

Domain Modeling

Domain Models capture the static, data-related aspects of a problem domain in terms of entity-relationship models. They serve as the basis for *persistence models* deployed to databases as well as for *service definitions*.

Table of Contents

- [Introduction](#)
 - [Capture Intent — What, not How!](#)
 - [Entity-Relationship Modeling](#)
 - [Aspect-oriented Modeling](#)
 - [Fuelling Generic Providers](#)
 - [Domain-Driven Design](#)
- [Best Practices](#)
 - [Keep it Simple, Stupid](#)
 - [Separation of Concerns](#)
 - [Naming Conventions](#)
- [Core Concepts](#)
 - [Namespaces](#)
 - [Domain Entities](#)
 - [Primary Keys](#)
 - [Data Types](#)
 - [Associations](#)
 - [Compositions](#)
- [Aspects](#)
 - [Authorization](#)
 - [Fiori Annotations](#)

- Localized Data
- Managed Data
 - Annotation: `@cds.on.insert`
 - Annotation: `@cds.on.update`
 - Aspect managed
- Pseudo Variables

Introduction

Capture Intent — *What, not How!*

CDS focuses on *conceptual modelling*: we want to capture intent, not imperative implementations — that is: What, not How. Not only does that keep domain models concise and comprehensible, it also allows us to provide optimized generic implementations.

For example, given an entity definition like that:

```
using { cuid, managed } from '@sap/cds/common';  
entity Books : cuid, managed {  
    title : localized String;  
    descr : localized String;  
    author : Association to Authors;  
}
```

In that model we used the **pre-defined aspects** `cuid` and `managed`, as well as the **qualifier** `localized` to capture generic aspects. We also used **managed associations**.

In all these cases, we focus on capturing our intent, while leaving it to generic implementations to provide best-possible implementations.

Entity-Relationship Modeling

Entity-Relationship Modelling (ERM) is likely the most widely known and applied conceptual modelling technique for data-centric applications. It is also one of the

foundations for CDS.

Assume we had been given this requirement:

"We want to create a bookshop allowing users to browse **Books** and **Authors**, and navigate from Books to Authors and vice versa. Books are classified by **Genre**".

Using CDS, we would translate that into an initial domain model as follows:

```
using { cuid } from '@sap/cds/common';cds

entity Books : cuid {
    title : String;
    descr : String;
    genre : Genre;
    author : Association to Authors;
}

entity Authors : cuid {
    name : String;
    books : Association to many Books on books.author = $self;
}

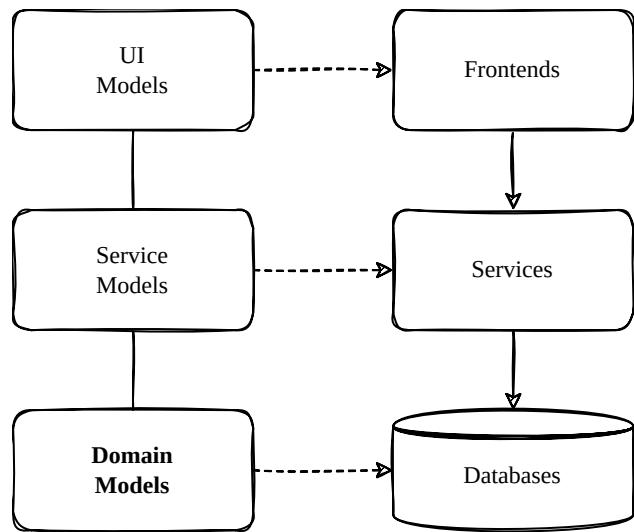
type Genre : String enum {
    Mystery; Fiction; Drama;
}
```

Aspect-oriented Modeling

CDS Aspects and Annotations provide powerful means for **separation of concerns**. This greatly helps to keep our core domain model clean, while putting secondary concerns into separate files and model fragments. → Find details in chapter **Aspects** below.

Fuelling Generic Providers

As depicted in the illustration below, domain models serve as the sources for persistence models, deployed to databases, as well as the underlying model for services acting as API facades to access data.



The more we succeeded in capturing intent over imperative implementations, the more we can provide optimized generic implementations.

Domain-Driven Design

CAP shares these goals and approaches with [Domain-driven Design](#) :

1. Placing projects' primary **focus on the core domain**
2. Close collaboration of **developers** and **domain experts**
3. Iteratively refining **domain knowledge**

We use CDS as our ubiquitous modelling language, with CDS Aspects giving us the means to separate core domain aspects from generic aspects. CDS's human-readable nature fosters collaboration of developers and domain experts.

As CDS models are used to fuel generic providers — the database as well as application services — we ensure the models are applied in the implementation. And as coding is minimized we can more easily refine and revise our models, without having to refactor large boilerplate code based.

Best Practices

Keep it Simple, Stupid

Domain modeling is a means to an end; your clients and consumers are the ones who have to understand and work with your models the most, much more than you as their creator. Keep that in mind and understand the task of domain modeling as a service to others.

Keep models **concise** and **comprehensible**

As said in the "[Keep it simple, stupid!](#)" Wikipedia entry: "... most systems work best if they're kept simple rather than made complicated; therefore, simplicity should be a key goal in design , and unnecessary complexity should be avoided."

Avoid overly abstract models

Even though domain models should abstract from technical implementations, don't overstress this and balance it with ease of adoption. For example if the vast majority of your clients use relational databases, don't try to overly abstract from that, as that would have all suffer from common denominator syndromes.

Prefer Flat Models

While CDS provides great support, you should always think twice before using structured types. Some technologies you or your customers use might not integrate with those out of the box. Moreover, flat structures are easier to understand and consume.

Good:

```
entity Contacts {  
    isCompany : Boolean;  
    company   : String;  
    title     : String;  
    firstname : String;  
    lastname  : String;  
}  
cds
```

Bad:

```
entity Contacts {  
    isCompany   : Boolean;  
    companyData : CompanyDetails;  
    personData  : PersonDetails;  
}  
type CompanyDetails {  
cds
```

```
    name : String;  
}  
  
type PersonDetails {  
    titles : AcademicTitles;  
    name : PersonName;  
}  
  
type PersonName : {  
    first : String;  
    last : String;  
}  
  
type AcademicTitles : {  
    primary : String;  
    secondary : String;  
}
```

Separation of Concerns

As highlighted with a few samples in the chapter above, always strive to keep your core domain model clean, concise and comprehensible.

CDS Aspects help you to do so, by decomposing models and definitions into separate files with potentially different life cycles, contributed by different *people*.

We strongly recommend to make use of that as much as possible.

Naming Conventions

We recommend adopting the following simple naming conventions as commonly used in many communities, for example, Java, JavaScript, C, SQL, etc.

To easily distinguish type / entity names from elements names we recommend to...

Capitalize *Type / Entity* Names

- Start *entity* and *type* names with capital letters — for example, *Authors*
- Start *elements* with a lowercase letter — for example, *name*

As entities represent not only data types, but also data sets, from which we can read from, we recommend following common SQL convention:

Pluralize *Entity* Names

- Use plural form for *entities* — for example, *Authors*
- Use singular form for *types* — for example, *Genre*

In general always prefer conciseness, comprehensibility and readability, and avoid overly lengthy names, probably dictated by overly strict systematics:

Prefer Concise Names

- Don't repeat contexts → for example *Authors.name* instead of *Authors.authorName*
- Prefer one-word names → for example *address* instead of *addressInformation*
- Use *ID* for technical primary keys → see also [Use Canonic Primary Keys](#)

Core Concepts

Namespaces

You can use **namespaces** to get to unique names without bloating your code with fully qualified names. For example:

```
namespace foo.bar;                                cds
entity Boo {}
entity Moo : Boo {}
```

... is equivalent to:

```
entity foo.bar.Boo {}
entity foo.bar.Moo : foo.bar.Boo {}                cds
```

Note:

- **Namespaces are just prefixes** — which are automatically applied to all relevant names in a file. Beyond this there's nothing special about them.
- **Namespaces are optional** — use namespaces if your models might be reused in other projects; otherwise, you can go without namespaces.
- The **reverse domain name** approach works well for choosing namespaces.

WARNING

Avoid short-lived ingredients in namespaces, or names in general, such as your current organization's name, or project code names.

Domain Entities

Entities represent a domain's data. When translated to persistence models, especially relational ones, entities become tables.

Entity definitions essentially declare structured types with named and typed elements, plus the **primary key** elements used to identify entries.

```
entity name {  
    key element1 : Type;  
    element2 : Type;  
    ...  
}
```

cds

↳ Learn more about entity definitions.

Views / Projections

Borrowing powerful view building from SQL, we can declare entities as (denormalized) views on other entities:

```
entity ProjectedEntity as select from BaseEntity {  
    element1, element2 as name, /*...*/  
};
```

cds

↳ Learn more about views and projections.

Primary Keys

Use the keyword `key` to signify one or more elements that form an entity's primary key:

```
entity Books {  
    key ID : UUID;  
    ***  
}
```

Do:

- Prefer *simple, technical* primary keys
- Prefer *canonic* primary keys
- Prefer *UUIDs* for primary keys

Don't:

- Don't use binary data as keys!
- **Don't interpret UUIDs!**

Prefer Simple, Technical Keys

While you can use arbitrary combinations of fields as primary keys, keep in mind that primary keys are frequently used in joins all over the place. And the more fields there are to compare for a join the more you'll suffer from poor performance. So prefer primary keys consisting of single fields only.

Moreover, primary keys should be immutable, that means once assigned on creation of a record they should not change subsequently, as that would break references you might have handed out. Think of them as a fingerprint of a record.

Prefer Canonic Keys

We recommend using canonically named and typed primary keys, as promoted **by aspect `cuid` from `@sap/cds/common`**.

```
// @sap/cds/common  
aspect cuid { key ID : UUID }  
  
using { cuid } from '@sap/cds/common';  
entity Books : cuid { ... }  
entity Authors : cuid { ... }
```

This eases the implementation of generic functions that can apply the same ways of addressing instances across different types of entities.

Prefer UUIDs for Keys

While UUIDs certainly come with an overhead and a performance penalty when looking at single databases, they have several advantages when we consider the total bill. So, you can avoid **the evil of premature optimization** by at least considering these points:

- **UUIDs are universal** — that means that they're unique across every system in the world, while sequences are only unique in the source system's boundaries.
Whenever you want to exchange data with other systems you'd anyways add something to make your records 'universally' addressable.
- **UUIDs allow distributed seeds** — for example, in clients. In contrast, database sequences or other sequential generators always need a central service, for example, a single database instance and schema. This becomes even more a problem in distributed landscape topologies.
- **Database sequences are hard to guess** — assume that you want to insert a *SalesOrder* with three *SalesOrderItems* in one transaction. `INSERT SalesOrder` will automatically get a new ID from the sequence. How would you get this new ID in order to use it for the foreign keys in subsequent `INSERTs` of the *SalesOrderItems*?
- **Auto-filled primary keys** — primary key elements with type UUID are automatically filled by generic service providers in Java and Node.js upon `INSERT`.

Prefer UUIDs for Keys

Use DB sequences only if you really deal with high data volumes. Otherwise, prefer UUIDs.

You can also have semantic primary keys such as order numbers constructed by customer name+date, etc. And if so, they usually range between UUIDs and DB sequences with respect to the pros and cons listed above.

Don't Interpret UUIDs!

It is an unfortunate anti pattern to validate UUIDs, such as for compliance to [RFC 4122](#). This not only means useless processing, it also impedes integration with existing data sources. For example, ABAP's **GUID_32s** are uppercase without hyphens.

UUIDs are unique opaque values! — The only assumption required and allowed is that UUIDs are unique so that they can be used for lookups and compared by equality —

nothing else! It's the task of the UUID generator to ensure uniqueness, not the task of subsequent processors!

On the same note, converting UUID values obtained as strings from the database into binary representations such as `java.lang.UUID`, only to render them back to strings in responses to HTTP requests, is useless overhead.

WARNING

- Avoid unnecessary assumptions, for example, about uppercase or lowercase
- Avoid useless conversions, for example, from strings to binary and back
- Avoid useless validations of UUID formats, for example, about hyphens

↳ See also: [Mapping UUIDs to OData](#)

↳ See also: [Mapping UUIDs to SQL](#)

Data Types

Standard Built-in Types

CDS comes with a small set of built-in types:

- `UUID`,
- `Boolean`,
- `Date`, `Time`, `DateTime`, `Timestamp`
- `Integer`, `UInt8`, `Int16`, `Int32`, `Int64`
- `Double`, `Decimal`
- `String`, `LargeString`
- `Binary`, `LargeBinary`

↳ See list of [Built-in Types](#) in the CDS reference docs.

Common Reuse Types

In addition, a set of common reuse types and aspects is provided with package `@sap/cds/common`, such as:

- Types `Country`, `Currency`, `Language` with corresponding value list entities

- Aspects *cuid* , *managed* , *temporal*

For example, usage is as simple as this:

```
using { Country, managed } from '@sap/cds/common';
entity Addresses : managed { //> using reuse aspect
    street : String;
    town   : String;
    country : Country; //> using reuse type
}
```

cds

↳ Learn more about reuse types provided by `@sap/cds/common`.

Use common reuse types and aspects...

... to keep models concise, and benefitting from improved interoperability, proven best practices, and out-of-the-box support through generic implementations in CAP runtimes.

Custom-defined Types

Declare custom-defined types to increase semantic expressiveness of your models, or to share details and annotations as follows:

```
type User : String; //> merely for increasing expressiveness
type Genre : String enum { Mystery; Fiction; ... }
type DayOfWeek : Number @assert.range:[1,7];
```

cds

Use Custom Types Reasonably

Avoid overly excessive use of custom-defined types. They're valuable when you have a decent **reuse ratio**. Without reuse, your models just become harder to read and understand, as one always has to look up respective type definitions, as in the following example:

```
using { sap.capire.bookshop.types } from './types';
namespace sap.capire.bookshop;
entity Books {
    key ID : types.BookID;
    name : types.BookName;
    descr : types.BookDescr;
```

cds

```
...  
}  
  
// types.cds  
namespace sap.capire.bookshop.types;  
type BookID : UUID;  
type BookName : String;  
type BookDescr : String;
```

cds

Associations

Use *Associations* to capture relationships between entities.

```
entity Books { ...  
    author : Association to Authors; //> to one  
}  
entity Authors { ...  
    books : Association to many Books on books.author = $self;  
}
```

cds

↳ Learn more about *Associations* in the *CDS Language Reference*.

Managed :1 Associations

The association `Books:author` in the sample above is a so-called *managed* association, with foreign key columns and on conditions added automatically behind the scenes.

```
entity Books { ...  
    author : Association to Authors;  
}
```

cds

In contrast to that we could also use *unmanaged* associations with all foreign keys and on conditions specified manually:

```
entity Books { ...  
    author : Association to Authors on author.ID = author_ID;  
    author_ID : type of Authors:ID;  
}
```

cds

Note: To-many associations are unmanaged by nature as we always have to specify an `on` condition. Reason for that is that backlink associations or foreign keys cannot be guessed reliably.

Prefer managed associations

For the sake of conciseness and comprehensibility of your models always prefer *managed Associations* for to-one associations.

To-Many Associations

Simply add the `many` qualifier keyword to indicate a to-many cardinality:

```
entity Authors { ...  
    books : Association to many Books;  
}
```

cds

If your models are meant to target APIs, this is all that is required. When targeting databases though, we need to add an `on` condition, like so:

```
entity Authors { ...  
    books : Association to many Books on books.author = $self;  
}
```

cds

The `on` condition can either compare a backlink association to `$self`, or a backlink foreign key to the own primary key, for example `books.author.ID = ID`.

Many-to-Many Associations

CDS currently doesn't provide dedicated support for *many-to-many* associations. Unless we add some, you have to resolve *many-to-many* associations into two *one-to-many* associations using a link entity to connect both. For example:

```
entity Projects { ...  
    members : Composition of many Members on members.project = $self;  
}  
entity Users { ...  
    projects : Composition of many Members on projects.user = $self;  
}  
entity Members: cuid { // link table  
    project : Association to Projects;
```

cds

```
    user : Association to Users;  
}
```

We can use **Compositions of Aspects** to reduce noise a bit:

```
entity Projects { ...  
  members : Composition of many { key user : Association to Users };  
}  
entity Users { ...  
  projects : Composition of many Projects.members on projects.user = $self;  
}
```



Behind the scenes the equivalent of the model above would be generated, with the link table called `Projects.members` and the backlink association to `Projects` in there called `up_`. Consider that for SAP Fiori elements 'project' and 'user' shall not be keys, even if their combination is unique, because as keys those fields can't be edited on the UI. In this case a different key is required, for example a UUID, and the unique constraint for `project` and `user` can be expressed via `@assert.unique`.

Compositions

Compositions represent contained-in relationships. CAP runtimes provide these special treatments to Compositions out of the box:

- **Deep Insert / Update** automatically filling in document structures
- **Cascaded Delete** is when deleting Composition roots
- **Composition targets** are **auto-exposed** in service interfaces

Modeling Document Structures

Compositions are used to model document structures. For example, in the following definition of `Orders`, the `orders:Items` composition refers to the `OrderItems` entity, with the entries of the latter being fully dependent objects of `Orders`.

```
entity Orders { ...  
  Items : Composition of many OrderItems on Items.parent = $self;  
}  
entity OrderItems { // to be accessed through Orders only  
  key parent : Association to Orders;
```

```

key pos      : Integer;
quantity    : Integer;
}

```

↳ Learn more about Compositions in the CDS Language Reference.

Composition of Aspects

We can use anonymous inline aspects to rewrite the above with less noise as follows:

```

entity Orders { ...
  Items : Composition of many {
    key pos : Integer;
    quantity : Integer;
  };
}

```

↳ Learn more about Compositions of Aspects in the CDS Language Reference.

Behind the scenes this will add an entity named `orders.Items` with a backlink association named `up_`, so effectively generating the same model as above. You can annotate the inline composition with UI annotations as follows:

```

annotate Orders.Items with @(
  UI.LineItem : [
    {Value: pos},
    {Value: quantity},
  ],
);

```

Aspects

CDS's **Aspects** provide powerful mechanisms to separate concerns. It allows decomposing models and definitions into separate files with potentially different life cycles, contributed by different people.

The basic mechanism use the `extend` or `annotate` directives to add secondary aspects to a core domain entity like so:

```
extend Books with {  
    someAdditionalField : String;  
}  
  
annotate Books with @some.entity.level.annotations {  
    title @some.field.level.annotations;  
};
```

cds

cds

Variants of this allow declaring and applying **named aspects** like so:

```
aspect NamedAspect { someAdditionalField : String }  
extend Books with NamedAspect;
```

cds

We can also apply named aspects as **includes** in an inheritance-like syntax:

```
entity Books : NamedAspect { ... }
```

cds

↳ Learn more about the usage of aspects in the *Aspect-oriented Modeling* section..

TIP

Consumers always see the merged effective models, with the separation into aspects fully transparent to them.

Authorization

CAP supports out-of-the-box authorization by annotating services and entities with `@requires` and `@restrict` annotations like that:

```
entity Books @restrict: [  
    { grant: 'READ', to: 'authenticated-user' },  
    { grant: 'CREATE', to: 'content-maintainer' },  
    { grant: 'UPDATE', to: 'content-maintainer' },  
    { grant: 'DELETE', to: 'admin' },  
]) {  
    ...  
}
```

cds

To avoid polluting our core domain model with the generic aspect of authorization, we can use aspects to separate concerns, putting the authorization annotations into a

separate file, maintained by security experts like so:

```
// core domain model in schema.cds                                         cds
entity Books { ... }
entity Authors { ... }

// authorization model
using { Books, Authors } from './schema.cds';

annotate Books with @restrict: [
  { grant: 'READ',   to: 'authenticated-user' },
  { grant: 'CREATE', to: 'content-maintainer' },
  { grant: 'UPDATE', to: 'content-maintainer' },
  { grant: 'DELETE', to: 'admin' },
];

annotate Authors with @restrict: [
  ...
];
```

Fiori Annotations

Similarly to authorization annotations we would frequently add annotations which are related to UIs, starting with `@title` annotations used for field or column labels in UIs, or specific Fiori annotations in `@UI`, `@Common`, etc. vocabularies.

Also here we strongly recommend to keep the core domain models clean of that, but put such annotation into respective frontend models:

```
// core domain model in db/schema.cds                                         cds
entity Books : cuid { ... }
entity Authors : cuid { ... }

// common annotations in app/common.cds
using { sap.capire.bookshop as my } from '../db/schema';

annotate my.Books with {
  ID      @title: '{i18n>ID}';
  title  @title: '{i18n>Title}';
  genre   @title: '{i18n>Genre}'    @Common: { Text: genre.name, TextArrangement:
```

```

author @title: '{i18n>Author}'  @Common: { Text: author.name, TextArrangement: '#TextOnly' };
price  @title: '{i18n>Price}'    @Measures.ISOCurrency : currency_code;
descr  @title: '{i18n>Description}' @UI.MultiLineText;
}

// Specific UI Annotations for Fiori Object & List Pages
using { sap.capire.bookshop as my } from '../db/schema';

annotate my.Books with @(
  Common.SemanticKey : [ID],
  UI: {
    Identification : [{ Value: title }],
    SelectionFields : [ ID, author_ID, price, currency_code ],
    LineItem       : [
      { Value: ID, Label: '{i18n>Title}' },
      { Value: author.ID, Label: '{i18n>Author}' },
      { Value: genre.name },
      { Value: stock },
      { Value: price },
      { Value: currency.symbol },
    ]
  }
) {
  ID @Common: {
    SemanticObject : 'Books',
    Text: title, TextArrangement : #TextOnly
  };
  author @ValueList.entity: 'Authors';
}
);

```

Localized Data

Business applications frequently need localized data, for example to display books titles and descriptions in the user's preferred language. With CDS we simply use the `localized` qualifier to tag respective text fields in your as follows.

Do:

```
entity Books { ...  
    title : localized String;  
    descr : localized String;  
}
```

cds

Don't:

In contrast to that, this is what you would have to do without CAP's `localized` support:

```
entity Books {  
    key ID : UUID;  
    title : String;  
    descr : String;  
    texts : Composition of many Books.texts on texts.book = $self;  
    ...  
}  
  
entity Books.texts {  
    key locale : Locale;  
    key ID : UUID;  
    title : String;  
    descr : String;  
}
```

cds

Essentially, this is also what CAP generates behind the scenes, plus many more things to ease working with localized data and serving it out of the box.

TIP

By generating `.texts` entities and associations behind the scenes, CAP's **out-of-the-box support** for `localized` data avoids polluting your models with doubled numbers of entities, and detrimental effects on comprehensibility.

↳ Learn more in the [Localized Data guide](#).

Managed Data

Annotation: `@cds.on.insert`

Annotation: `@cds.on.update`

Use the annotations `@cds.on.insert` and `@cds.on.update` to signify elements to be auto-filled by the generic handlers upon insert and update. For example, you could add fields to track who created and updated data records and when:

```
entity Foo { //...
    createdAt : Timestamp @cds.on.insert: $now;
    createdBy : User      @cds.on.insert: $user;
    modifiedAt : Timestamp @cds.on.insert: $now  @cds.on.update: $now;
    modifiedBy : User      @cds.on.insert: $user @cds.on.update: $user;
}
```

↳ Learn more about pseudo variables `$now` and `$user` below.

These **rules** apply:

- Data *cannot* be filled in from external clients → payloads are cleansed
 - Data *can* be filled in from custom handlers or from `.csv` files
- Note the differences to [defaults](#)...

In Essence:

Managed data fields are filled in automatically and are write-protected for external clients.

Limitations

In case of `UPSERT` operations, the handlers for `@cds.on.update` are executed, but not the ones for `@cds.on.insert`.

Domain Modeling > Managed Data

Aspect *managed*

You can also use the pre-defined aspect `managed` from `@sap/cds/common` to get the very same as by the definition above:

```
using { managed } from '@sap/cds/common';
entity Foo : managed { /*...*/ }
```

↳ Learn more about `@sap/cds/common`.

With this we keep our core domain model clean and comprehensible.

Pseudo Variables

The pseudo variables used in the annotations above are resolved as follows:

- `$now` is replaced by the current server time (in UTC)
- `$user` is the current user's ID as obtained from the authentication middleware
- `$user.<attr>` is replaced by the value of the respective attribute of the current user
- `$uuid` is replaced by a version 4 UUID

↳ Learn more about *Authentication in Node.js*.

↳ Learn more about *Authentication in Java*.

[Edit this page](#)

Last updated: 17/03/2025, 12:28

Previous page
[Cookbook](#)

Next page
[Providing Services](#)

Was this page helpful?

