Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2024, 11(4):159-164

Research Article ISSN: 2394 - 658X

SAP CAPM Tools and Capabilities - Part 2

Deepak Kumar

Wilmington, USA, Deepak3830@gmail.com

ABSTRACT

The SAP Cloud Application Programming Model (CAPM) is a comprehensive framework for building sophisticated, cloud-native applications within the SAP ecosystem. Grounded in domain-driven design (DDD), CAP empowers developers to model complex business domains using Core Data Services (CDS). CDS' unified, declarative syntax streamlines development by defining entities, relationships, and business logic in one place. This approach aligns applications closely with business needs, enabling rapid iteration and adaptation. CAP fosters modular, independently deployable services that seamlessly interact with each other and external systems through robust API capabilities. Leveraging Node.js for server-side logic, CAP applications excel in scalability, responsiveness, and handling asynchronous operations.

CAP combines proven open-source and SAP technologies. Its infrastructure supports Node.js and Java, offering developers flexibility. The Node.js SDK, built on Express, provides a rich ecosystem of libraries. CAP also supports Java for enterprises. SAP offers tools for both environments, including SAP Business Application Studio and Visual Studio Code. Core Data Services (CDS) is CAP's foundation, modeling both domain models and service definitions. CDS models can be deployed to databases like SAP HANA. CAP's service SDKs for Node.js and Java enable service implementation and access to SAP Business Technology Platform services like authentication and authorization.

CAP smoothly integrates with the SAP Cloud Platform and SAP S/4HANA enhancing data accessibility, security, and implementation. This collaboration maximizes investments and broadens functionalities throughout the SAP environment. Prioritizing features, for businesses, adaptability and expandability CAP enables companies to develop adaptable scalable applications that fulfill contemporary business needs and foster innovation moving forward.

Keywords: SAP CAP, SAP BTP, SAP Fiori, Node.js, Java, SAP S4 HANA

INTRODUCTION

SAP Business Technology Platform (BTP) is a cloud-based platform that allows users to develop integrate and enhance SAP applications. It provides Platform as a Service (PaaS) features such, as analytics, artificial intelligence/machine learning (AI/ML), and Internet of Things (IoT) facilitating development and growth opportunities.

SAP Fiori provides user interfaces, for SAP software. Its user-friendly and adaptable layout boosts efficiency, on devices by streamlining business operations.

Node.js, a JavaScript runtime built on Chrome's V8 engine, excels at scalable, real-time applications. Its non-blocking I/O and event-driven architecture optimize server-side programming.

Java is an object-oriented programming language recognized for its adaptability and dedicated user base. It drives business applications on platforms. Plays a crucial role, in numerous backend operations.

SAP HANA is an in-memory database and application platform that accelerates data processing and analytics. Offering real-time insights, predictive analytics, spatial processing, and machine learning, it's a core technology for modern enterprises.

Pre-requisite: To understand this paper thoroughly prerequisite is the SAP CAPM Tools and Capabilities - Part 1

DEFINING SERVICES AND APIs

When it comes to Cloud Application Programming (CAP) models, services play a role as they encapsulate business logic, data operations and integration points. Making these services accessible involves ensuring they can be reached by parts of the application or external users through defined APIs. Here's a guide, on exposing services in CAP applications; Start by defining each service within your CAP application. These services can cover a range of functions like data handling, business processes, or external connections. Create consistent APIs for each service detailing endpoints, HTTP methods, request/response formats, and security measures. Develop the service logic in line with the API requirements. Utilize CAP frameworks to simplify tasks such as data modeling, validation and integration. Integrate the registered services into the CAP application framework by configuring service endpoints, security protocols, and access permissions. Implement security protocols like authentication (OAuth, JWT) and authorization (role-based access control) to safeguard exposed services, against entry.

In applications developed using CAP service definitions serve as descriptions of the services encompassing data structures, operations, and APIs. Annotations play a role by providing information that enhances these definitions and guides the framework on how to manage and expose services effectively. When it comes to service definitions and annotations, in CAP applications key elements involve defining data models using CAP data modeling capabilities such as CDS (Core Data Services). This includes specifying entities, relationships and attributes that represent business entities or data objects. Service operations are implemented using scripting or programming languages supported by CAP like JavaScript or TypeScript. These operations encompass CRUD functionalities (Create, Read, Update, Delete) along with any custom business logic required. To further enrich service definitions annotations are utilized to offer context or instructions for behavior to the CAP framework. Examples of annotations include @cds.service which designates a module as a service provider, @cds.persistence for defining persistence options related to entities like database table mapping and @cds.deploy which specifies deployment settings such as endpoint paths and security configurations. It is essential to document APIs by outlining details such as endpoint URLs supported by HTTP methods, query parameters used along with request/response formats like JSON or XML, and procedures for handling errors. Additionally considering API versioning strategies can aid in managing changes while ensuring compatibility, with existing applications.

Consuming Services from Other Applications

CAP applications often need to consume services provided by other applications or external systems. Effective consumption involves integrating external APIs seamlessly into your CAP application ecosystem.

When setting up your CAP application make sure to identify and list the services and APIs it needs to use. Understand their web addresses how they need to be accessed and the way they organize data. Select appropriate ways to connect with these services; Communicate, with links through HTTP requests (like GET, POST, PUT, DELETE) Utilize SOAP-based web services for structured data exchange or Connect with message brokers (such as RabbitMQ or Kafka) for asynchronous communication styles. Set up the security measures like API keys or OAuth tokens for authentication. Make sure you have the permissions in place to access protected data following the rules of each service. Have plans in place for dealing with errors (such as retries or circuit breakers) to handle failures and keep your application running when interacting with external services. Convert data formats, between your CAP application and external services as needed (like changing from JSON to XML).

IMPLEMENTING BUSINESS LOGIC WITH CAPM

Developing services, with a Cloud Application Programming Model (CAP) entails using frameworks and tools that are specifically designed to make the creation, deployment, and maintenance of applications and services easier. When it comes to incorporating business logic into cloud applications using Node.js it's all about making the most of server-side programming, event-driven architecture, and workflow management within a Cloud Application Programming Model (CAP). This section offers a walkthrough from principles to more advanced techniques enabling developers to construct cloud-native applications that are scalable, robust, and efficient. Cloud Application Programming Models (CAP) offer frameworks and approaches that streamline the process of developing, deploying, and managing native applications. These CAP frameworks typically prioritize scalability, adaptability and seamless integration, with cloud services.

Node.js

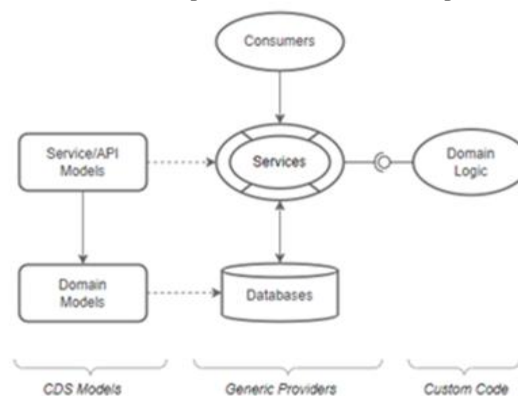
Imagine JavaScript, the language usually confined to your browser, breaking free to power the behind-the-scenes magic of websites. That's Node.js! Think of Node.js as a versatile tool that lets you build the backbone of web applications. It's incredibly fast because it uses Google Chrome's lightning-quick engine under the hood. Plus, it's designed to handle multiple tasks at once without slowing down, making it perfect for websites with lots of visitors.

Node.js has a huge community of developers sharing tips, tricks, and pre-built code snippets. It's like having a team of helpful neighbors ready to lend a hand. And the best part? You can use Node.js on Windows, Mac, or Linux – it works everywhere! So, whether you're creating a bustling online chat room, a high-speed web API, or the

foundation for a massive website, Node.js is your go-to sidekick. It's the unsung hero behind many of the websites you use every day.

HOW TO DEFINE AND IMPLEMENT SERVICES, LEVERAGING GENERIC IMPLEMENTATIONS

Think of a CAP application as a bustling city. Each business or building in the city is a service, offering specific products or services. These businesses interact with customers (or other businesses) through various channels, like walk-in visits or online orders. In the CAP world, these interactions are events. A customer walking into a store or placing an online order triggers a series of actions within the business. CAP treats all these interactions, whether in-person or digital, as equal events. Each business (or service) in CAP is constantly on the lookout for these events. When an event happens, the business responds accordingly. It's like a waiter in a restaurant who reacts when a customer arrives or a kitchen staff member who responds when an order is placed.



In its most basic form, a service definition simply declares the data entities and operations it serves. Simple service definitions like that are all we need to run full-fledged servers out of the box, served by CAP's generic runtimes, without any implementation coding required.

Services as Facades:

In contrast to the all-in-one definition, services usually expose views, aka projections, on underlying domain model entities. This way, services become facades to encapsulated domain data, exposing different aspects. The cds facade object provides access to all CAP Node.js APIs.

```
const cds = require('@sap/cds')

let csn = cds.compile(`entity Foo {}`)
```

To optimize performance and memory usage, CDS employs a modular structure. Submodules are loaded only when needed, reducing initial startup time. For clarity and efficiency, accessing classes and functions through a facade object is recommended. This approach simplifies interactions with complex underlying systems by providing a single, unified point of access. The Facade pattern, a well-established design principle, is extended in CDS through the concept of "services as facades." By encapsulating intricate backend systems, microservices, or APIs behind a simplified interface, CDS shields users from unnecessary complexities, promoting ease of use and development efficiency.

Facades streamline interactions with backend services by providing a unified access point. This simplifies development and reduces learning curves. By abstracting away complex details like communication protocols and data formats, facades make integration smoother and more consistent. Facades serve as versatile intermediaries, aggregating data and functionalities from multiple backend services into a cohesive interface. This consolidated view simplifies client interactions and reduces integration efforts. Offering a consistent API, facades insulate clients from the complexities of heterogeneous backend systems. This promotes easier maintenance and updates as facades can evolve independently. Additionally, they serve as a centralized point for implementing security measures and governance policies.

Key Terms to Remember

Denormalized Views: Instead of exposing access to underlying data in a 1:1 fashion, services frequently expose denormalized views, tailored to specific use cases.

Auto-Exposed Entities: Annotate entities with @cds.autoexpose to automatically include them in services containing entities with Association referencing to them.

Redirected Associations: When exposing related entities, associations are automatically redirected. This ensures that clients can navigate between projected entities as expected.

cds.version: Returns the version of the @sap/cds package from which the current instance of the cds facade module was loaded.
cds.home: Returns the pathname of the @sap/cds installation folder from which the current instance of the CDS facade module was loaded.
cds.root: Returns the project root that is used by all CAP runtime file access as the root directory. By default, this is process.cwd(), but can be set to a different root folder. It's guaranteed to be an absolute folder name.
cds.entities: Is a shortcut to cds.db.entities. Used as a function, you can specify a namespace.
cds.env: Provides access to the effective configuration of the current process, transparently from various sources, including the local package.json or .cdsrc.json, service bindings, and process environments.
cds.requires: is an overlay and convenience shortcut to cds.env.requires, with additional entries for services with names different from the service definition's name in cds models.
cds.services: A dictionary and cache of all instances of cds.Service constructed through cds.serve(), or connected to by cds.connect().
cds.model: The effective CDS model loaded during bootstrapping, which contains all service and entity definitions, including required services. Many framework operations use that as a default where models are required.
cds.app: The express.js Application object constructed during bootstrapping. Several framework operations use that to add express handlers or middlewares.
cds.db: A shortcut to cds.services.db, the primary database connected to during bootstrapping. Many framework operations use that to address and interact with the primary database.

UNDERSTANDING GENERIC PROVIDERS

CAP's Node.js and Java runtimes offer a robust set of pre-built functionalities to handle common tasks. These generic implementations automate routine operations like search, pagination, and input validation, streamlining development. This guide primarily explores these core features.
CRUD operations are handled automatically by CAP for CDS-modeled services. Read and write actions on the primary database are managed without requiring custom code.

1) GET /Entity/201 → reading single data entities
2) GET /Entity?... → reading data entity sets with advanced query options
3) POST //Entity {....} → creating new data entities
4) PUT/PATCH /Entity 201 {...} → updating data entities
5) DELETE /Entity /201 → deleting data entities

Deep Reads / Writes: CDS and the runtimes have advanced support for modelling and serving document-oriented data. The runtimes provide generic handlers for serving deeply nested document structures out of the box.
Deep INSERT: Create a parent entity along with child entities in a single operation. The Associations and Compositions are handled differently in (deep) inserts and updates: Compositions → runtime deeply creates or updates entries in target entities. Associations → runtime fills in foreign keys to existing target entries.
Deep UPDATE: Deep UPDATE of the deeply nested documents look very similar to deep INSERT. Depending on existing data, child entities will be created, updated, or deleted. Entries existing on the database, but not in the payload, are deleted. Entries existing on the database, and in the payload are updated. Entries not existing on the database are created. Omitted fields get reset to default values or null in case of PUT requests; they are left untouched for PATCH requests. Omitted compositions have no effect, whether during PATCH or during PUT. That is, to delete all children, the payload must specify null or [], respectively, for the to-one or to-many composition.
Deep DELETE: Deleting a root of a composition hierarchy results in a cascaded delete of all nested children.
Auto-Generated Keys: On CREATE operations, key elements of type UUID are filled in automatically. In addition, on deep inserts and upserts, respective foreign keys of newly created nested objects are filled in accordingly.
Searching Data: CAP runtimes provide out-of-the-box support for advanced search of a given text in all textual elements of an entity including nested entities along composition hierarchies.

Pagination & Sorting:
Implicit Pagination: To optimize performance, READ requests are limited to a maximum of 1,000 records by default. When more data exists, a link to fetch subsequent pages is provided in the response.
Reliable Pagination: To prevent data inconsistencies caused by modifications during pagination, CAP offers a reliable pagination mechanism based on the last row of a page. This feature is exclusive to OData V4 and has specific limitations:

- Ordering must be based on simple data types, not calculated values or expressions.
- All ordered fields should be included in the selected fields.
- Complex result set combinations are unsupported.

Actions & Functions

Beyond standard CRUD operations, CAP allows for custom operations tailored to specific business needs. These operations require custom implementations within event handlers.

The distinction between Actions and Functions, as well as bound and unbound operations, aligns with OData specifications. Actions modify server-side data, while Functions retrieve data. Bound operations implicitly receive the primary key of the associated entity, similar to object references in languages like Java or JavaScript. Unbound operations operate independently without entity context.

Event handlers for custom operations follow a similar structure to those for CRUD operations, replacing the CRUD operation name with the custom operation name. Method-style implementations using JavaScript methods within cds.Service subclasses are also supported for actions and functions.

SERVING MEDIA DATA

CAP provides out-of-the-box support for serving media and other binary data. You can use the following annotations in the service model to indicate that an element in an entity contains media data.

@Core.MediaType : Indicates that the element contains media data (directly or using a redirect). The value of this annotation is either a string with the contained MIME type, or is a path to the element that contains the MIME type.

@Core.IsMediaType : Indicates that the element contains a MIME type. The @Core.MediaType annotation of another element can reference this element.

@Core.IsURL @Core.MediaType : Indicates that the element contains a URL pointing to the media data (redirect scenario).

@Core.ContentDisposition.Filename : Indicates that the element is expected to be displayed as an attachment, that is downloaded and saved locally. The value of this annotation is a path to the element that contains the Filename.

@Core.ContentDisposition.Type : Can be used to instruct the browser to display the element inline, even if @Core.ContentDisposition.Filename is specified, by setting to inline. If omitted, the behavior is @Core.ContentDisposition.Type: 'attachment'.

Reading Media Resources: Read media data using GET requests of the form /Entity(<ID>)/mediaProperty. Read media data with @Core.ContentDisposition.Filename in the model.

Creating a Media Resource: As a first step, create an entity without media data using a POST request to the entity. After creating the entity, you can insert a media property using the PUT method. The MIME type is passed in the Content-Type header.

Updating Media Resources: The media data for an entity can be updated using the PUT method.

Deleting Media Resources: One option is to delete the complete entity, including all media data. Alternatively, you can delete a media data element individually.

Limitations: The usage of binary data in some advanced constructs like the $apply query option and /any() might be limited. On SQLite, binary strings are stored as plain strings, whereas a buffer is stored as binary data. As a result, if in a CDS query, a binary string is used to query data stored as binary, this wouldn't work. SQLite doesn't support streaming. That means, that LargeBinary fields are read as a whole (not in chunks) and stored in memory, which can impact performance. SAP HANA Database Client for Node.js (HDB) and SAP HANA Client for Node.js (@sap/hana-client) packages handle binary data differently. For example, HDB automatically converts binary strings into binary data, whereas SAP HANA Client doesn't. In the Node.js Runtime, all binary strings are converted into binary data according to SAP HANA property types. To disable this default behavior, you can set the environment variable cds.env.hana.base64_to_buffer to false.

KEY CONSIDERATIONS FOR SERVICE CONNECTION AND DEPLOYMENT

Destinations: Your Connection Blueprint

Destinations serve as configuration blueprints for connecting to external systems. Beyond basic URLs, they include essential details like authentication credentials. CAP leverages the SAP Cloud SDK's destination capabilities to streamline this process.

For services imported using cds import, destination configuration is managed within the package.json file. Specify the destination name and path prefix to connect to the service. For testing purposes, consider using profile-based destinations to isolate production settings.

While SAP BTP destinations are commonly used, you can also define destinations directly within your application. However, this approach has limitations in terms of supported properties and authentication methods compared to SAP BTP destinations.

For microservices sharing the same XSUAA instance, destination configuration can be simplified. By forwarding the authorization token, direct connections without explicit destinations are often possible.

To access external systems, your microservice might require bindings to the XSUAA, Destination, and potentially Connectivity services, depending on the system's location and authentication method.

Industry Standards

Single-Purposed Services: It strongly recommends designing your services for single-use cases. Services in CAP are cheap, so there is no need to save on them.

One Service Per Use Case: when entities are tailored for different use cases such as order and customer their access types are different in that case use a separate service.

Late-Cut Microservices: Compared to Microservices, CAP services are 'Nano'. You should design your application as a set of loosely coupled, single-purposed services, which can all be served embedded in a single-server process at first (that is, a monolith). Yet, given such loosely coupled services, and enabled by CAP's uniform way to define and consume services, you can decide later on to separate, deploy, and run your services as separate micro services, even without changing your models or code. This flexibility allows you to, again, focus on solving your domain problem first, and avoid the efforts and costs of premature microservice design and DevOps overhead, at least in the early phases of development.

CONCLUSION

SAP Cloud Application Programming Model (CAPM) empowers developers to build robust, scalable, and efficient cloud-native applications within the SAP ecosystem. By providing a comprehensive framework for defining services, implementing business logic, and integrating with external systems, CAP significantly accelerates development cycles and enhances application quality. This paper delved into core CAPM concepts, including service definition and API creation, leveraging generic providers for common tasks, and handling specific data types like media. We explored practical considerations for service connection and deployment, emphasizing industry best practices. Through effective utilization of CAPM's capabilities, organizations can streamline development processes, improve application performance, and deliver exceptional user experiences. By adhering to recommended practices and leveraging the power of CAPM, developers can unlock the full potential of cloud-native architectures and drive digital transformation initiatives.

Declarations

Ethics approval and consent to participate: Not Applicable

Availability of data and materials: Not Applicable

Competing interests: Not Applicable

Funding: Not Applicable

REFERENCES

[1]. “SAP Cloud Application Programming Model | SAP Community,” pages.community.sap.com. https://pages.community.sap.com/topics/cloud-application-programming

[2]. “Home | capire,” cap.cloud.sap. https://cap.cloud.sap/docs/

[3]. Daniel7, “Introducing the Cloud Application Programming Model (CAP),” SAP Community, Jun. 05, 2018. https://community.sap.com/t5/technology-blogs-by-sap/introducing-the-cloud-application-programming-model-cap/ba-p/13354172

[4]. “SAP Cloud Application Programming Model | SAP Community,” pages.community.sap.com. https://pages.community.sap.com/topics/cloud-application-programming

[5]. “SAP CAP: How Does It Help Enterprises in Agile Development?,” www.gemini-us.com, Nov. 09, 2023. https://www.gemini-us.com/sap/sap-cap-how-does-it-help-enterprises-in-agile-development#:~:text=Additional%20Advantages

[6]. kumarsanjeev, “Part#1. SAP CDS views Demystification,” SAP Community, Oct. 21, 2019. https://community.sap.com/t5/enterprise-resource-planning-blogs-by-members/part-1-sap-cds-views-demystification/ba-p/13399722

[7]. R. Glushach, “Domain-Driven Design (DDD): A Guide to Building Scalable, High-Performance Systems,” Medium, Oct. 07, 2023. https://romanglushach.medium.com/domain-driven-design-ddd-a-guide-to-building-scalable-high-performance-systems-5314a7fe053c

[8]. “SAP Help Portal,” help.sap.com. https://help.sap.com/docs/btp/sap-business-technology-platform/developing-business-applications-using-node-js

[9]. “SAP Help Portal,” help.sap.com. https://help.sap.com/docs/bas/sap-business-application-studio/what-is-sap-business-application-studio

[10]. “SAP HANA Cloud | SAP Community,” pages.community.sap.com. https://pages.community.sap.com/topics/hana/cloud