🔷 1. RAP in One Line

RAP = CDS + Behaviour + OData + Fiori-ready Business Logic

RAP lets you define **Business Objects (BOs)** in a structured, declarative way where:

- Data modelling happens in **CDS Views** (Entity Definitions).
- Business logic (create/update/delete/validation) is defined in **Behaviour Definitions (BDEF)**.
- Logic execution occurs in **Behaviour Implementations (BIMP)**.
- Exposure happens automatically via **OData V4 Services**.

🔷 2. What Are Managed and Unmanaged BOs?

Type	Definition	Who Handles Transaction Logic?	Who Handles Data Persistence?
Managed BO	The RAP framework **manages CRUD** (Create/Read/Update/Delete) operations automatically. You only write custom logic (validations, determinations, actions).	RAP Framework	RAP Framework
Unmanaged BO	You, the developer, must **manually implement CRUD** logic (like classic ABAP). RAP only provides the skeleton.	Developer	Developer

In short:

🟢 **Managed** = "You define what, RAP handles how"
🔵 **Unmanaged** = "You define both what and how"

🔷 3. When to Choose Which?

Scenario Type	Choose Managed BO	Choose Unmanaged BO
CRUD on standard transparent table	✅ Perfect use case	❌ Not needed
You own the data model (custom Z-table)	✅ Framework can manage	⚠️ Use if complex business logic exists

Scenario Type	Choose Managed BO	Choose Unmanaged BO
CRUD on external system / legacy table	❌ RAP cannot manage	✅ You handle manually
Integrating with BAPI / Function Modules	❌ Framework can’t call BAPIs internally	✅ Perfect scenario
Need full control on SAVE sequence	❌ Framework controls	✅ You define it
CRUD via CDS-based entities	✅ Recommended	⚠️ Not mandatory
Migration from classic apps	⚠️ Partial fit	✅ More flexible
Simple master data apps	✅ Ideal	❌ Overkill
Complex transactional data	⚠️ Limited flexibility	✅ Full control

---

🔷 4. Understanding the Building Blocks

📘 Managed BO

- CDS defines **data model** (root and composition).
- Behavior Definition declares **behavior**.
- RAP automatically creates **runtime CRUD handlers**.
- You only implement business rules (if any).

📌 Example: Customer Master Maintenance

Step 1: Define Root CDS

```
@EndUserText.label: 'Customer Entity'
define root view entity ZI_Customer
  as select from zcustomer
{
  key customer_id,
      name,
      country,
      email
}
```

Step 2: Define Behavior

```
managed implementation in class zbp_i_customer unique;
define behavior for ZI_Customer alias Customer
persistent table zcustomer
lock master
{
  create;
  update;
  delete;

  field (mandatory) name;
  field (readonly) customer_id;
  determination set_default_country on modify { create; }
  validation check_email on save { create; update; }
}
```

Step 3: Implement Behavior

```
CLASS zbp_i_customer IMPLEMENTATION.

  METHOD set_default_country.
    LOOP AT entities ASSIGNING FIELD-SYMBOL(<cust>).
      IF <cust>-country IS INITIAL.
        <cust>-country = 'IN'.
      ENDIF.
    ENDLOOP.
  ENDMETHOD.

  METHOD check_email.
    LOOP AT entities ASSIGNING FIELD-SYMBOL(<cust>).
      IF <cust>-email NOT CP '*@*'.
        APPEND VALUE #( %msg = new_message_with_text( 'Invalid Email!' ) )
TO reported-customer.
      ENDIF.
    ENDLOOP.
  ENDMETHOD.

ENDCLASS.
```

🧩 **Real-time scenario:**
A customer master maintenance Fiori app — where end users create or edit customer data — works seamlessly with this setup. RAP manages the **database persistence**, **transaction commits**, **drafts**, and **locks**.

📘 Unmanaged BO

- CDS defines **data model**, but **you control persistence**.
- CRUD operations are manually implemented in BIMP class.
- Ideal when integrating with **existing logic, BAPIs, or custom validations**.

📌 Example: Sales Order Integration via BAPI

Step 1: Define Root CDS

```
@EndUserText.label: 'Sales Order Root Entity'
define root view entity ZI_SalesOrder
  as select from vbak
{
  key vbeln,
      auart,
      kunnr,
      erdat,
      netwr
}
```

Step 2: Define Behavior

```
unmanaged implementation in class zbp_i_salesorder unique;
define behavior for ZI_SalesOrder alias SalesOrder
lock master
{
  create;
  update;
  delete;
  read;

  field (readonly) vbeln;

  action submit_sales_order result [1] $self;

  mapping for vbak {
    vbeln = vbeln;
  }
}
```

Step 3: Implement Behavior (CRUD via BAPI)

```
CLASS zbp_i_salesorder IMPLEMENTATION.

  METHOD create.
    LOOP AT entities ASSIGNING FIELD-SYMBOL(<so>).
      CALL FUNCTION 'BAPI_SALESORDER_CREATEFROMDAT2'
        EXPORTING
          order_header_in = VALUE(bapisdhd1)(doc_type = <so>-auart customer
= <so>-kunnr).
      COMMIT WORK.
    ENDLOOP.
  ENDMETHOD.

  METHOD update.
    LOOP AT entities ASSIGNING FIELD-SYMBOL(<so>).
      CALL FUNCTION 'BAPI_SALESORDER_CHANGE'
        EXPORTING
          salesdocument = <so>-vbeln.
      COMMIT WORK.
    ENDLOOP.
  ENDMETHOD.

ENDCLASS.
```

🧩 **Real-time scenario:**
An organization already uses standard BAPIs to create/update sales orders. Instead of writing new DB logic, you can build a RAP unmanaged BO that **wraps BAPIs** and exposes them as **Fiori OData V4** services — achieving modernization without breaking legacy integration.

🔷 5. Real-World Comparative Scenarios

Scenario	Classic ABAP	Managed BO	Unmanaged BO
Z Table CRUD (Employee Master)	Manual forms, modules	Framework handles all	Overhead, not needed
Integrating with BAPI_SALESORDER_CREATEFROMDAT2	Use CALL FUNCTION manually	RAP cannot handle automatically	Perfect fit (wraps BAPI)
Draft-enabled document (Leave Request)	Manual temp tables	Built-in RAP draft handling	Complex to build
Complex validation before save	Manual coding	Validation section in BDEF	Validation in methods
Updating multiple entities	Nested updates	Deep Update supported	Developer controls sequence
Multi-level business object (Order → Item)	Need to code parent-child link	Composition in RAP manages automatically	Must code links manually
Display-only analytics	ALV / CDS view	RAP readonly projection	RAP read-only suitable
Lock handling	ENQUEUE/DEQUEUE manually	Automatic lock objects	Manual lock coding
Transaction rollback	Commit/Rollback manually	Framework handles	You manage rollback
Integrate with external API	HTTP call manually	Not ideal	Ideal (custom persistence logic)

🔷 6. Draft Handling — Only in Managed

Managed BOs support *draft-enabled* transactions — where a user can save work-in-progress data without committing to the DB.

📌 **Example Scenario:**
A user fills a **Purchase Requisition** but doesn't want to submit yet.

- Classic ABAP: You'd create a temporary Z table for "in-progress" records.
- RAP Managed: Just add `with draft;` in behavior — RAP does the rest (auto creates draft table, manages temporary state, locks, and activation logic).

```
define behavior for ZI_PurchaseReq alias PurchaseReq
persistent table zpurreq
lock master
with draft
{
  create;
  update;
  delete;
}
```

🔷 7. Key Advantages Comparison

Feature	Managed BO	Unmanaged BO
CRUD Implementation	Auto by framework	Manual by developer
Draft Handling	✅ Yes	❌ No
Save Sequence	Framework Controlled	Developer Controlled
Integration with BAPIs	❌ Hard	✅ Easy
Data Persistence	Automatic	Manual
Best for	Pure RAP apps	Migration / Hybrid apps
Upgrade Safety	High	Medium
Complexity	Low	Higher
Lock / Transaction	Managed automatically	Developer responsibility

---

🔷 8. Migration Perspective (Classic → RAP)

Old Practice	New RAP Managed Equivalent	New RAP Unmanaged Equivalent
Module Pool screen	Fiori Elements app	Fiori Elements app using custom persistence
BDC / Call Transaction	Not applicable	Wrap via unmanaged RAP
AUTHORITY-CHECK	@AccessControl annotations	Same annotations applicable
Temporary tables for drafts	Draft concept	Not supported
FORM routines	Behavior methods	Behavior methods
COMMIT WORK	Handled by RAP	Handled in your implementation
SELECT/UPDATE SQL	Auto handled by framework	Developer defines SELECT/UPDATE logic

---

🔷 9. How to Decide in a Project

If your project has...	Choose...
Fresh custom Z-tables, full control	Managed
Integration with legacy code or BAPIs	Unmanaged
Multiple dependent entities (header/item)	Managed
Existing procedural logic reused	Unmanaged
High need for performance with HANA logic	Either (CDS base for both)
Need draft / Fiori standard templates	Managed
Need to wrap old logic into new UI	Unmanaged

🔷 10. Golden Rules for ABAPers

1. **Think in Entities, not Tables.**
   Each CDS entity = a logical object (like a customer, sales order).
2. **Move away from procedural mindset.**
   Instead of "what to do step-by-step," describe *how the entity behaves*.
3. **Framework owns the transaction.**
   Don't use `COMMIT WORK` in managed BOs.
4. **Use Determinations & Validations wisely.**
   - *Determination*: system action (e.g., set default values).
   - *Validation*: user checks (e.g., email format, duplicate data).
5. **Use Drafts for business transactions.**
   Save time by avoiding temporary tables.
6. **Think of "Managed" as Smart CRUD** and "Unmanaged" as **Controlled CRUD**.

🔷 11. Practical Real-Time Use Cases Summary

Use Case	Preferred RAP Type	Why
Employee master maintenance	Managed	Simple, single-table CRUD
Purchase Requisition draft	Managed	Draft handling
Sales Order via BAPI	Unmanaged	BAPI integration
Vendor master sync with external API	Unmanaged	Custom persistence logic
Internal order tracking	Managed	Simple CRUD
Multi-level BOM	Managed	Composition logic
Legacy migration	Unmanaged	Wrap old logic
Project WBS structure	Managed	Hierarchical CRUD
Invoice posting using BAPI_ACC_DOCUMENT_POST	Unmanaged	Perfect for BAPI calls
Customer 360° dashboard (read-only)	Managed (read-only projection)	No CRUD needed

---

🔷 12. One-Liner Takeaway

🟢 Managed BO = Modern ABAP, framework-driven, perfect for clean CRUD apps.
🔵 Unmanaged BO = Flexible ABAP, developer-driven, perfect for legacy integration and control-heavy scenarios.
