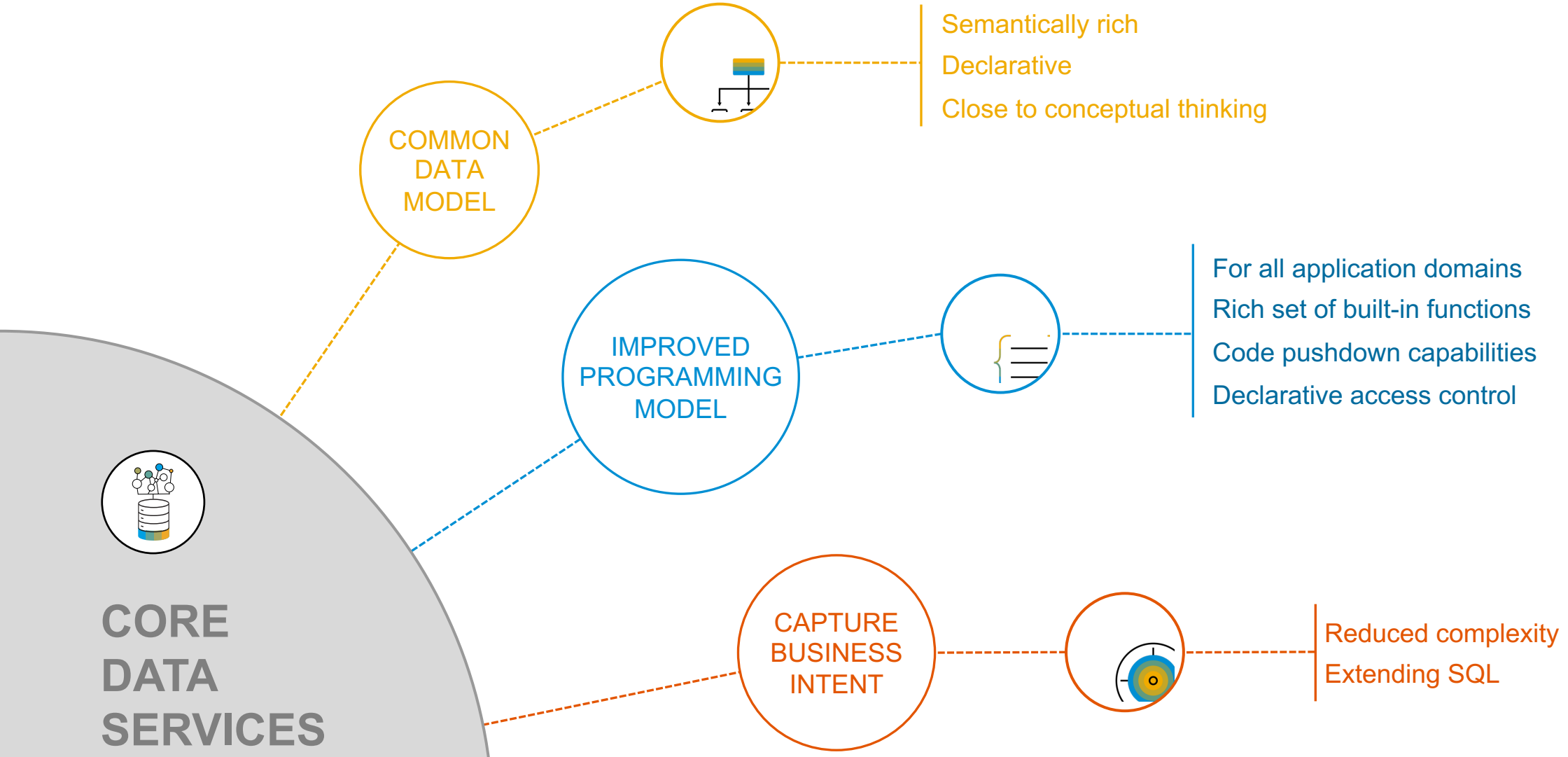
SAP Fiori Development Enablement

Core Data Services (CDS) Overview

Leonardo Britz, SAP

Q2 2024

Next generation data modeling and access



Definition Language (CDL) - Entities

```
entity Books : managed {
  key ID : Integer;
  title : localized String;
  authors : Association to many Authors;
  edition : String;
  stock  : Integer;
  price  : Decimal(9,2); }
```

```
entity Authors : managed {
    key ID : UUID;
    name : String;
    biography : String;
    books : Association to many Books; }
```

Definition Language (CDL) – Custom Structured Types

You can declare and use custom struct types as follows:

```
type Amount {
  value : Decimal(10,3);
  currency : Currency;
}

entity Books {
  price : Amount;
}
```

Elements can also be specified with anonymous inline struct types.

```
entity Books {
  price : {
    value : Decimal(10,3);
    currency : Currency;
  };
}
```

Definition Language (CDL) - Views

Use `as select from` or `as projection on` to derive new entities from existing ones by projections,

→ very much like views in SQL

→The entity signature is inferred from the projection.

Definition Language (CDL) - Views: The as select from Variant

Use the *as select from:* use **all possible features** supported by underlying relational database

CQL queries

```
entity Foo1 as SELECT from Bar; //> implicit {*}

entity Foo2 as SELECT from Employees { * };

entity Foo3 as SELECT from Employees LEFT JOIN Bar on Employees.ID=Bar.ID {
  foo, bar as car, sum(boo) as moo
} where exists (
  SELECT 1 as anyXY from SomeOtherEntity soe where soe.x = y
)
group by foo, bar
order by moo asc;
```

Definition Language (CDL) - Views: The as projection on Variant

Use the *as projection on:* you don't use the full power of SQL in your query.

→ e.g. allows us to serve such an entity from external OData services.

```
entity Authors as projection on db.Authors;
```

Currently the restrictions of *as projection on* compared to *as select from* are:

- no explicit, manual *JOINs*
- no explicit, manual *UNIONs*
- no sub selects in from clauses

Definition Language (CDL) - Associations

Associations capture relationships between entities.

- Unmanaged Associations
- Managed Associations
- To-many Associations
- Many-to-many Associations

Definition Language (CDL) - Unmanaged & Managed Associations

Unmanaged Associations

```
entity Employees {
  address : Association to Addresses on
address.ID = address_ID;
  address_ID : Integer;  //> foreign key
}

entity Addresses {
  key ID : Integer;
}
```

Managed Associations

```
entity Employees {
  address : Association to Addresses;
}
```

- For to-one associations
- *address_ID* added automatically upon activation to a SQL database

Definition Language (CDL) - Managed To-Many Associations

Simply add the keyword *many* to indicate a *0..** cardinality. Express and check all additional restrictions about cardinality, such as min 1 or max 2, as constraints, for example, using *not null*.

In order to manage to-many associations by generic providers later on, we usually need to specify a reverse to-one association on the target side using SQL-like on condition.
Use the canonic form of *on <target>.assoc = $self* as in our definition of *Authors*:

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

Definition Language (CDL) - To-many & Many-to-many Associations

To-many Associations

→ specify an on condition

→ *<assoc>.<backlink> = $self*

```
entity Employees {
  key ID : Integer;
  addresses : Association to many Addresses
    on addresses.owner = $self;
}

entity Addresses {
  owner : Association to Employees;
  //> the backlink
}
```

→ Backlink can be any managed to-one association

Many-to-many Associations

→ two one-to-many associations using a link entity to connect both.

```
entity Employees {
  key ID : Integer;
  addresses : Association to many Addresses
    on addresses.emp = $self;
}

entity Emp2Addr {
  key emp : Association to Employees;
  key adr : Association to Addresses;
}
```

Definition Language (CDL) - Compositions

Compositions constitute document structures through ‘contained-in’ relationships

```
entity Orders : cuid, managed {
  OrderNo  : String @title:'Order Number'; //> readable key
  Items    : Composition of many OrderItems on Items.parent = $self;
  currency : Currency;
}

entity OrderItems : cuid {
  parent    : Association to Orders;
  book      : Association to Books;
  amount    : Integer;
  netAmount : Decimal(9,2);
}
```

Definition Language (CDL) – Associations vs Compositions

In general, go for compositions in the following cases:

- →Parent and child share a life-cycle.
- →You can semantically establish a clear parent-child-hierarchy.
- →You never expose a child entity on its own.
- →The parent of an item never changes.
- →You want to keep entities together transactionally.

In general, go for associations in the following cases:

- →You can’t semantically establish a clear parent-child-hierarchy:
- →You expose a document entities fully on their own.
- →Relationships are likely to change over time.
- →Individual entities should have individual life-cycles.

Definition Language (CDL) - Annotation Syntax

- Prefixed with an `@` character
- Can be placed before a definition, after the defined name or at the end of simple definitions.

```
@before entity Foo @inner {
  @before simpleElement @inner : String @after;
  @before structElement @inner { /* elements */ }
}
```

Multiple annotations: separated by whitespaces or enclosed in `@(...)` and separated by comma

```
@my.annotation:foo
@another.one: 4711
entity Foo { /* elements */ }
```

```
entity Foo @(
  my.annotation: foo,
  another.one: 4711
) { /* elements */ }
```

Definition Language (CDL) - Annotation Propagation

Annotations are inherited from types and base types to derived types, entities, and elements.

The rules are:

1. Entity-level properties and annotations are inherited from the **primary** underlying source entity — here *Books*.

2. Each element that can **unambiguously** be traced back to a single source element, inherits that element's properties.

3. An explicit **cast** in the select clause cuts off the inheritance, for example, as for *genre* in our previous example.

```
using Books from './bookshop-model';
entity BooksList as SELECT from Books {
  ID,
  genre : Genre,
  title,
  author.name as author
};
```

- *BooksList* would inherit annotations from *Books*
- *BooksList.ID* would inherit from *Books.ID*
- *BooksList.author* would inherit from *Books.author.name*
- *BooksList.genre* would inherit from type *Genre*

Definition Language (CDL) - Service Definition

Service interfaces = collections of exposed entities

```
service CatalogService {
  entity Books as projection on db.Books;
  entity Authors as projection on db.Authors;
  entity Orders as projection on db.Orders { *, book.title, book.author.name as author }
  where createdBy = $user;
  action cancel(order:UUID);
}
```

- Endpoint: "./Catalog/Books/
- Overwrite the path, you can add the *@path* annotation as follows:

```
@path: 'myCustomServicePath'
service SomeService { ... }
```

Definition Language (CDL) - Extending Services

Extend **services** with additional entities and actions

```
extend service CatalogService with {
  entity Foo {};
  function getRatings() returns Integer;
}
```

Similarly, you can extend **entities** with additional (bound)actions as you would add new elements:

```
extend entity CatalogService.Products with actions {
  function getRatings() returns Integer;
}
```

Domain Modeling

What is Domain Modeling

Keep it simple, Stupid!

To reach these goals of domain focus, but also for the sake of simplicity and hence quality, robustness and consumeability, it is of utter importance to keep your models:

Clean:

→ no technical details → use Aspects!

→ separating concerns (Fiori Markup ; Authorization ; Persistence ; etc…) in same or different files or projects.

Concise: be on point, use short names, simple flat models, etc.

Comprehensible: domain modeling as a service to others!

Conceptual Modeling

Apply classical conceptual modeling methods to find your domain models

“We want to create an online service for our library allowing users to browse Books.

In addition, they should be able to:

- *browse Authors,*
- *and navigate from Authors to Books and vice versa.”*

```
namespace our.library;

entity Books {
  key ID : UUID;
  title  : String;
  descr  : String;
  author : Association to Authors;
}
entity Authors {
  key ID : UUID;
  name   : String;
  books  : Association to many Books on books.author=$self;
  birth  : Date;
  death  : Date;
}
```

Best practices: Naming Conventions

Naming Conventions

- Start ***entity*** and ***type*** names with ***uppercase letters*** – for example, ***Books***
- Start ***elements*** with a ***lowercase letter*** – for example, ***title***
- Use ***plural*** form for ***entities*** – for example, ***Authors***
- Use ***singular*** form for ***types*** – for example, ***Genre***

```
entity Books {
  key ID : UUID;
  title : String;
  genre : Genre;
  author : Association to Authors;
}
type Genre : String enum {
  Mystery; Fiction; Drama;
}
```

Prefer Concise Names

- *author.name* instead of *Author.authorName*
- *address* instead of *addressInformation*
- use *ID* for technical primary keys

Best practices: Avoid Dogmatic Separation of Entity Types

Do not fully separate entity types from entity sets

Do

```
entity Books {
  key ID : UUID;
  author : Association to Authors;
  title : String;
  descr : String;
}
```

Don’t

```
type Book {
  title : String;
  descr : String;
}
entity Books : Book {
  key ID : UUID;
  author : Association to Authors;
}
```

Best practices: Avoid over-normalization

Avoid Normalization

Do

```
entity Contacts {
  key ID : UUID;
  name : String;
  emails : array of {
    kind : String;
    address : String;
    primary : Boolean;
  };
  phones : array of {...}
}
```

Don’t

```
entity Contacts {
  key ID : UUID;
  name : String;
  emails : Composition of many EmailAddresses on emails.contact=$self;
  phones : Composition of many PhoneNumbers on phones.contact=$self;
}
entity EmailAddresses {
  contact : Association to Contacts;
  key ID  : UUID;
  kind    : String;
  address : String;
  primary : Boolean;
}
entity PhoneNumbers {...}
```

Best practices: Use Managed Associations

Keep your domain models to a conceptual level by using managed associations

Do

```
entity Books {
  key ID : UUID;
  title : String;
  author : Association to Authors;
}
entity Authors {
  key ID : UUID;
  name : String;
  books : Association to many Books on books.
author = $self;
}
```

Don’t

```
entity Books {
  key ID : UUID;
  title : String;
  author_ID : UUID;
  author : Association to Authors on author.ID = author_ID;
}
entity Authors {
  key ID : UUID;
  name : String;
  books : Association to many Books on books.author_ID = ID;
}
```

Thank you.

Contact information:

Leonardo Britz
Senior UX Consultant
leonardo.britz@sap.com

Follow all of SAP

www.sap.com/contactsap

THE BEST RUN SAP®