CDS-based Authorization

Authorization means restricting access to data by adding respective declarations to CDS models, which are then enforced in service implementations. By adding such declarations, we essentially revoke all default access and then grant individual privileges.

Table of Contents

Authentication as Prerequisite

In essence, authentication verifies the user's identity and the presented claims such as granted roles and tenant membership. Briefly, **authentication** reveals *who* uses the service. In contrast, **authorization** controls *how* the user can interact with the application's resources according to granted privileges. As the access control needs to rely on verified claims, authentication is a prerequisite to authorization.

From perspective of CAP, the authentication method is freely customizable. For convenience, a set of authentication methods is supported out of the box to cover most common scenarios:

- XS User and Authentication and Authorization service (XSUAA) is a full-fleged OAuth 2.0 authorization server which allows to protect your endpoints in productive environments. JWT tokens issued by the server not only contain information about the user for authentication, but also assigned scopes and attributes for authorization.

- Identity Authentication Service (IAS) is an OpenId Connect compliant service for next-generation identity and access management. As of today, CAP provides IAS authentication for incoming requests only. Authorization has to be explicitly managed by the application.
- For *local development* and *test* scenario mock user authentication is provided as built-in feature.

Find detailed instructions for setting up authentication in these runtime-specific guides:

- Set up authentication in Node.js.
- Set up authentication in Java.

In *productive* environment with security middleware activated, **all protocol adapter endpoints are authenticated by default**[1], even if no restrictions are configured. Multi-tenant SaaS-applications require authentication to provide tenant isolation out of the box. In case there is the business need to expose open endpoints for anonymous users, it's required to take extra measures depending on runtime and security middleware capabilities.

> [1] Starting with CAP Node.js 6.0.0 resp. CAP Java 1.25.0. *In previous versions endpoints without restrictions are public in single-tenant applications*.

Defining Internal Services

CDS services which are only meant for *internal* usage, shouldn't be exposed via protocol adapters. In order to prevent access from external clients, annotate those services with `@protocol: 'none'`:

```cds
@protocol: 'none'
service InternalService {
  ...
}
```

The `InternalService` service can only receive events sent by in-process handlers.

User Claims

CDS authorization is *model-driven*. This basically means that it binds access rules for CDS model elements to user claims. For instance, access to a service or entity is dependent on the role a user has been assigned to. Or you can even restrict access on an instance level, for example, to the user who created the instance.
The generic CDS authorization is built on a *CAP user concept*, which is an *abstraction* of a concrete user type determined by the platform's identity service. This design decision makes different authentication strategies pluggable to generic CDS authorization.
After successful authentication, a (CAP) user is represented by the following properties:

- Unique (logon) *name* identifying the user. Unnamed users have a fixed name such as `system` or `anonymous`.
- *Tenant* for multitenant applications.
- *Roles* that the user has been granted by an administrator (see User Roles) or that are derived by the authentication level (see Pseudo Roles).
- *Attributes* that the user has been assigned by an administrator.

In the CDS model, some of the user properties can be referenced with the `$user` prefix:

User Property	Reference
Name	`$user`
Attribute (name <attribute>)	`$user.<attribute>`

> A single user attribute can have several different values. For instance, the `$user.language` attribute can contain `['DE','FR']`.

User Roles

As a basis for access control, you can design conceptual roles that are application specific. Such a role should reflect how a user can interact with the application. For instance, the role `Vendor` could describe users who are allowed to read sales articles and update sales figures. In contrast, a `ProcurementManager` can have full access to sales articles. Users can have several roles, that are assigned by an administrative user in the platform's authorization management solution.

> **TIP**
>
> CDS-based authorization deliberately refrains from using technical concepts, such as *scopes* as in *OAuth*, in favor of user roles, which are closer to the conceptual domain of business applications. This also results in much **smaller JWT tokens**.

Pseudo Roles

It's frequently required to define access rules that aren't based on an application-specific user role, but rather on the *authentication level* of the request. For instance, a service could be accessible not only for identified, but also for anonymous (for example, unauthenticated) users. Such roles are called pseudo roles as they aren't assigned by user administrators, but are added at runtime automatically.

The following predefined pseudo roles are currently supported by CAP:

- `authenticated-user` refers to named or unnamed users who have presented a valid authentication claim such as a logon token.
- `system-user` denotes an unnamed user used for technical communication.
- `internal-user` is dedicated to distinguish application internal communication.
- `any` refers to all users including anonymous ones (that means, public access without authentication).

system-user

The pseudo role `system-user` allows you to separate access by *technical* users from access by *business* users. Note that the technical user can come from a SaaS or the PaaS tenant. Such technical user requests typically run in a *privileged* mode without any restrictions on an instance level. For example, an action that implements a data replication into another system needs to access all entities of subscribed SaaS tenants and can't be exposed to any business user. Note that `system-user` also implies `authenticated-user`.

> **TIP**
>
> For XSUAA or IAS authentication, the request user is attached with the pseudo role `system-user` if the presented JWT token has been issued with grant type `client_credentials` or `client_x509` for a trusted client application.

internal-user

Pseudo-role `internal-user` allows to define application endpoints that can be accessed exclusively by the own PaaS tenant (technical communication). The advantage is that similar to `system-user` no technical CAP roles need to be defined to protect such internal endpoints. However, in contrast to `system-user`, the endpoints protected by this pseudo-role do not allow requests from any external technical clients. Hence is suitable for **technical intra-application communication**, see Security > Application Zone.

> **TIP**
>
> For XSUAA or IAS authentication, the request user is attached with the pseudo role `internal-user` if the presented JWT token has been issued with grant type `client_credentials` or `client_x509` on basis of the **identical** XSUAA or IAS service instance.

> **WARNING**
>
> All technical clients that have access to the application's XSUAA or IAS service instance can call your service endpoints as `internal-user`. **Refrain from sharing this service instance with untrusted clients**, for instance by passing services keys or SAP BTP Destination Service instances.

Mapping User Claims

Depending on the configured authentication strategy, CAP derives a *default set* of user claims containing the user's name and attributes:

CAP User Property	XSUAA JWT Property	IAS JWT Property
`$user`	`user_name`	`sub`
`$user.<attribute>`	`xs.user.attributes.<attribute>`	All non-meta attributes

> **TIP**
>
> CAP does not make any assumptions on the presented claims given in the token. String values are copied as they are.

In most cases, CAP's default mapping will match your requirements, but CAP also allows you to customize the mapping according to specific needs. For instance, `user_name` in XSUAA tokens is generally not unique if several customer IdPs are connected to the underlying identity service. Here a combination of `user_name` and `origin` mapped to `$user` might be a feasible solution that you implement in a custom adaptation. Similarly, attribute values can be normalized and prepared for instance-based authorization. Find details and examples how to programmatically redefine the user mapping here:

- Set up Authentication in Node.js.
- Custom Authentication in Java.

> **Be very careful when redefining `$user`**
>
> The user name is frequently stored with business data (for example, `managed` aspect) and might introduce migration efforts. Also consider data protection and privacy regulations when storing user data.

Restrictions

According to authentication, CAP endpoints are closed to anonymous users. But **by default, CDS services have no access control** which means that authenticated users are not restricted. To protect resources according to your business needs, you can define restrictions that make the runtime enforce proper access control. Alternatively, you can add custom authorization logic by means of an authorization enforcement API.

Restrictions can be defined on *different CDS resources*:

- Services
- Entities
- (Un)bound actions and functions

You can influence the scope of a restriction by choosing an adequate hierarchy level in the CDS model. For instance, a restriction on the service level applies to all entities in the service. Additional restrictions on entities or actions can further limit authorized requests. See combined restrictions for more details.

Beside the scope, restrictions can limit access to resources with regards to *different dimensions*:

- The event of the request, that is, the type of the operation (what?)
- The roles of the user (who?)
- Filter-condition on instances to operate on (which?)

@readonly and @insertonly

Annotate entities with `@readonly` or `@insertonly` to statically restrict allowed operations for **all** users as demonstrated in the example:

```cds
service BookshopService {
  @readonly entity Books {...}
  @insertonly entity Orders {...}
}
```

Note that both annotations introduce access control on an entity level. In contrast, for the sake of input validation, you can also use `@readonly` on a property level.

In addition, annotation `@Capabilities` from standard OData vocabulary is enforced by the runtimes analogously:

```cds
service SomeService {
  @Capabilities: {
    InsertRestrictions.Insertable: true,
    UpdateRestrictions.Updatable: true,
    DeleteRestrictions.Deletable: false
  }
  entity Foo { key ID : UUID }
}
```

Events to Auto-Exposed Entities

In general, entities can be exposed in services in different ways: they can be **explicitly exposed** by the modeler (for example, by a projection), or they can be **auto-exposed** by the CDS compiler due to some reason. Access to auto-exposed entities needs to be controlled in a specific way. Consider the following example:

```cds
context db {
  @cds.autoexpose
  entity Categories : cuid { // explicitly auto-exposed (by @cds.autoexpose)
    ...
  }

  entity Issues : cuid { // implicitly auto-exposed (by composition)
    category: Association to Categories;
    ...
  }

  entity Components : cuid { // explicitly exposed (by projection)
    issues: Composition of many Issues;
    ...
  }
```

```
}

service IssuesService {
  entity Components as projection on db.Components;
}
```

As a result, the `IssuesService` service actually exposes *all* three entities from the `db` context:

- `db.Components` is explicitly exposed due to the projection in the service.
- `db.Issues` is implicitly auto-exposed by the compiler as it is a composition entity of `Components`.
- `db.Categories` is explicitly auto-exposed due to the `@cds.autoexpose` annotation.

In general, **implicitly auto-exposed entities cannot be accessed directly**, that means, only access via a navigation path (starting from an explicitly exposed entity) is allowed.

In contrast, **explicitly auto-exposed entities can be accessed directly, but only as `@readonly`**. The rationale behind that is that entities representing value lists need to be readable at the service level, for instance to support value help lists.

See details about `@cds.autoexpose` in Auto-Exposed Entities.

This results in the following access matrix:

Request	*READ*	*WRITE*
`IssuesService.Components`	✓	✓
`IssuesService.Issues`	✗	✗
`IssuesService.Categories`	✓	✗
`IssuesService.Components[<id>].issues`	✓	✓
`IssuesService.Components[<id>].issues[<id>].category`	✓	✗

> **TIP**
>
> CodeLists such as `Languages`, `Currencies`, and `Countries` from `sap.common` are annotated with `@cds.autoexpose` and so are explicitly auto-exposed.

@requires

You can use the `@requires` annotation to control which (pseudo-)role a user requires to access a resource:

```cds
annotate BrowseBooksService with @(requires: 'authenticated-user');
annotate ShopService.Books with @(requires: ['Vendor', 'ProcurementManager'])
annotate ShopService.ReplicationAction with @(requires: 'system-user');
```

In this example, the `BrowseBooksService` service is open for authenticated but not for anonymous users. A user who has the `Vendor` or `ProcurementManager` role is allowed to access the `ShopService.Books` entity. Unbound action `ShopService.ReplicationAction` can only be triggered by a technical user.

> **TIP**
>
> When restricting service access through `@requires`, the service's metadata endpoints (that means, `/$metadata` as well as the service root `/`) are restricted by default as well. If you require public metadata, you can disable the check with a custom express middleware using the privileged user (Node.js) or through config `cds.security.authentication.authenticateMetadataEndpoints = false` ☼ (Java), respectively. Please be aware that the `/$metadata` endpoint is *not* checking for authorizations implied by `@restrict` annotation.

@restrict

You can use the `@restrict` annotation to define authorizations on a fine-grained level. In essence, all kinds of restrictions that are based on static user roles, the request operation, and instance filters can be expressed by this annotation.
The building block of such a restriction is a single **privilege**, which has the general form:

```cds
{ grant:<events>, to:<roles>, where:<filter-condition> }
```

whereas the properties are:

- `grant` : one or more events that the privilege applies to
- `to` : one or more user roles that the privilege applies to (optional)

- *where* : a filter condition that further restricts access on an instance level (optional).

The following values are supported:

- *grant* accepts all standard CDS events (such as *READ* , *CREATE* , *UPDATE* , and *DELETE*) as well as action and function names. *WRITE* is a virtual event for all standard CDS events with write semantic (*CREATE* , *DELETE* , *UPDATE* , *UPSERT*) and * is a wildcard for all events.
- The *to* property lists all user roles or pseudo roles that the privilege applies to. Note that the *any* pseudo-role applies for all users and is the default if no value is provided.
- The *where* -clause can contain a Boolean expression in CQL-syntax that filters the instances that the event applies to. As it allows user values (name, attributes, etc.) and entity data as input, it's suitable for *dynamic authorizations based on the business domain*. Supported expressions and typical use cases are presented in instance-based authorization.

A privilege is met, if and only if **all properties are fulfilled** for the current request. In the following example, orders can only be read by an *Auditor* who meets *AuditBy* element of the instance:

```cds
entity Orders @(restrict: [
    { grant: 'READ', to: 'Auditor', where: (AuditBy = $user) }
]) {/*...*/}
```

If a privilege contains several events, only one of them needs to match the request event to comply with the privilege. The same holds, if there are multiple roles defined in the *to* property:

```cds
entity Reviews @(restrict: [
    { grant:['READ', 'WRITE'], to: ['Reviewer', 'Customer'] }
]) {/*...*/}
```

In this example, all users that have the *Reviewer* *or* *Customer* role can read *or* write to *Reviews* .

You can build restrictions based on *multiple privileges*:

```cds
entity Orders @(restrict: [
    { grant: ['READ','WRITE'], to: 'Admin' },
```

```
    { grant: 'READ', where: (buyer = $user) }
  ]) {/*...*/}
```

A request passes such a restriction **if at least one of the privileges is met**. In this example, `Admin` users can read and write the `Orders` entity. But a user can also read all orders that have a `buyer` property that matches the request user.

Similarly, the filter conditions of matched privileges are combined with logical OR:

```cds
entity Orders @(restrict: [
    { grant: 'READ', to: 'Auditor', where: (country = $user.country) },
    { grant: ['READ','WRITE'], where: (CreatedBy = $user) },
  ]) {/*...*/}
```

Here an `Auditor` user can read all orders with matching `country` or that they have created.

> Annotations such as @requires or @readonly are just convenience shortcuts for @restrict, for example:

- `@requires: 'Viewer'` is equivalent to `@restrict: [{grant:'*', to: 'Viewer'}]`
- `@readonly` is the same as `@restrict: [{ grant:'READ' }]`

Currently, the security annotations **are only evaluated on the target entity of the request**. Restrictions on associated entities touched by the operation aren't regarded. This has the following implications:

- Restrictions of (recursively) expanded or inlined entities of a `READ` request aren't checked.
- Deep inserts and updates are checked on the root entity only.

↳ *See solution sketches for information about how to deal with that.*

Supported Combinations with CDS Resources

Restrictions can be defined on different types of CDS resources, but there are some limitations with regards to supported privileges:

CDS Resource	`grant`	`to`	`where`	Remark
service	n/a	✓	n/a	`= @requires`
entity	✓	✓	✓[1]	

CDS Resource	*grant*	*to*	*where*	Remark
action/function	n/a	✓	n/a[2]	= *@requires*

> [1]For bound actions and functions that aren't bound against a collection, Node.js supports instance-based authorization at the entity level. For example, you can use *where* clauses that *contain references to the model*, such as *where: CreatedBy = $user* . For all bound actions and functions, Node.js supports simple static expressions at the entity level that *don't have any reference to the model*, such as *where: $user.level = 2* .[2] For unbound actions and functions, Node.js supports simple static expressions that *don't have any reference to the model*, such as *where: $user.level = 2* .

Unsupported privilege properties are ignored by the runtime. Especially, for bound or unbound actions, the *grant* property is implicitly removed (assuming *grant: '*'* instead). The same also holds for functions:

```cds
service CatalogService {
  entity Products as projection on db.Products { ... }
  actions {
    @(requires: 'Admin')
    action addRating (stars: Integer);
  }
  function getViewsCount @(restrict: [{ to: 'Admin' }]) () returns Integer;
}
```

Combined Restrictions

Restrictions can be defined on different levels in the CDS model hierarchy. Bound actions and functions refer to an entity, which in turn refers to a service. Unbound actions and functions refer directly to a service. As a general rule, **all authorization checks of the hierarchy need to be passed** (logical AND). This is illustrated in the following example:

```cds
service CustomerService @(requires: 'authenticated-user') {
  entity Products @(restrict: [
    { grant: 'READ' },
    { grant: 'WRITE', to: 'Vendor' },
    { grant: 'addRating', to: 'Customer'}
  ]) {/*...*/}
```

```
  actions {
     action addRating (stars: Integer);
  }
  entity Orders @(restrict: [
    { grant: '*', to: 'Customer', where: (CreatedBy = $user) }
  ]) {/*...*/}
  action monthlyBalance @(requires: 'Vendor') ();
}
```

> The privilege for the *addRating* action is defined on an entity level.

The resulting authorizations are illustrated in the following access matrix:

Operation	*Vendor*	*Customer*	*authenticated-user*	no au
CustomerService.Products (*READ*)	✓	✓	✓	✗
CustomerService.Products (*WRITE*)	✓	✗	✗	✗
CustomerService.Products.addRating	✗	✓	✗	✗
CustomerService.Orders (*)	✗	✓[1]	✗	✗
CustomerService.monthlyBalance	✓	✗	✗	✗

> [1] A *Vendor* user can only access the instances that they created.

The example models access rules for different roles in the same service. In general, this is *not recommended* due to the high complexity. See best practices for information about how to avoid this.

Draft Mode

Basically, the access control for entities in draft mode differs from the general restriction rules that apply to (active) entities. A user, who has created a draft, should also be able to edit (*UPDATE*) or cancel the draft (*DELETE*). The following rules apply:

- If a user has the privilege to create an entity (*CREATE*), he or she also has the privilege to create a **new** draft entity and update, delete, and activate it.
- If a user has the privilege to update an entity (*UPDATE*), he or she also has the privilege to **put it into draft mode** and update, delete, and activate it.

- Draft entities can only be edited by the creator user.
    - In the Node.js runtime, this includes calling bound actions/functions on the draft entity.

> **TIP**
>
> As a result of the derived authorization rules for draft entities, you don't need to take care of draft events when designing the CDS authorization model.

Auto-Exposed and Generated Entities

In general, **a service actually exposes more than the explicitly modeled entities from the CDS service model**. This stems from the fact that the compiler auto-exposes entities for the sake of completeness, for example, by adding composition entities. Another reason is generated entities for localization or draft support that need to appear in the service. Typically, such entities don't have restrictions. The emerging question is, how can requests to these entities be authorized?

For illustration, let's extend the service `IssuesService` from Events to Auto-Exposed Entities by adding a restriction to `Components`:

```cds
annotate IssuesService.Components with @(restrict: [
  { grant: '*', to: 'Supporter' },
  { grant: 'READ', to: 'authenticated-user' } ]);
```

Basically, users with the `Supporter` role aren't restricted, whereas authenticated users can only read the `Components`. But what about the auto-exposed entities such as `IssuesService.Issues` and `IssuesService.Categories`? They could be a target of an (indirect) request as outlined in Events to Auto-Exposed Entities, but none of them are annotated with a concrete restriction. In general, the same also holds for service entities, which are generated by the compiler, for example, for localization or draft support.

To close the gap with auto-exposed and generated entities, the authorization of such entities is delegated to a so-called **authorization entity**, which is the last entity in the request path, which bears authorization information, that means, which fulfills at least one of the following properties:

- Explicitly exposed in the service
- Annotated with a concrete restriction
- Annotated with `@cds.autoexpose`

So, the authorization for the requests in the example is delegated as follows:

Request Target	Authorization Entity
`IssuesService.Components`	n/a[1]
`IssuesService.Issues`	n/a[1]
`IssuesService.Categories`	*`IssuesService.Categories`* [2]
`IssuesService.Components[<id>].issues`	*`IssuesService.Components`* [3]
`IssuesService.Components[<id>].issues[<id>].category`	*`IssuesService.Categories`* [2]

[1] Request is rejected.
[2] *`@readonly`* due to *`@cds.autoexpose`*
[3] According to the restriction. *`<id>`* is relevant for instance-based filters.

Inheritance of Restrictions

Service entities inherit the restriction from the database entity, on which they define a projection. An explicit restriction defined on a service entity *replaces* inherited restrictions from the underlying entity.

Entity *`Books`* on a database level:

```cds
namespace db;
entity Books @(restrict: [
  { grant: 'READ', to: 'Buyer' },
]) {/*...*/}
```

Services *`BuyerService`* and *`AdminService`* on a service level:

```cds
service BuyerService @(requires: 'authenticated-user'){
  entity Books as projection on db.Books; /* inherits */
}

service AdminService @(requires: 'authenticated-user'){
  entity Books @(restrict: [
    { grant: '*', to: 'Admin'} /* overrides */
  ]) as projection on db.Books;
}
```

Events	*Buyer*	*Admin*	*authenticated-user*
BuyerService.Books (*READ*)	✓	✗	✗
AdminService.Books (*)	✗	✓	✗

TIP

We recommend defining restrictions on a database entity level only in exceptional cases. Inheritance and override mechanisms can lead to an unclear situation.

Warning

A service level entity can't inherit a restriction with a *where* condition that doesn't match the projected entity. The restriction has to be overridden in this case.

Instance-Based Authorization

The restrict annotation for an entity allows you to enforce authorization checks that statically depend on the event type and user roles. In addition, you can define a *where* -condition that further limits the set of accessible instances. This condition, which acts like a filter, establishes an *instance-based authorization*.

The condition defined in the *where* -clause typically associates domain data with static user claims. Basically, it *either filters the result set in queries or accepts only write operations on instances that meet the condition*. This means that, the condition applies to following standard CDS events only[1]:

- *READ* (as result filter)
- *UPDATE* (as reject condition[2])
- *DELETE* (as reject condition[2])

[1] Node.js supports *static expressions* that *don't have any reference to the model* such as *where: $user.level = 2* for all events. [2] CAP Java uses a filter condition by default.

For instance, a user is allowed to read or edit `Orders` (defined with the `managed` aspect) that they have created:

```cds
annotate Orders with @(restrict: [
  { grant: ['READ', 'UPDATE', 'DELETE'], where: (CreatedBy = $user) } ]);
```

Or a `Vendor` can only edit articles on stock (that means `Articles.stock` positive):

```cds
annotate Articles with @(restrict: [
  { grant: ['UPDATE'], to: 'Vendor',  where: (stock > 0) } ]);
```

You can define `where` -conditions in restrictions based on CQL-where-clauses. Supported features are:

- Predicates with arithmetic operators.
- Combining predicates to expressions with `and` and `or` logical operators.
- Value references to constants, user attributes, and entity data (elements including paths)
- Exists predicate based on subselects.

Avoid enumerable keys

In case the filter condition is not met in an `UPDATE` or `DELETE` request, the runtime rejects the request (response code 403) even if the user is not even allowed to read the entity. To avoid to disclosure the existence of such entities to unauthorized users, make sure that the key is not efficiently enumerable.

User Attribute Values

To refer to attribute values from the user claim, prefix the attribute name with ' `$user.` ' as outlined in static user claims. For instance, `$user.country` refers to the attribute with the name `country` .

In general, `$user.<attribute>` contains a **list of attribute values** that are assigned to the user. The following rules apply:

- A predicate in the `where` clause evaluates to `true` if one of the attribute values from the list matches the condition.

- An empty (or not defined) list means that the user is fully restricted with regards to this attribute (that means that the predicate evaluates to `false`).

For example, the condition `where: $user.country = countryCode` will grant a user with attribute values `country = ['DE', 'FR']` access to entity instances that have `countryCode = DE` or `countryCode = FR`. In contrast, the user has no access to any entity instances if the value list of country is empty or the attribute is not available at all.

Unrestricted XSUAA Attributes

By default, all attributes defined in XSUAA instances require a value (`valueRequired:true`) which is well-aligned with the CAP runtime that enforces restrictions on empty attributes. If you explicitly want to offer unrestricted attributes to customers, you need to do the following:

1. Switch your XSUAA configuration to `valueRequired:false`
2. Adjust the filter-condition accordingly, for example: `where: $user.country = countryCode or $user.country is null`.

> If `$user.country` is undefined or empty, the overall expression evaluates to `true` reflecting the unrestricted attribute.

> **WARNING**
>
> Refrain from unrestricted XSUAA attributes as they need to be designed very carefully as shown in the following example.

Consider this bad example with *unrestricted* attribute `country` (assuming `valueRequired:false` in XSUAA configuration):

```cds
service SalesService @(requires: ['SalesAdmin', 'SalesManager']) {
  entity SalesOrgs @(restrict: [
     { grant: '*',
       to: ['SalesAdmin', 'SalesManager'],
       where: ($user.country = countryCode or $user.country is null) } ]) {
     countryCode: String; /*...*/
  }
}
```

Let's assume a customer creates XSUAA roles `SalesManagerEMEA` with dedicated values (`['DE', 'FR', ...]`) and 'SalesAdmin' with *unrestricted* values. As expected, a user

assigned only to 'SalesAdmin' has access to all `SalesOrgs`. But when role `SalesManagerEMEA` is added, *only* EMEA orgs are accessible suddenly!

The preferred way is to model with restricted attribute `country` (`valueRequired:true`) and an additional grant:

```cds
service SalesService @(requires: ['SalesAdmin', 'SalesManager']) {
  entity SalesOrgs @(restrict: [
     { grant: '*',
       to: 'SalesManager',
       where: ($user.country = countryCode) },
     { grant: '*',
       to: 'SalesAdmin' } ]) {
     countryCode: String; /*...*/
  }
}
```

Exists Predicate

In many cases, the authorization of an entity needs to be derived from entities reachable via association path. See domain-driven authorization for more details. You can leverage the `exists` predicate in `where` conditions to define filters that directly apply to associated entities defined by an association path:

```cds
service ProjectService @(requires: 'authenticated-user') {
  entity Projects @(restrict: [
     { grant: ['READ', 'WRITE'],
       where: (exists members[userId = $user and role = `Editor`]) } ]) {
    members: Association to many Members; /*...*/
  }
  @readonly entity Members {
    key userId  : User;
    key role: String enum { Viewer; Editor; }; /*...*/
  }
}
```

In the `ProjectService` example, only projects for which the current user is a member with role `Editor` are readable and editable. Note that with exception of the user ID (`$user`) **all authorization information originates from the business data**.

Supported features of `exists` predicate:

- Combine with other predicates in the `where` condition (`where: 'exists a1[...] or exists a2[...]`).
- Define recursively (`where: 'exists a1[exists b1[...]]`).
- Use target paths (`where: 'exists a1.b1[...]`).
- Usage of user attributes.

> **WARNING**
>
> Paths *inside* the filter (`where: (exists a1[b1.c = ...])`) are not yet supported.

The following example demonstrates the last two features:

```cds
service ProductsService @(requires: 'authenticated-user') {
 entity Products @(restrict: [
   { grant: '*',
     where: (exists producers.division[$user.division = name])}]): cuid {
    producers : Association to many ProducingDivisions
                on producers.product = $self;
  }
  @readonly entity ProducingDivisions {
    key product : Association to Products;
    key division : Association to Divisions;
  }
  @readonly entity Divisions : cuid {
    name : String;
    producedProducts : Association to many ProducingDivisions
                       on producedProducts.division = $self;
  }
}
```

Here, the authorization of `Products` is derived from `Divisions` by leveraging the *n:m relationship* via entity `ProducingDivisions` . Note that the path `producers.division` in the `exists` predicate points to target entity `Divisions` , where the filter with the user-dependent attribute `$user.division` is applied.

> **Consider Access Control Lists**
>
> Be aware that deep paths might introduce a performance bottleneck. Access Control List (ACL) tables, managed by the application, allow efficient queries and might be the

better option in this case.

Association Paths

The *where* -condition in a restriction can also contain CQL path expressions that navigate to elements of associated entities:

```cds
service SalesOrderService @(requires: 'authenticated-user') {
  entity SalesOrders @(restrict: [
     { grant: 'READ',
       where: (product.productType = $user.productType) } ]) {
    product: Association to one Products;
  }
  entity Products {
    productType: String(32); /*...*/
  }
}
```

Paths on 1:n associations (*Association to many*) are only supported, *if the condition selects at most one associated instance*. It's highly recommended to use the exists predicate instead.

> **TIP**
>
> Be aware of increased execution time when modeling paths in the authorization check of frequently requested entities. Working with materialized views might be an option for performance improvement in this case.

> ***Warning***
>
> In Node.js association paths in *where* -clauses are currently only supported when using SAP HANA.

Best Practices

CAP authorization allows you to control access to your business data on a fine granular level. But keep in mind that the high flexibility can end up in security vulnerabilities if not applied appropriately. In this perspective, lean and straightforward models are preferred. When modeling your access rules, the following recommendations can support you to design such models.

Choose Conceptual Roles

When defining user roles, one of the first options could be to align roles to the available *operations* on entities, which results in roles such as `SalesOrders.Read`, `SalesOrders.Create`, `SalesOrders.Update`, and `SalesOrders.Delete`, etc. What is the problem with this approach? Think about the resulting number of roles that the user administrator has to handle when assigning them to business users. The administrator would also have to know the domain model precisely and understand the result of combining the roles. Similarly, assigning roles to operations only (`Read`, `Create`, `Update`, ...) typically doesn't fit your business needs.
We strongly recommend defining roles that describe **how a business user interacts with the system**. Roles like `Vendor`, `Customer`, or `Accountant` can be appropriate. With this approach, the application developers define the set of accessible resources in the CDS model for each role - and not the user administrator.

Prefer Single-Purposed, Use-Case Specific Services

Have a closer look at this example:

```cds
service CatalogService @(requires: 'authenticated-user') {
   entity Books @(restrict: [
    { grant: 'READ' },
    { grant: 'WRITE', to: 'Vendor', where: ($user.publishers = publisher) },
    { grant: 'WRITE', to: 'Admin' } ])
  as projection on db.Books;
  action doAccounting @(requires: ['Accountant', 'Admin']) ();
}
```

Four different roles (`authenticated-user`, `Vendor`, `Accountant`, `Admin`) *share* the same service - `CatalogService`. As a result, it's confusing how a user can use `Books` or `doAccounting`. Considering the complexity of this small example (4 roles, 1 service, 2 resources), this approach can introduce a security risk, especially if the model is larger

and subject to adaptation. Moreover, UIs defined for this service will likely appear unclear as well.
The fundamental purpose of services is to expose business data in a specific way. Hence, the more straightforward way is to **use a service for each of the roles**:

```cds
@path:'browse'
service CatalogService @(requires: 'authenticated-user') {
  @readonly entity Books
  as select from db.Books { title, publisher, price };
}

@path:'internal'
service VendorService @(requires: 'Vendor') {
  entity Books @(restrict: [
    { grant: 'READ' },
    { grant: 'WRITE', to: 'vendor', where: ($user.publishers = publisher) } ]
  as projection on db.Books;
}

@path:'internal'
service AccountantService @(requires: 'Accountant') {
  @readonly entity Books as projection on db.Books;
  action doAccounting();
}
/*...*/
```

> **TIP**
>
> You can tailor the exposed data according to the corresponding role, even on the level of entity elements like in *`CatalogService.Books`* .

Prefer Dedicated Actions for Specific Use-Cases

In some cases it can be helpful to restrict entity access as much as possible and create actions with dedicated restrictions for specific use cases, like in the following example:

```cds
service GitHubRepositoryService @(requires: 'authenticated-user') {
  @readonly entity Organizations as projection on GitHub.Organizations actions
    @(requires: 'Admin') action rename(newName : String);
    @(requires: 'Admin') action delete();
```

```
  };
}
```

This service allows querying organizations for all authenticated users. In addition, `Admin` users are allowed to rename or delete. Granting `UPDATE` to `Admin` would allow administrators to change organization attributes that aren't meant to change.

Think About Domain-Driven Authorization

Static roles often don't fit into an intuitive authorization model. Instead of making authorization dependent from static properties of the user, it's often more appropriate to derive access rules from the business domain. For instance, all users assigned to a department (in the domain) are allowed to access the data of the organization comprising the department. Relationships in the entity model (for example, a department assignment to organization), influence authorization rules at runtime. In contrast to static user roles, **dynamic roles** are fully domain-driven.

Revisit the ProjectService example, which demonstrates how to leverage instance-based authorization to induce dynamic roles.

Advantages of dynamic roles are:

- The most flexible way to define authorizations
- Induced authorizations according to business domain
- Application-specific authorization model and intuitive UIs
- Decentralized role management for application users (no central user administrator required)

Drawbacks to be considered are:

- Additional effort for modeling and designing application-specific role management (entities, services, UI)
- Potentially higher security risk due to lower use of the framework functionality
- Sharing authorization management with other (non-CAP) applications is harder to achieve
- Dynamic role enforcement can introduce a performance penalty

Control Exposure of Associations and Compositions

Note that exposed associations (and compositions) can disclose unauthorized data. Consider the following scenario:

```cds
namespace db;
entity Employees : cuid { // autoexposed!
  name: String(128);
  team: Association to Teams;
  contract: Composition of Contracts;
}
entity Contracts @(requires:'Manager') : cuid { // autoexposed!
  salary: Decimal;
}
entity Teams : cuid {
  members: Composition of many Employees on members.team = $self;
}


service ManageTeamsService @(requires:'Manager') {
  entity Teams as projection on db.Teams;
}

service BrowseEmployeesService @(requires:'Employee') {
  @readonly entity Teams as projection on db.Teams; // navigate to Contracts!
}
```

A team (entity `Teams`) contains members of type `Employees`. An employee refers to a single contract (entity `Contracts`) which contains sensitive information that should be visible only to `Manager` users. `Employee` users should be able to browse the teams and their members, but aren't allowed to read or even edit their contracts.
As `db.Employees` and `db.Contracts` are auto-exposed, managers can navigate to all instances through the `ManageTeamsService.Teams` service entity (for example, OData request `/ManageTeamsService/Teams?$expand=members($expand=contract)`).
It's important to note that this also holds for an `Employee` user, as **only the target entity `BrowseEmployeesService.Teams` has to pass the authorization check in the generic handler, and not the associated entities**.

To solve this security issue, introduce a new service entity `BrowseEmployeesService.Employees` that removes the navigation to `Contracts` from the projection:

```cds
service BrowseEmployeesService @(requires:'Employee') {
  @readonly entity Employees
  as projection on db.Employees excluding { contracts }; // hide contracts!

  @readonly entity Teams as projection on db.Teams;
}
```

Now, an *Employee* user can't expand the contracts as the composition isn't reachable anymore from the service.

> **TIP**
>
> Associations without navigation links (for example, when an associated entity isn't exposed) are still critical with regards to security.

Design Authorization Models from the Start

As shown before, defining an adequate authorization strategy has a deep impact on the service model. Apart from the fundamental decision, if you want to build your authorizations on dynamic roles, authorization requirements can result in rearranging service and entity definitions completely. In the worst case, this means rewriting huge parts of the application (including the UI). For this reason, it's *strongly* recommended to take security design into consideration at an early stage of your project.

Keep it as Simple as Possible

- If different authorizations are needed for different operations, it's easier to have them defined at the service level. If you start defining them at the entity level, all possible operations must be specified, otherwise the not mentioned operations are automatically forbidden.
- If possible, try to define your authorizations either on the service or on the entity level. Mixing both variants increases complexity and not all combinations are supported either.

Separation of Concerns

Consider using CDS Aspects to separate the actual service definitions from authorization annotations as follows:

services.cds

```cds
service ReviewsService {
  /*...*/
}

service CustomerService {
  entity Orders {/*...*/}
  entity Approval {/*...*/}
}
```

services-auth.cds

```cds
service ReviewsService @(requires: 'authenticated-user'){
  /*...*/
}

service CustomerService @(requires: 'authenticated-user'){
  entity Orders @(restrict: [
    { grant: ['READ','WRITE'], to: 'admin' },
    { grant: 'READ', where: 'buyer = $user' },
  ]){/*...*/}
  entity Approval @(restrict: [
    { grant: 'WRITE', where: '$user.level > 2' }
  ]){/*...*/}
}
```

This keeps your actual service definitions concise and focused on structure only. It also allows you to give authorization models separate ownership and lifecycle.

Programmatic Enforcement

The service provider frameworks **automatically enforce** restrictions in generic handlers. They evaluate the annotations in the CDS models and, for example:

- Reject incoming requests if static restrictions aren't met.
- Add corresponding filters to queries for instance-based authorization, etc.

If generic enforcement doesn't fit your needs, you can override or adapt it with **programmatic enforcement** in custom handlers:

- Authorization Enforcement in Node.js
- Enforcement API & Custom Handlers in Java

Role Assignments with IAS and AMS

The Authorization Management Service (AMS) as part of SAP Cloud Identity Services (SCI) provides libraries and services for developers of cloud business applications to declare, enforce and manage instance based authorization checks. When used together with CAP the AMS "Policies" can contain the CAP roles as well as additional filter criteria for instance based authorizations that can be defined in the CAP model. transformed to AMS policies and later on refined by customers user and authorization administrators in the SCI administration console and assigned to business users.

Use AMS as Authorization Management System on SAP BTP

SAP BTP is currently replacing the authorization management done with XSUAA by an integrated solution with AMS. AMS is integrated into SAP Cloud Identity (SCI), which will offer authentication, authorization, user provisioning and management in one place.

For newly build applications the usage of AMS is generally recommended. The only constraint that comes with the usage of AMS is that customers need to copy their users to the Identity Directory Service as the central place to manage users for SAP BTP applications. This is also the general SAP strategy to simplify user management in the future.

Case For XSUAA

There is one use case where currently an XSUAA based authorization management is preferable: When XSUAA based services to be consumed by a CAP application come

with their own business user roles and thus make user role assignment in the SAP Cloud Cockpit necessary. This will be resolved in the future when the authorization management will be fully based on the SCI Admin console.

For example, SAP Task Center you want to consume an XSUAA-based service that requires own end user role. Apart from this, most services should be technical services that do not require an own authorization management that is not yet integrated in AMS.

↳ *Learn more about using IAS and AMS with CAP Node.js*

Role Assignments with XSUAA

Information about roles and attributes has to be made available to the UAA platform service. This information enables the respective JWT tokens to be constructed and sent with the requests for authenticated users. In particular, the following happens automatically behind-the-scenes upon build:

1. Roles and Attributes Are Filled into the XSUAA Configuration

Derive scopes, attributes, and role templates from the CDS model:

```sh
cds add xsuaa
```

This generates an *xs-security.json* file:

xs-security.json

```json
{
  "scopes": [
    { "name": "$XSAPPNAME.admin", "description": "admin" }
  ],
  "attributes": [
    { "name": "level", "description": "level", "valueType": "s" }
  ],
  "role-templates": [
    { "name": "admin", "scope-references": [ "$XSAPPNAME.admin" ], "descripti
```

```
  ]
}
```

For every role name in the CDS model, one scope and one role template are generated with the exact name of the CDS role.

> **Re-generate on model changes**
>
> You can have such a file re-generated via
>
> ```sh
> cds compile srv --to xsuaa > xs-security.json
> ```

See Application Security Descriptor Configuration Syntax in the SAP HANA Platform documentation for the syntax of the *xs-security.json* and advanced configuration options.

> **Avoid invalid characters in your models**
>
> Roles modeled in CDS may contain characters considered invalid by the XSUAA service.

If you modify the *xs-security.json* manually, make sure that the scope names in the file exactly match the role names in the CDS model, as these scope names will be checked at runtime.

2. XSUAA Configuration Is Completed and Published

Through MTA Build

If there's no *mta.yaml* present, run this command:

```sh
cds add mta
```

▶ *See what this does in the background...*

Inline configuration in the *mta.yaml* `config` block and the *xs-security.json* file are merged. If there are conflicts, the MTA security configuration has priority.

↳ *Learn more about **building and deploying MTA applications**.*

3. Assembling Roles and Assigning Roles to Users

This is a manual step an administrator would do in SAP BTP Cockpit. See Set Up the Roles for the Application for more details. If a user attribute isn't set for a user in the IdP of the SAP BTP Cockpit, this means that the user has no restriction for this attribute. For example, if a user has no value set for an attribute "Country", they're allowed to see data records for all countries. In the *xs-security.json*, the `attribute` entity has a property `valueRequired` where the developer can specify whether unrestricted access is possible by not assigning a value to the attribute.

4. Scopes Are Narrowed to Local Roles

Based on this, the JWT token for an administrator contains a scope `my.app.admin`. From within service implementations of `my.app` you can reference the scope:

```js
req.user.is ("admin")
```

... and, if necessary, from others by:

```js
req.user.is ("my.app.admin")
```

> See the following sections for more details:
>
> - Developing Security Artifacts in SAP BTP
> - Maintaining Application Security in XS Advanced