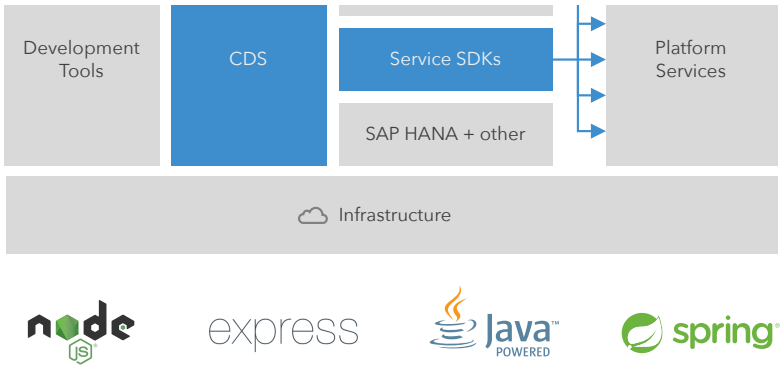


On top of open source technologies, CAP mainly adds:

- **Core Data Services (CDS)** as our universal modeling language for both domain models and service definitions.
- **Service SDKs and runtimes** for Node.js and Java, offering libraries to implement and consume services as well as generic provider implementations serving many requests automatically.

Open *and* Opinionated

That might sound like a contradiction, but isn't: While CAP certainly gives *opinionated* guidance, we do so without sacrificing openness and flexibility. At the end of the day, **you stay in control** of which tools or technologies to choose, or which architecture patterns to follow as depicted in the table below.

CAP is *Opinionated* in...	CAP is *Open* as...
Higher-level concepts and APIs abstracting from and avoiding lock-ins to low-level platform features and protocols	All abstractions follow a glass-box pattern that allows unrestricted access to lower-level things, if required
Best Practices served out of the box with generic solutions for many recurring tasks	You can always handle things your way in custom handlers, decide whether to adopt CQRS or Event Sourcing, for example ... while CAP simply tries to get the tedious tasks out of your way.
Out-of-the-box support for SAP Fiori and SAP HANA	You can also choose other UI technologies, like Vue.js, or databases, by providing new database integrations.
Dedicated tools support provided in SAP Business Application Studio or Visual Studio Code.	CAP doesn't depend on those tools. Everything in CAP can be done using the `@sap/cds-dk` CLI and any editor or IDE of your choice.

Agnostic Design

challenge when having to hardwire too many things to today's technologies, which might soon become obsolete. **CAP avoids such lock-ins** through **higher-level concepts and APIs**, which abstract low-level platform features and protocols to a large extent. In particular, this applies to things like:

- Platform-specific deployment approaches and techniques
- Platform-specific identity providers and authentication strategies
- On-/Offboarding of tenants in SaaS solutions and tenant isolation
- Synchronous protocols like *REST*, *OData*, or *GraphQL*
- Asynchronous channels and brokers like *SAP Event Mesh*, *MQ*, or *Kafka*
- Different database technologies including *SQL* and *NoSQL*

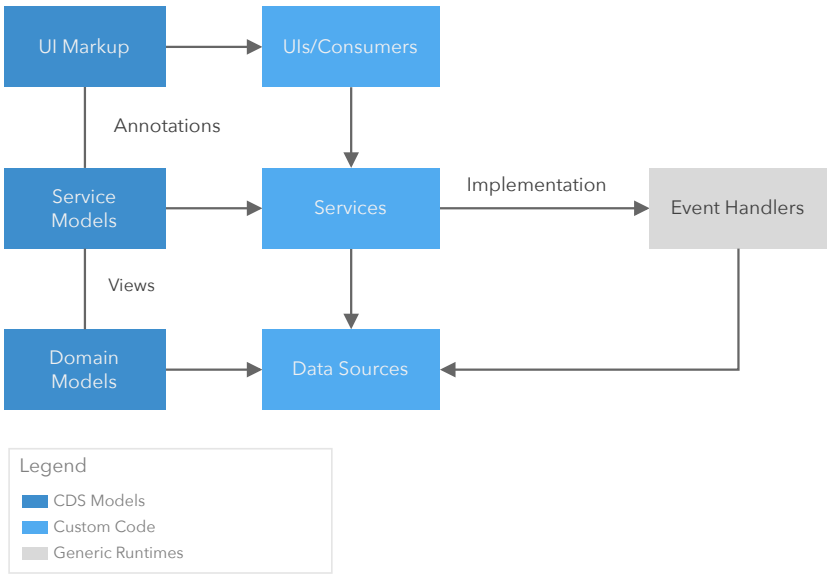
These abstractions allow us to quickly adapt to new emerging technologies or platforms, without affecting application code, thus **safeguarding your investments**.

Focus on Domain

CAP places **primary focus on domain**, by capturing *domain knowledge* and *intent* instead of imperative coding – that means, *What, not How* – thereby promoting:

- Close collaboration of *developers* and *domain experts* in domain modeling.
- *Out-of-the-box* implementations for *best practices* and recurring tasks.
- *Platform-agnostic* approach to *avoid lock-ins*, hence *protecting investments*.

The figure below illustrates the prevalent use of CDS models (in the left column), which fuel generic runtimes, like the CAP service runtimes or databases.



Anatomy of a Typical Application

Core Data Services (CDS)

CDS is our universal modeling language to capture static, as well as behavioral aspects of problem domains in **conceptual**, **concise**, and **comprehensible** ways, and hence serves as the very backbone of CAP.

Domain Models in CDS

```cds
entity Books : cuid {
  title  : localized String;
  author : Association to Authors;
}

entity Orders : cuid, managed {
  descr : String;
  Items : Composition of many {
    book : Association to Books;
    quantity : Integer;
  }
}
```

Domain Models capture static aspects of problem domains as well-known *entity-relationship models*.

Associations capture relationships. *Compositions* extend that to easily model **document structures**.

Annotations allow enriching models with additional metadata, such as for UIs, Validations, Input Validation or Authorization.

CDS Aspects & Mixins

```cds
// Separation of Concerns
extend Books with @restrict: [
  { grant:'WRITE', to:'admin' }
];

// Verticalization
extend Books with {
  ISBN : String
};

// Customization
extend Orders with {
  customer_specific : String
};
```

Aspects allow to flexibly **extend** models in same or separate modules, packages, or projects; at design time or dynamically at runtime.

This greatly promotes adaptability in *verticalization* and *customization* scenarios, especially in SaaS solutions.

Moreover, that fosters **separation of concerns**, for example to keep domain models clean and comprehensible, by factoring out technical concerns.

Querying & Views

All data access in CAP is through dynamic queries, which allows clients to request the exact information they really need. These powerful intrinsic querying capabilities are key enablers for **serving requests automatically**.

> Note: The querying-based approach to process data is in strong contrast to Object-Relational Mapping (→ see also *Related Concepts: CAP != ORM*)

Core Query Language (CQL)

CQL is CDS's advanced query language. It enhances standard SQL with elements to easily query deeply nested **object graphs** and **document structures**. For example, here's a query in CQL:

```sql
SELECT ID, addresses.country.name from Employees
```

... and the same in plain SQL:

```
LEFT JOIN Addresses ON Addresses.emp_ID=Employees.ID
LEFT JOIN Countries AS Countries ON Addresses.country_ID = Countries.ID
```

Queries as first-order Objects (CQN)

```js
// In JavaScript code
orders = await SELECT.from (Orders, o=>
  o.ID, o.descr, o.Items (oi=>{
    oi.book.title, oi.quantity
  })
})
```

Queries are first-order objects – using CQN as a plain object notation – sent to **local** services directly, to **remote** services through protocols like *OData* or *GraphQL*, or to **database** services, which translate them to native database queries for optimized execution with **late materialization**.

```http
// Via OData
GET .../Orders?$select=ID,descr
$expand=Items(
  $select=book/title,quantity
)
```

Projections at Design Time

```cds
// Projections in CDS
service OrdersService {
  define entity OrderDetails
  as select from Orders {
     ID, descr, Items
  }
}
```

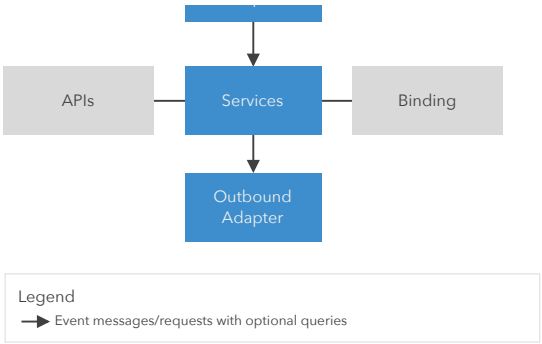
We also use CQL in CDS to declare *de-normalized views* on the underlying domain model, such as in tailored service APIs.

Services & Events

All behavioral aspects in CAP are based on ubiquitous notions of Services and Events, as expressed in this manifest:

1. *All active things are Services* – local ones, remote ones, as well as databases
2. *Services are declared in CDS* – reflected and used in generic service providers
3. *Services provide uniform APIs* – consumed by other services or frontends
4. *Services react on Events* – covering *synchronous* and *asynchronous* APIs
5. *Services consume other Services* – in event handler implementations
6. *All data is passive* – that is, without its own behavior, adhering to *REST*

Services in CAP are **stateless** and with a **minimal footprint**, which allows you to modularize solutions into single-purposed (nano) services or functions-as-a-service.



Hexagonal Architecture à la CAP

Service Definitions in CDS

```cds
// Service Definition in CDS
service OrdersService {
  entity Orders as projection on my.Or
  action cancelOrder (ID:Orders.ID);
  event orderCanceled : { ID:Orders.ID
}
```

Services are declared in CDS models, used to serve requests automatically. They embody the behavioral aspects of a domain in terms of exposed **entities**, **actions**, and **events**.

Uniform Consumption

```js
// Consumption in JavaScript
let srv = cds.connect.to('OrdersServic
let { Orders } = srv.entities
order = await SELECT.one.from (Orders)
  .where({ ID:4711 })
srv.cancelOrder (order.ID)
```

Every active thing in CAP is a service, including *local* services or *remote* ones – even *databases* are represented as services.

All services provide a **uniform** API for programmatic consumption. Thus, application code stays **agnostic** to underlying protocols.

```http
// Consumption via REST APIs
GET /orders/Orders/4711
POST /orders/cancelOrder/4711
```

> ***Late-cut μ services***
>
> This protocol-agnostic API allows mocking remote services, as well as late changes to service topologies, for example, co-locating services in a single process or deploying them to separate micro services later on.

Ubiquitous Events

```js
// Service Implementation
cds.service.impl (function(){
  this.on ('UPDATE','Orders', (req)=>{
  this.on ('cancelOrder', (req)=>{})
})
```

Everything in CAP happens in response to events. CAP features a ubiquitous notion of events, which represent both, *requests* coming in through **synchronous** APIs, as well as **asynchronous** *event messages*, thus blurring the line between both worlds.

↳ *See also the Features Overview*

Grow As You Go

Following the principle of **convention over configuration**, there's no need to set up things upfront. CAP allows you to **jumpstart** projects within seconds and have a team starting development right away, using generic providers, on top of a lightweight in-memory database → see *Getting Started in a Nutshell*.

CAP also offers **mocks for many platform features**, which allow **fast dev-test-run cycles** with minimal development environment complexity – aka *Airplane Mode*. Similarly, CAP greatly facilitates **integration scenarios** by simply importing an API from, say, an SAP S/4 backend or from SAP API Hub and running mocks for this locally.

Over time, you **add things gradually**, only when they're needed. For example, you can move ahead to running your apps in close-to-productive setups for integration tests and delivery, without any change in models or code. → see *Grow-as-you-Go*.

Finally, projects are encouraged to **parallelize workloads**. For example, following a **contracts-first** approach, a service definition is all that is required to automatically run a full-fledged REST or OData service. So, projects could spawn two teams in parallel: one working on the frontend, while the other one works on the backend part. A third one could start setting up CI/CD and delivery in parallel.

Features Overview

↳ *See an overview of all features.*

Related Concepts

The sections below provide additional information about CAP in the context of, and in comparison to, these related concepts:

CAP == *Hexagonal Architecture*

> CAP's service architecture is designed with the same ideas in mind as the blueprints of *Hexagonal Architecture* (or Onion Architecture, or Clean Architecture). With that, CAP facilitates projects that choose to apply the principles and designs of those architecture patterns.

CAP promotes Domain-Driven Design

problem domain, providing CDS as a powerful language to capture domain models and thereby facilitating close collaboration between domain experts and developers. CAP's core concepts fit well to the DDD counterparts of *Entities*, *Value Objects*, *Services*, and *Events*.

In contrast to DDD however, CAP prefers a strong distinction of active services and passive data; for example, there's no such thing as instance methods of entities in CAP. CAP also stays at a more axiomatic level of key concepts: the DDD concepts of *Repositories*, *Aggregates*, and *Factories* intentionally don't have first-class counterparts in CAP, but can be realized using CAP's core concepts.

CAP promotes CQRS

Similar to CQRS, CAP strongly recommends separating write APIs from read APIs by defining separate, single-purposed services. CDS's reflexive view building eases the task of declaring derived APIs exposing use case-specific de-normalized views on underlying domain models. Service actions in CAP can be used to represent pure commands. There's no restriction to 'verb-only' dogmas in CAP though, as CAP focuses on business applications, which are mostly data-oriented by nature, hence frequently 'entity/noun-centric'.

CAP and Event Sourcing

CAP can be combined with event sourcing patterns, that is, by tracking events in an event store, like Apache Kafka, instead of maintaining a snapshot of data in a relational or NoSQL database. Currently we don't provide out-of-the-box integration to such event sources (we may do so in near future), however this can be easily done in projects by respective service implementations using CAP's built-in capabilities to send and receive messages.

CAP supports SQL

CDS borrows reflexive view building from SQL to declare derived models and APIs as projections/transformation of underlying models, such as domain models. CQL is based on SQL DML to allow direct mapping to SQL databases. However, it extends SQL with Associations, Path Expressions, and Nested Projections to overcome the need to deal with JOINs. Instead, these extensions allow working with data in a structured document-oriented way.

CAP supports NoSQL

The previously mentioned extensions in CQL feature the modeling of nested document structures as well as view building and querying using navigation instead of cross products, joins, and unions. This actually brings CDS close to the concepts of NoSQL databases, with the data models playing the role of schemas for validation. Although CAP currently doesn't provide out-of-the-box support for concrete NoSQL databases, it's easy to do so in project-specific solutions.

CAP and the Relational Model

While CDS extends SQL and the relational model by means to describe, read, and write deeply nested document structures (see relationship to SQL below), it stays compatible to the principles of relational models with a specified mapping to relational databases.

CAP == Entity-Relationship Modeling

data structures of a given domain. Relationships are modeled as Associations and Compositions.

CAP == Aspect-Oriented Programming

Aspects in CDS are borrowed from AOP, especially *Mixins*. With that, CAP greatly facilitates separation of concerns by "...factoring out technical concerns (such as security, transaction management, logging) from a domain model, and as such makes it easier to design and implement domain models that focus purely on the business logic." (source: Wikipedia)

CAP == Functional Programming

Similar to Functional Programming, CAP promotes a declarative programming paradigm, which declaratively captures domain knowledge and intent (what), instead of writing imperative boilerplate code (how), as much as possible. This helps to automate many recurring tasks using best practices. Also similar to functional programming, and in contrast to object-oriented and object-relational approaches, CAP promotes the distinction of passive data (~immutable data) and active, stateless services (~pure functions).

In addition, CAP features *queries as first-class and higher-order objects*, allowing us to apply late evaluation and materialization, similar to first-class and higher-order functions in Functional Programming.

CAP != Object-Relational Mapping

CAP and CDS aren't *Object-Relational Mapping* (ORM). Instead, **we prefer querying** using CQL to read and write data, which allows declaratively expressing which data you're interested in by means of projection and selection instead of loading object graphs automatically. Result sets are pure REST data, that are snapshot data representations. One reason for this is the assumption that the lifetime of object cache entries (which are essential for ORMs to perform) is frequently in the range of milliseconds for *REST* services.

CAP != Business Objects

Business Object Patterns promote the notion of active objects, which provide instance methods to modify their internal state. In contrast to that, CAP promotes a strict separation of passive data, read and exchanged in RESTful ways, and pure, stateless services. (→ see also the relationship to Functional Programming above).

About Capire

"*Capire*" (Italian for 'understand') is the name of our CAP documentation you're looking at right now. It's organized as follows:

- *About CAP* – a brief introduction and overview of key concepts
- *Getting Started* – a few guides to get you started quickly
- *Cookbook* – task-oriented guides from an app developer's point of view
- *Tools* – choose your preferred tools
- *Reference docs* → for *CDS*, *Node.js*, *Java*
- *Releases* – information about what is new and what has changed

TL;DR – too long; didn't read – is a common social phenomenon these days. Therefore, capire is rather meagre with text and greased with code. You have to read between the lines, sorry.

Glossary

- **CAP** – Colloquial shorthand for "SAP Cloud Application Programming Model". Not an official product name, though.
- **CDS** – Core Data and Services : a family of languages, tools and libraries to declare, process and consume conceptual, semantically enriched data models.
- **CDL** – CDS Definition Language : a human-readable representation of CDS models.
- **CSN** – Core Schema Notation : a plain (JavaScript) object-based representation of CDS models. (pronounce like *'Ceason'*)
- **CQL** – CDS Query Language : an extension of SQL leveraging Associations defined in CDS models. (pronounce like *'Cequel'*)
- **CQN** – Core Query Notation : a plain (JavaScript) object-based representation to capture queries. (also contained in CSN)
- **CXN** – Core Expression Notation : a plain (JavaScript) object-based representation to capture expressions. (also contained in CQN)

Badges

- since @sap/... 1.2.3 – The marked feature is only available with the given version or higher.
- alpha – Alpha features are experimental. They may never be generally available. If released subsequently, the APIs and behavior might change.
- beta – Beta features are planned to be generally available in subsequent releases, however, APIs and their behavior are not final and may change in the general release.
- concept – Concept features are ideas for potential future enhancements and an opportunity for you to give feedback. This is not a commitment to implement the feature though.