

Policy sets are used to structure multiple policies and provide an order for the policies they contain. Thus their policies can be evaluated one after another.

A policy set definition starts with the keyword `set`.

Name

The keyword `set` is followed by the policy set name. The name is a string *identifying* the policy set. Thus it has to be unique within all policy sets and policies.

Combining Algorithm

The name is followed by a combining algorithm. This algorithm describes how to combine the results from evaluating every policy to come to a result for the policy set.

Possible values are:

- `deny-unless-permit`
- `permit-unless-deny`
- `only-one-applicable`
- `deny-overrides`
- `permit-overrides`
- `first-applicable`

The combining algorithms are described in more detail later.

Target Expression

Subsequent to the combining algorithm, an **optional** target expression can be specified. The target expression is a condition for applying the policy set. It starts with the keyword `for` followed by an expression which must evaluate to either `true` or `false`. If the condition evaluates to `true` for a certain authorization subscription the policy set *matches* this subscription. In case the target expression is missing the policy set matches any authorization subscription.

The policy sets' target expression is used to select matching policy sets from a large collection of policy documents before evaluating them. As this needs to be done efficiently, there are no attribute finder steps allowed at this place.

Variable Assignments

The target expression can be followed by any number of variable assignments. Variable assignments are used to make a value available in all subsequent policies under a certain name. An assignment starts with the keyword `var`, followed by an identifier under which the assigned value should be available, followed by `=` and an expression (see above).

Since variable assignments are evaluated only if the policy set's target matches, attribute finders may be used.

In case a policy within the policy set assigns a variable already assigned in the policy set, the assignment in the policy overwrites the old. The overwritten value only exists within the particular policy. In other policies, the variable has the value defined in the policy set.

Policies

The policy set must contain one or more policies. See above how to describe a SAPL policy. If the combining algorithm `first-applicable` is used, the policies are evaluated in the order in which they appear in the policy set.

In each policy, functions and attribute finders imported at the beginning of the SAPL document can be used under their shorter name. All variables assigned for the policy set (see Value Assignments) are available within the policies, but can be overwritten for a particular policy. The same applies to imports - imports at the policy level overwrite imports defined for the policy set, but are only valid for the particular policy.

Language Elements

The descriptions of the policy and policy set structure sometimes refers to language elements like identifiers and strings. These elements are explained in this section.

Identifiers

Multiple elements in policies or policy sets require identifiers. E.g. a variable assignments expects an identifier after the keyword `var` - the name under which the assigned value will be available.

An identifier only consists of alphanumeric characters, `_` and `$` and must not start with a number.

Valid Identifiers

```
a_long_name
aLongName
$name
_name
name123
```

Invalid Identifiers

```
a#name
1name
```

A caret `^` before the identifier may be used to avoid a conflict with SAPL keywords.

Strings

Whenever strings are expected, the SAPL document must contain any sequence of characters enclosed by single quotes `'` or double quotes `"`. Any enclosing quote character occurring in the string must be escaped by a preceding `\`, e.g., `"the name is \"John Doe\""`.

Comments

Comments are used to store information in a SAPL document which is only intended for human readers and has no meaning for the PDP. Comments are simply ignored when the PDP evaluates a document.

SAPL supports single line and multi line comments. A single line comment starts with `//` and ends at the end of the line, no matter which characters follow.

Sample Single Line Comment

```
policy "test" // a policy for testing
```

Multi line comments start with `/*` and end with `*/`. Everything in between is ignored.

Sample Multi Line Comment

```
policy "test"
/* A policy for testing.
Remove before deployment! */
```

SAPL Expressions

To ensure flexibility, various parts of a policy can be **expressions** which are evaluated at runtime. E.g., a policy's target has to be an expression evaluating to `true` or `false`. SAPL contains a uniform expression language which offers various useful features while still being easy to read and write.

Since JSON is the base data model, each expression evaluates to a JSON data type. These data types and the expression syntax are described in this section.

JSON Data Types

SAPL is based on the **JavaScript Object Notation** or **JSON**, an ECMA Standard for the representation of structured data. Any value occuring within the SAPL language is a JSON data type and any expression within a policy evaluates to a JSON data type. The types and their JSON notations are:

- Primitive Types
  - **Number**: A signed decimal number, e.g., `-1.9`. There is no distinction between integer and floating-point numbers. In case an integer is expected (e.g. for an numeric index), the decimal number is rounded to an integer number.
  - **String**: A sequence of zero or more characters, written in double or single quotes, e.g. `"a string"` or `'a string'`.
  - **Boolean**: Either `true` or `false`.
  - **null**: Marks an empty value, `null`.
- Structured Types
  - **Object**: An unordered set of name/value pairs. The name is a string, the value has to be one of the available data types. It can also be an object itself. The name/value pair is also called attribute of the object. E.g.

```
{
    "firstAttribute" : "first value",
    "secondAttribute" : 123
}
```

- **Array**: An ordered sequence of zero or more values of any JSON data type. E.g.

```
[
    "A value",
    123,
    {"attribute" : "value"}
]
```

Expression Types

SAPL knows **basic expressions** and **operator expressions** (created from other expressions using operators).

A **basic expression** is either a

- **Value Expression**: a value explicitly defined in the corresponding JSON notation (e.g. `"a value"`)
- **Identifier Expression**: the name of a variable or of a authorization subscription attribute (`subject`, `resource`, `action` or `environment`)
- **Function Expression**: a function call (e.g. `simple.get_minimum(resource.array)`)
- **Relative Expression**: `@`, which refers to a certain value depending on the context
- **Grouped Expression**: any expression enclosed in parantheses, e.g. `(1 + 1)`

Each of these basic expressions can contain one or more **selection steps** (e.g., `subject.name` which is the identifier expression `subject` followed by the selection step `.name` selecting the value of the `name` attribute). Eventually a basic expression can contain a **filter component** (`|- Filter`) which will be applied to the evaluation result. If the expression evaluates to an array, instead of applying a filter each item can be transformed using a **subtemplate component** (`:: Subtemplate`).

Operator expressions can be constructed using prefix or infix **operators** (e.g., `1 + subject.age` or `! subject.isBlocked`). SAPL supports infix and prefix operators. They may be applied in connection with any expression. An operator expression within parantheses (e.g., `(1 + subject.age)`) is a basic expression again and thus may contain selection steps, filter or subtemplate statements.

Value Expressions

A basic value expression is the most simple type. The value is denoted in the corresponding JSON format.

`true`, `false` and `null` are value expressions as well as `"a string"`, `'a string'` or any number (like `6` or `100.51`).

For denoting objects the keys need to be strings and the values can be any expression, e.g.

```
{
    "id" : (3+5),
    "name" : functions.generate_name()
}
```

For arrays the items can be any expression, e.g.

```
[
    (3+5),
    subject.name
]
```

Identifier Expressions

A basic identifier expression consists of the name of a variable or the name of an authorization subscription attribute (i.e., `subject`, `resource`, `action` or `environment`).

It evaluates to the variable's or the attribute's value.

Function Expressions

A basic function expression consists of a function name and any number of arguments between parentheses which are separated by commas. The arguments have to be expressions, e.g.

```
library.a_function(subject.name, (environment.day_of_week + 1))
```

Each function is available under its fully qualified name. The fully qualified name starts with the library name, consisting of one or more identifiers separated by periods `.` (e.g. `sapl.functions.simple`). The library name is followed by a period `.` and an identifier for the function name (e.g. `sapl.functions.simple.append`). Which function libraries are available depends on the configuration of the PDP.

Imports at the beginning of a SAPL document can be used to make functions available under shorter names. If a function is imported via a basic import or a wildcard import, it is available under its function name (e.g., `append`). A library alias import provides an alternative library name (e.g., with the import statement `import sap.functions.simple as simple`, the append function would be available under `simple.append`.

If there are no arguments passed to the function, empty parentheses have to be denoted (e.g., `random_number()`).

When evaluating a function expression the expressions representing the function call arguments are evaluated first. Afterwards the results are passed as arguments to the function. The expression evaluates to the function's return value.

Relative Expressions

The basic relative expression is the `@` symbol.

It can be only used in various contexts. Those contexts are characterized by an implicit loop with `@` dynamically evaluating to the current element. Assuming the variable `array` contains an array with multiple numbers. The expression `array[?(@ > 10)]` can be used to return any element greater than 10. In this context, `@` evaluates to the array item for which the condition is currently checked.

The contexts in which `@` can be used are:

- Expression within a condition step (`@` evaluates to the array item or attribute value for which the condition expression is currently evaluated)
- Subtemplate (`@` evaluates to the array item which is currently going to be replaced by the subtemplate)
- Arguments of a filter function if `each` is used (`@` evaluates to the array item to which the filter function is going to be applied)

Operators

SAPL provides a collection of arithmetic, comparison, logical, string and filtering operators which can be used to build expressions from other expressions.

Arithmetic Operators

Assuming `exp1` and `exp2` are expressions evaluating to numbers, the following operators can be applied. All of them evaluate to a number.

- `-exp1` (negation)
- `exp1 * exp2` (multiplication)
- `exp1 / exp2` (division)
- `exp1 + exp2` (addition)
- `exp1 - exp2` (subtraction)

An expression can contain multiple arithmetic operators. The order in which they are evaluated can be specified using **parentheses**, e.g., `(1 + 2) * 3`.

In case multiple operators are used without parentheses (e.g., `4 + 3 * 2`) the **operator precedence** determines about how the expression is evaluated. Operators with higher precedence are evaluated first. The following precedence is assigned to arithmetic operators:

- `-` (negation): precedence **4**
- `*` (multiplication), `/` (division): precedence **2**
- `+` (addition), `-` (subtraction): precedence **1**

As `*` has a higher precedence than `+`, `4 + 3 * 2` would be evaluated like `4 + (3 * 2)`.

Except for the negation, multiple operators with the same precedence (e.g., `5 - 2 + 1`) are **left-**

associative, i.e., `5 - 2 + 1` is evaluated like `(5 - 2) + 1`. The negation is non-associative, i.e., `--1` needs to be replaced by `-(-1)`.

Comparison Operators

1. Number Comparison

   Assuming `exp1` and `exp2` are expressions evaluating to numbers, the following operators can be applied. All of them evaluate to `true` or `false`.

   a. `exp1 < exp2` (`true` if `exp2` is greater than `exp1`)

   b. `exp1 > exp2` (`true` if `exp1` is greater than `exp2`)

   c. `exp1 <= exp2` (`true` if `exp2` is equal to or greater than `exp1`)

   d. `exp1 >= exp2` (`true` if `exp1` is equal to or greater than `exp2`)

2. Equals

   Assuming `exp1` and `exp2` are expressions, the equals-operator can be used to compare the results:

   ```
   exp1 == exp2
   ```

   The expression evaluates to `true` if the result of evaluating `exp1` is equal to the result of evaluating `exp2`.

3. Regular Expression

   Assuming `exp1` and `exp2` are expressions evaluating to strings, the regular expression match operator can be used:

   ```
   exp1 =~ exp2
   ```

   The expression evaluates to `true` if the result of evaluating `exp1` matches the pattern contained in the result of evaluating `exp2`. The pattern needs to be specified according to the java.util.regex package.

4. `in` (element of)

   Assuming `exp1` is an expression and `exp2` is an expression evaluating to an array, the `in` operator can be used:

   ```
   exp1 in exp2
   ```

   The expression evaluates to `true` if the array `exp2` evaluates to contains the result of evaluating `exp1`. Otherwise the expression evaluates to `false`.

5. Precedence and Associativity

   All comparison operators have the precedence **3**. This is important for combining them with logical operators (see below).

   `<`, `>`, `<=`, `>=`, `==`,`=~` and `in` are **non-associative**, i.e., an expression may not contain multiple

comparison operators (like `3 < var < 5`). However they can be combined with logical operators which have a different precedence (thus, the faulty example could be replaced by `3 < var && var < 5`).

Logical Operators

Assuming `exp1` and `exp2` are expressions evaluating to `true` or `false`, the following operators can be applied. The new expression evaluates to `true` or `false`:

- `!exp1` (negation), precedence **4**
- `exp1 && exp2` or `exp1 & exp2` (logical AND), precedence **2**
- `exp1 || exp2` or `exp1 | exp2` (logical OR), precedence **1**

The difference between `&&` and `&` (or `||` and `|`) is that for `&&` lazy evaluation is used while `&` causes eager evaluation. Using `&&`, if the left side evaluates to `false` and the right side would cause an error, the result of the operator is `false`, the right side is not evaluated. The same applies for `||` if the left side evaluates to `true`. In this case, the operator evaluates to `true`, even if the right side would cause an error - the right side is ignored if the result can already be determined. This is different for `&` and `|` which always evaluate both sides first (eager evaluation). Whenever there is an error, the expression does not return a result. In a target expression, only the eager evaluation expressions `&` and `|` can be used.

The operators are already listed in descending order of their **precedence**, i.e. `!` has the highest precedence followed by `&&`/`&` and `||`/`|`. The order of evaluation can be changed by the use of parentheses.

`&&` and `||` are left-associative, i.e., in case an expression contains multiple operators the leftmost operator is evaluated first. `!` is non-associative, i.e., `!!true` has to be replaced by `!(!true)`.

String Concatenation

The operator `+` concatenates two strings, e.g. `"Hello" + " World!"` evaluates to `"Hello World!"`.

String concatenation is applied if the left operand is an expression evaluating to a string. If the right expression evaluates to a string as well, the two strings are concatenated. Otherwise an error is thrown.

Selection Steps

SAPL provides an easy way of accessing attributes of an object (or items of an array). The **basic access** mechanism has a similar syntax to programming languages like JavaScript or Java (e.g., `object.attribute`, `user.address.street` or `array[10]`). Beyond that SAPL offers **extended possibilities** for expressing more sophisticated queries against JSON structures (e.g., `persons[?(@.age >= 50)]`).

Overview

The following table provides an overview of the different types of selection steps.

Given that the following object is stored in the variable `object`:

Structure of `object`

```
{
    "key" : "value1",
    "array1" : [
        { "key" : "value2" },
        { "key" : "value3" }
    ],
    "array2" : [
        1, 2, 3, 4, 5
    ]
}
```

Table 1. Selection Steps Overview

Expression	Returned Value	Explanation

`object.key`
`object['key']`
`object["key"]`
	`"value1"`	**Key step** in dot notation and bracket notation

`object.array1[0]`	`{ "key" : "value2" }`	**Index step**
`object.array2[-1]`	`5`	**Index step** with negative value n returns the n-th last element
`object.*` `object[*]`	`[ "value1", [ { "key" : "value2" }, { "key" : "value3" } ], [ 1, 2, 3, 4, 5 ] ]`	**Wildcard step** applied to an object returns an array with the value of each attribute - applied to an array it returns the array itself
`object.array2[0:-2:2]`	`[ 1, 3 ]`	**Array slicing step** starting from first to second last element with a step size of two

`object..key`
`object..['key']`
`object..["key"]`
	`[ "value1", "value2", "value3" ]`	**Recursive descent step** looking for an attribute

`object..[0]`	`[ { "key" : "value2" }, 1 ]`	**Recursive descent step** looking for an array index

Expression	Returned Value	Explanation
`object.array2[(3+1)]`	`5`	**Expression step** that evaluates to a number (index) - can also evaluate to an attribute name
`object.array2[?(@>2)]`	`[ 3, 4, 5 ]`	**Condition step** that evaluates to true/false, `@` is a reference to the currently examined item - can also be applied to an object
`object.array2[2,3]`	`[ 3 , 4 ]`	**Union step** for more than one array index
`object["key","array2"]`	`[ "value1", [ 1, 2, 3, 4, 5 ] ]`	**Union step** for more than one attribute

Basic Access

The basic access syntax is quite similar to accessing an object's attributes in JavaScript or Java:

- **Attributes of an object** can be accessed by their key (**key step**) using the *dot notation* (`resource.key`) or the *bracket notation* (`resource["key"]`,`resource['key']`). Both expressions return the value of the specified attribute. For using the dot notation the specified key has to be an identifier. Otherwise the bracket notation with a string between square brackets is necessary, e.g., if the key contains whitespace characters (`resource['another key']`).
- **Indices of an array** may be accessed by putting the index between square brackets (**index step**, `array[3]`). The index can be a negative number `-n` which evaluates to the `n`-th element from the end of the array, and starting with -1 as the last element's index. `array[-2]` would return the second last element of the Array `array`.

Multiple selection steps can be **chained**. The steps are evaluated from left to right. Each step is applied to the result returned from the previous step.

> **Example**
>
> The expression `object.array[2]` first selects the attribute with key `array` from the object `object` (first step). Then it returns the third element (index `2`) of that array (second step).

Extended Possibilities

SAPL supports querying for specific parts of a JSON structure. Except for an **expression step**, all of these steps return an array since the number of elements found can vary. Even if only a single result is retrieved the expression returns an array containing one item.

Expression Step `[(Expression)]`

Returns the value of an attribute with a key or an array item with an index specified by an expression. `Expression` must evaluate to a string or a number. If `Expression` evaluates to a string, the selection can only be applied to an object. If `Expression` evaluates to a number, the selection can only be applied to an array.

NOTE The expression step can be used to refer to custom variables (`object.array[(anIndex+2)]`) or apply custom functions (`object.array[(max_value(object.array))]`.

Wildcard Step `.*` or `[*]`

Can be applied to an object or an array. When applied to an object, it returns an array containing all attribute values. As attributes of an object have no ordering, the sorting of the result is not defined. When applied to an array, the step just leaves the array untouched.

NOTE Applied to an object `{"key1":"value1", "key2":"value2"}`, the selection step `.*` or `[*]` returns the following array: `["value1", "value2"]` (possibly with a different sorting of the items). Applied to an array `[1, 2, 3]`, the selection step `. or []` returns the original array `[1, 2, 3]`.

Recursive Descent Step `..key`, `..["key"]`, `..[1]`, `..*` or `..[*]`

Looks for the specified key or array index in the current object or array and, recursively, in its children (i.e., the values of its attributes or its items). The recursive descent step can be applied to both an object and an array. It returns an array containing all attribute values or array items found. If the specified key is an asterisk (`.. or []`, wildcard), all attribute values and array items in the whole structure are returned.

As attributes of an object are not sorted, the sorting of items in the result array is not guaranteed.

NOTE

Applied to an `object`

```
{
    "key" : "value1",
    "anotherkey" : {
        "key" : "value2"
    }
}
```

the selection step `object..key` returns the following array: `["value1", "value2"]` (any attribute value with key `key`, the items may be in a different order).

The wildcard selection step `object.. or object..[]` returns `["value1", {"key":"value2"}, "value2"]` (recursively each attribute value and array item in the whole structure `object`, the sorting may be different).

Condition `[?(Condition)]`

Returns an array containing all attribute values or array items for which `Condition` evaluates to `true`. Can be applied to both an object (then it checks each attribute value) and an array (then it checks each item). `Condition` must be an expression, in which relative expressions starting with `@` can be used. `@` evaluates to the current attribute value or array item for which the condition is evaluated and can be followed by further selection steps.

As attributes have no order, the sorting of the result array of a condition step applied to an object is not specified.

NOTE	Applied to the array `[1, 2, 3, 4, 5]`, the selection step `[?(@ > 2)]` returns the array `[3, 4, 5]` (containing all values that are greater than 2).

Array Slicing `[Start:End:Step]`

The slice contains the items with indices between `Start` and `Stop` with `Start` being inclusive and `Stop` being exclusive. `Step` describes the distance between the elements to be included in the slice, i.e., with a `Step` of 2, only each second element would be included (with `Start` as the first element's index). All parts except the first colon are optional. `Step` defaults to 1.

In case `Step` is positive, `Start` defaults to 0 and `Stop` defaults to the length of the array. If `Step` is negative, `Start` defaults to the length of the array minus 1 (i.e., the last element's index) and `Stop` defaults to -1. A `Step` of 0 leads to an error.

NOTE	Applied to the Array `[1, 2, 3, 4, 5]`, the selection step `[-2:]` returns the Array `[4, 5]` (the last two elements).

Index Union `[index1, index2, ⋯]`

Using the bracket notation, a set of multiple array indices (numbers) can be denoted separated by commas. This returns an array containing the items of the original array if the item's index is contained in the specified indices. Since a **set** of indices is specified, the indices' order is ignored and duplicate elements are removed. The result array contains the specified elements in their original order. Indices which do not exist in the original array are ignored.

NOTE	Both `[3, 2, 2]` and `[2, 3]` return the same result.

Attribute Union `["attribute1", "attribute2", ⋯]`

Using the bracket notation, a set of multiple attribute keys (strings) can be denoted separated by commas. This returns an array containing the values of the denoted attributes. Since a **set** of attribute keys is specified, the keys' order is ignored and duplicate elements are removed. As attributes have no order, the sorting of the result array is not specified. Attributes which do not exist are ignored.

Attribute Selection on Array

Although arrays do not have attributes (they have items), a key step can be applied to an array (e.g. `array.value`). This will loop through each item of the array and look for the specified attribute in

this item. An array containing all values of the attributes found is returned. In other words the selection step is not applied to the result of the previous step (the array) but to each item of the result and the (sub-)results are concatenated. In case an array item is no object or does not contain the specified attribute, it is skipped.

NOTE

Applied to an object

```
{
    "array":[
        {"key":"value1"},
        {"key":"value2"}
    ]
}
```

`array.key` returns the following array: `["value1", "value2"]` (the value of the `key` attribute of each item of `array`).

Attribute Finder `.<finder.name>`

In SAPL it is possible to receive attributes which are not contained in the authorization subscription. Those attributes can be provided by external PIPs and obtained through attribute finders. An attribute finder is called via the selection step `.<finder.name>`. `finder.name` either is a fully qualified attribute finder name or can be a shorter name if imports are used (the finder name or the library alias followed by a period `.` and the finder name).

The attribute finder receives the result of the previous selection as an argument and returns a JSON value.

NOTE

Assuming a doctor should only be allowed to access patient data from patients on her unit. The following expression retrieves the unit (attribute finder `pip.hospital_units.by_patientid`) by the requested patient id (`action.patientid`) and selects the id of the supervising doctor (`.doctorid`):

```
action.patientid.<pip.hospital_units.by_patientid>.doctorid
```

Attribute finders are described in greater detail below.

Filtering

SAPL provides syntax elements for having a value pass specified **filters** and thereby modifying it.

Filters can only be applied to basic expressions (remember that an expression in parentheses is a basic expression). Filtering is denoted by the `|-` operator after the expression. Which **filter function** is applied in what way can be defined by a **simple filtering component** or by an **extended filtering component**, which consists of several filter statements.

Filter Functions

SAPL provides three **built-in filter functions**:

remove

Removes a whole attribute (key and value pair) of an object or an item of an array without leaving a replacement.

filter.replace(replacement)

Replaces an attribute or an element by the result of evaluating the expression `replacement`.

filter.blacken(disclose_left=0,disclose_right=0,replacement="X")

Replaces each char of a attribute or item (which has to be a string) by `replacement`, leaving `show\_left` chars from the beginning and `show\_right` chars from the end unchanged. By default, no chars are visible and each char is replaced by `X`.

NOTE	`filter.blacken` could be used to reveal only the first digit of the credit card number and replace the other digits by `X`.
NOTE	`filter.replace` and `filter.blacken` are part of the library `filter`. Importing this library through `import filter` makes the functions available under their simple names.

Example

We take the following object:

Object Structure

```
{
    "value" : "aValue",
    "id" : 5
}
```

If `value` is **removed**, the resulting object is `{ "id" : 5 }`.

If instead **filter.replace** is applied to `value` with the Expression `null`, the resulting object is `{ "value" : null, "id" : 5 }`.

If the function **filter.blacken** is applied to `value` without specifying any arguments, the result would be `{ "value" : "XXXXXX", "id" : 5 }`.

Simple Filtering

A simple filter component applies a **filter function** to the preceding value. The syntax is:

```
BasicExpression |- Function
```

`BasicExpression` is evaluated to a value, the function is applied to this value and the result is returned. If no other arguments are passed to the function, the empty parentheses `()` after the function name can be omitted.

In case `BasicExpression` evaluates to an array, the whole array is passed to the filter function. The **keyword** `each` before `Function` can be used to apply the function to each array item instead:

```
Expression |- each Function
```

Example

Let's assume our resource contains an array of credit card numbers:

```
{
    "numbers": [
        "1234123412341234",
        "2345234523452345",
        "3456345634563456"
    ]
}
```

The function `blacken(1)` without any additional parameters takes a string and replaces everything by `X` except the first char. We can receive the blackened numbers through the basic expression `resource.numbers |- each blacken(1)`:

```
[
    "1XXXXXXXXXXXXXXX",
    "2XXXXXXXXXXXXXXX",
    "3XXXXXXXXXXXXXXX"
]
```

Without the keyword `each`, the function `blacken` would be applied to the array itself, resulting in an error (since as stated above, `blacken` can only be applied to a String).

Extended Filtering

Extended filtering can be used to state more precisely how a value will be altered.

E.g., the expression

```
resource |- { @.credit_card : blacken }
```

would return the original resource except for the value of the attribute `credit_card` being blackened.

Extended filtering components consist of one or more **filter statements**. Each filter statement has a target expression and specifies a filter function that shall be applied to the attribute value (or to each of its items if the keyword `each` is used). The basic syntax is:

```
Expression |- {
    FilterStatement,
    FilterStatement,
    ...
}
```

The syntax of a filter statement is:

```
each TargetRelativeExpression : Function
```

`each` is an optional keyword. If used, the `TargetRelativeExpression` has to evaluate to an array. In this case `Function` is applied to each item of that array.

`TargetRelativeExpression` contains a basic relative expression starting with `@`. The character `@` references the result of the evaluation of `Expression`, so attributes of the value to filter can be accessed easily. Mind that attribute finder steps are not allowed at this place. The value of the attribute selected by the target expresion is replaced by the result of the filter function.

The filter statements are applied successively from top to bottom.

WARNING

Some filter functions can be applied to both arrays and other types (e.g. `remove`). Yet there are selection steps resulting in a "helper array" that cannot be modified. If, for instance, `.*` is applied to the object `{"key1" : "value1", "key2" : "value2"}`, the result would be `["value1", "value2"]`. It is not possible to apply a filter function directly to this array because changing the array itself would not have any effect. The array has been constructed merely to hold multiple values for further processing. In this case the policy would **have to** use the keyword `each` and apply the function to each item. Attempting to alter a helper array results in an error.

Custom Filter Functions

Any function available in SAPL can be used in a filter statement, thus it is easy to add custom filter functions.

When used in a filter statement, the value to filter is passed to the function as its first argument. Consequently the arguments specified in the function call are passed as second, third etc. arguments.

NOTE	Assuming a filter function `roundto` should round a value to the closest multiple of a given number, e.g., `207 |- roundto(100)` should return `200`. In its definition the function needs two formal parameters, the first one is taken for the original value and the second one for the number to round to.

Subtemplate

It is possible to define a subtemplate for an array in order to replace each item of the array by this subtemplate. A subtemplate component is an optional part of a basic expression.

E.g., the basic expression

```
resource.patients :: {
    "name" : @.name
}
```

would return the `patients` array from the resource but with each item containing only one attribute `name`.

The subtemplate is denoted after a double colon:

```
Array :: Expression
```

`Expression` represents the replacement template. In this expression, basic relative expressions (starting with `@`) can be used to access the attributes of the current array item. `@` references the array item which is currently being replaced. `Array` has to evaluate to an array. For each item of `Array`, `Expression` is evaluated and the item is replaced by the result.

Example

Given the variable `array` contains the following array:

```
[
    { "id" : 1 },
    { "id" : 2 }
]
```

The basic expression

```
array :: {
    "aKey" : "aValue"
    "identifier" : @.id
}
```

would evaluate to:

```
[
    {"aKey" : "aValue", "identifier" : 1 },
    {"aKey" : "aValue", "identifier" : 2 }
]
```

Authorization Subscription Evaluation

For any authorization subscription the PDP evaluates each top level SAPL document against the subscription and combines the decisions. If a top level document is a policy set, it contains multiple policies which have to be evaluated first. Their decisions are combined to form an evaluation decision for the policy set. Finally a resource might be added to the final result as well as obligations and advice.

The underlying concept assumes that during evaluation, a decision is assigned to each document. This process will be explained in the following sections.

Policy

Evaluating a policy against an authorization subscription means assigning a value of `NOT_APPLICABLE`, `INDETERMINATE`, `PERMIT` and `DENY` to it. The assigned value depends on the result of evaluating the policy's target and condition (which are conditions that can either be `true` or `false`):

Table 2. Policy Evaluation Table

Target Expression	Condition	Policy Value
`false` (not matching)	don't care	`NOT_APPLICABLE`
`true` (matching)	`false`	`NOT_APPLICABLE`
Error	don't care	`INDETERMINATE`
`true` (matching)	*Error*	`INDETERMINATE`
`true` (matching)	`true`	Policy's **Entitlement** (`PERMIT` or `DENY`)

Policy Set

A decision value (`NOT_APPLICABLE`, `INDETERMINATE`, `PERMIT` or `DENY`) can also be assigned to a policy set. This value depends on the result of evaluating the policy set's target expression and the policies contained in the policy set:

Table 3. Policy Set Evaluation Table

Target Expression	Policy Values	Policy Set Value
`false` (not matching)	don't care	`NOT_APPLICABLE`
`true` (matching)	don't care	Result of the **Combining Algorithm** applied to the Policies
Error	don't care	`INDETERMINATE`

Authorization Subscription

The value which is assigned to the authorization subscription, i.e., the final authorization decision to be returned by the PDP, is the result of applying a combining algorithm to the values assigned to all top level SAPL documents.

Finally in case the decision is `PERMIT` and there is a `transform` statement, the transformed resource is added to the authorization decision. Additionally there might be obligation and advice contained in the policies which has to be added to the authorization decision.

Combining Algorithm

There are two layers with possibly multiple decisions which finally need to be consolidated in a single decision:

- A policy set might contain multiple policies evaluating to different decisions. There must be a final decision for the policy set (*Policy Combination*).
- The PDP might know multiple policy sets and policies which may evaluate to different decisions. In the end the PDP has to include a final decision in the SAPL authorization decision (*Document Combination*).

A combining algorithm describes how to come to the final decision. Both the PDP itself and each policy set have to be configured with a combining algorithm.

Some complexity is added to the algorithms if transformation statements in policies are used: There is no possibility to combine multiple transformation statements. Hence the combining algorithms have to deal with the situation that multiple policies evaluate to `PERMIT` and at least one of them contains a transformation part. In case of such *transformation uncertainty* the decision must not be `PERMIT`.

SAPL provides the following combining algorithms:

- `deny-unless-permit`
- `permit-unless-deny`
- `only-one-applicable`
- `deny-overrides`
- `permit-overrides`
- `first-applicable` (not allowed on PDP level for document combination)

The algorithms work similarly on the PDP and on the policy set level. Thus the following section describes their function in general, using the term *policy document* for a policy and a policy set. If the algorithm is used on the PDP level, a *policy document* could be either a (top level) policy or a policy set, on the policy set level a *policy document* is always a policy.

`deny-unless-permit`

This strict algorithm is used if the decision should be `DENY` except for there is a `PERMIT`. It ensures that any decision is either `DENY` or `PERMIT`.

It works as follows:

1. If any policy document evaluates to `PERMIT` and there is no *transformation uncertainty* (multiple policies evaluate to `PERMIT` and at least one of them has a transformation statement), the decision is `PERMIT`.
2. Otherwise the decision is `DENY`.

`permit-unless-deny`

This generous algorithm is used if the decision should be `PERMIT` except for there is a `DENY`. It ensures that any decision is either `DENY` or `PERMIT`.

It works as follows:

1. If any policy document evaluates to `DENY` or if there is a *transformation uncertainty* (multiple policies evaluate to `PERMIT` and at least one of them has a transformation statement), the decision is `DENY`.
2. Otherwise the decision is `PERMIT`.

`only-one-applicable`

This algorithm is used if policy sets and policies are constructed in a way that multiple policy documents with a matching target are considered an error. A `PERMIT` or `DENY` decision will only be returned if there is exactly one policy set or policy with matching target expression and if this policy document evaluates to `PERMIT` or `DENY`.

It works as follows:

1. If any target evaluation results in an error (`INDETERMINATE`) or if more than one policy documents have a matching target, the decision is `INDETERMINATE`.
2. Otherwise:
    a. If there is no matching policy document, the decision is `NOT_APPLICABLE`.
    b. Otherwise, i.e., there is exactly one matching policy document, the decision is the result of evaluating this policy document.

NOTE	Transformation uncertainty may not occur using the `only-one-applicable` combining algorithm.

`deny-overrides`

This algorithm is used if a `DENY` decision should prevail a `PERMIT` without setting a default decision.

It works as follows:

1. If any policy document evaluates to `DENY`, the decision is `DENY`.
2. Otherwise:
    a. If there is any `INDETERMINATE` or there is a *transformation uncertainty* (multiple policies evaluate to `PERMIT` and at least one of them has a transformation statement), the decision is `INDETERMINATE`.
    b. Otherwise:
        i. If there is any `PERMIT` the decision is `PERMIT`.
        ii. Otherwise the decision is `NOT_APPLICABLE`.

`permit-overrides`

This algorithm is used if a `PERMIT` decision should prevail a `DENY` without setting a default decision.

It works as follows:

1. If any policy document evaluates to `PERMIT` and there is no *transformation uncertainty* (multiple policies evaluate to `PERMIT` and at least one of them has a transformation statement), the

decision is `PERMIT`.

2. Otherwise:
    a. If there is any `INDETERMINATE` or there is a *transformation uncertainty* (multiple policies evaluate to `PERMIT` and at least one of them has a transformation statement), the decision is `INDETERMINATE`.
    b. Otherwise:
        i. If there is any `DENY` the decision is `DENY`.
        ii. Otherwise the decision is `NOT_APPLICABLE`.

`first-applicable`

This algorithm is used if the policy administrator manages the policy's priority by their order in a policy set. As soon as the first policy returns `PERMIT`, `DENY` or `INDETERMINATE`, its result is the final decision. Thus a "default" can be specified by creating a last policy without any conditions. If a decision is found, errors which might occur in later policies are ignored.

Since there is no order in the policy documents known to the PDP, the PDP cannot be configured with this algorithm. `first-applicable` might only be used for policy combination inside a policy set.

It works as follows:

1. Each policy is evaluated in the order specified in the policy set.
    a. If it evaluates to `INDETERMINATE`, the decision is `INDETERMINATE`.
    b. If it evaluates to `PERMIT` or `DENY`, the decision is `PERMIT` or `DENY`
    c. If it evaluates to `NOT_APPLICABLE`, the next policy is evaluated.
2. If no policy with a decision different from `NOT_APPLICABLE` has been found, the decision of the policy set is `NOT_APPLICABLE`.

Transformation

A policy with entitlement `permit` can contain a transformation statement. If the decision is `PERMIT` and there is a policy evaluating to `PERMIT` with transformation, the result of evaluating the expression after the keyword `transform` is returned as `resource` in the authorization decision.

The combining algorithms ensure that transformation is always unambiguous. Consequently, there either is exactly one transformation or none.

Obligation / Advice

Finally obligation and advice might be added to the authorization decision. Both of them can be defined for each individual policy. If a final decision is `PERMIT` there can be multiple policies and policy sets evaluating to `PERMIT`, each of them containing an obligation and/or advice statement - same goes for `DENY`. The final authorization decision with a certain decision has to contain all obligation and advice from policy documents evaluating to this decision.

On the two levels (PDP and policy set), collection of obligation and advice works as follows:

- **Policy Set**: If the policy set evaluates to a certain decision (`PERMIT` or `DENY`), the obligation and advice from all contained policies evaluating to this decision is bundled as the obligation and advice of the policy set.

  (For the combining algorithm `first-applicable` not all policies might be evaluated. A value `PERMIT` or `DENY` is only assigned to evaluated policies. Thus the policy set's obligation and advice merely contains obligation and advice from evaluated policies.)

- **PDP**: If the final decision is `PERMIT` or `DENY`, the obligation and advice from all top level policy documents evaluating to this final decision is collected as the final decision's obligation and advice.

Functions

Functions can be used within SAPL expressions (basic function expressions). A function takes a number of inputs (called *arguments*) and returns an output value.

Functions are organized in function libraries. Each function library has a *name* consisting of one or more identifiers separated by periods `.` (e.g., `simple.string` or `filter`). The *fully qualified name* of a function consists of the library name followed by a period and the function name (e.g., `simple.string.append`).

Functions can be used in any part of a SAPL document, especially in the target expression. Thus their output should only depend on the input arguments and they should not access external resources. Functions do not have access to environment variables.

The Standard Function Library

SAPL will come with a standard function library providing the most important functions.

Custom Function Libraries

The standard functions can be extended by custom functions. Function libraries available in SAPL documents are collected in the PDP's function context. The embedded PDP provides an `AnnotationFunctionContext` where Java classes with annotations can be provided as function libraries:

- To be recognized as a function library, a class has to be annotated with `@FunctionLibrary`. The optional annotation attribute `name` contains the library's name as it will be available in SAPL policies. The attribute value has to be a string consisting of one or more identifiers separated by periods. If the attribute is missing, the name of the Java class is used. The optional annotation attribute `description` contains a string describing the library for documentation purposes.

```
@FunctionLibrary(name = "sample.functions", description = "a sample library")
public class SampleFunctionLibrary {
    ...
}
```

- The annotation `@Function` identifies a function in the library. An optional annotation attribute `name` can contain a function name. The attribute is a string containing an identifier. By default, the name of the Java function will be used. The annotation attribute `docs` can contain a string describing the function.

  ```
  @Function(docs = "returns the length")
  public static JsonNode length(
          @Text parameter
  ) {
      ...
  }
  ```

  Each parameter can be annotated with any number of `@Array`, `@Bool`, `@Int`, `@JsonObject`, `@Long`, `@Number` and `@Text`. The annotations describe which types are allowed for the parameter (in case of multiple annotations, each of these types is allowed).

Attribute Finders

Attribute finders are used to receive attributes which are not included in the authorization subscription context from external PIPs. Just like in `subject.age`, the selection step `.age` selects the attribute `age`s value, `subject.<user.age>` could be used to fetch an `age` attribute which is not included in `subject` but can be obtained from a PIP named `user`.

Attribute finders are organized in libraries as well and follow the same naming conventions as functions, including the use of imports. An attribute finder library constitutes a PIP (e.g., `user`) and can contain any number of attributes (e.g., `age`). They are called by a selection step applied to any value, e.g., `subject.<user.age>`. The attribute finder step receives the previous selection result (in the example: `subject`) and returns the requested attribute.

The concept of attribute finders can be used in a flexible manner: There may be finders which take an object (like in the example above, `subject.<user.age>`) as well as attribute finders which expect a primitive value (e.g., `subject.id.<user.age>` with `id` being a number). In addition attribute finders may also return an object which can be traversed in future selection steps (e.g., `subject.<user.profile>.age`). It is even possible to join multiple attribute finder steps in one expression (e.g., `subject.<user.profile>.supervisor.<user.profile>.age`).

Attribute finders often receive the information from external data sources such as files, databases or HTTP requests which may take a certain amount of time. Therefore they must not be used in a target expression. Attribute finders can access environment variables.

The Standard Attribute Finders

Most attribute finders will be specific to an organization or an environment. E.g., a hospital might provide information about medical units, doctors and patients and use specific attribute finders for this information. Yet SAPL will offer a number of general purpose attribute finders.

Custom Attribute Finders

Attribute finders are functions which take exactly one argument and return any value. Each attribute finder library (also called PIP) has to be known to the PDP. The embedded PDP provides an `AnnotationAttributeContext` which takes Java classes as PIPs.

- To be recognized as a PIP, the class has to be annotated with `@PolicyInformationPoint`. The optional annotation attribute `name` contains the PIP's name as it will be available in SAPL policies. The attribute value is a string consisting of one or more identifiers separated by periods. If the attribute is missing, the name of the Java class is used. The optional annotation attribute `description` contains a string describing the PIP for documentation purposes.

  ```
  @PolicyInformationPoint(name = "user", description = "a sample pip")
  public class SampleUserPIP {
      ...
  }
  ```

- The annotation `@Attribute` identifies an attribute finder, i.e., a function returning an attribute. An optional annotation attribute `name` can contain a name for the attribute. The attribute has to be a string containing an identifier. By default, the name of the function will be used. The annotation attribute `docs` can contain a string describing the attribute.

  ```
  @Attribute(name = "profile", docs = "returns user profile as JSON")
  public static JsonNode userprofile(
          @Number id
  ) {
      ...
  }
  ```

  Each attribute function must have exactly one parameter which can be annotated with any number of `@Array`, `@Bool`, `@Int`, `@JsonObject`, `@Long`, `@Number` and `@Text`. The annotations describe which types are allowed for the argument passed (in case of multiple annotations each of these types is allowed).

Appendix: Formal EBNF grammar

The SAPL syntax is described by the following grammar provided in Extended Backus–Naur form:

```
(* SAPL grammar in EBNF, according to ISO/IEC 14977 *)
```

```
sapl                = { import }, ( policy-set | policy );
import              = "import", ID, { ".", ID }, ".", ( ID | "*" )
                    | "import", { ID, "." }, ID, "as", ID;
policy-set          = "set", STRING, combining-algorithm,
                      [ "for", target-expression ],
                      { value-definition, ";" }, policy, { policy };
combining-algorithm = "deny-overrides" | "permit-overrides"
                    | "first-applicable" | "only-one-applicable"
                    | "deny-unless-permit" | "permit-unless-deny";
target-expression   = expression;
value-definition    = "val", ID, ":=", expression ;
policy              = "policy", STRING, entitlement,
                      [ target-expression ], [ "where", policy-body ],
                      [ "obligation", expression ],
                      [ "advice", expression ],
                      [ "transform" expression ];
entitlement         = "permit" | "deny";
policy-body         = statement, ";", { statement, ";" };
statement           = value-definition | expression;
expression          = addition;
addition            = multiplication,
                      { ( "+" | "-" | "&&" | "&" ), multiplication };
multiplication      = comparison,
                      { ( "*" | "/" | "||" | "|"), comparison };
comparison          = prefixed,[ ( "==" | "=~" | "<" | "<="
                                 | ">=" | ">" | "in" ),
                      prefixed ] ;
prefixed            = [ ( "-" | "!" ) ], basic-expression;
basic-expression    = ( value | "@" | ID | function-call
                      | ( "(", expression, ")" ) ),
                      { selection-step }, [ ("|-", filter)
                      | ( "::", value ) ];
function-call       = ID, { ".", ID }, "(", [ expression,
                      {",", expression } ], ")" ;
selection-step      = key-step | index-step | wildcard-step |
                    | rec-descent-step | rec-wildcard-step
                    | slicing-step | expression-step | condition-step
                    | union-step | attr-finder-step
key-step            = ".", ID
                    | "[", STRING, "]";
index-step          = "[", NUMBER, "]";
wildcard-step       = ".", "*"
                    | "[", "*", "]" ;
rec-descent-step    = "..", ID
                    | "..", "[", STRING, "]"
                    | "..", "[", NUMBER, "]";
rec-wildcard-step   = "..", "*"
                    | "..", "[", "*", "]";
slicing-step        = "[", [ NUMBER ], ":", [ [ NUMBER ],
                      [ ":", [ NUMBER ] ] ], "]";
```

```
expression-step     = "[", "(", expression, ")", "]";
condition-step      = "[", "?", "(", expression, ")", "]";
union-step          = "[", NUMBER, ",", NUMBER, { ",", NUMBER }, "]"
                    | "[", ID, ",", ID, { ",", ID }, "]";
attr-finder-step    = ".", "<", ID, { ".", ID }, ">";
filter              = [ "each" ], filter-function
                    | "{", filter-statement, { ",", filter-statement }, "}";
filter-statement    = [ "each" ], "@", {selection-step}, ":", filter-function;
filter-function     = ID, { ".", ID },
                      [ "(", [ expression, { ",", expression } ], ")" ];
value               = object | array | NUMBER | STRING | "true" | "false"
                    | "null" | "undefined";
object              = "{", [ STRING, ":", expression,
                      { ",", STRING, ":", expression } ], "}";
array               = "[", [ expression, { ",", expression } ], "]" ;
ID                  = ( LETTER | "_" | "$" ), { LETTER | DIGIT | "_" | "$" };
LETTER              = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z" ;
DIGIT               = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
STRING              = '"', ? any character except " ?, '"'
                    | "'", ? any character except ' ?, "'";
NUMBER              = ? JavaScript number definition ?;
```