**Capstone Project 3 – Part -1/2 V2D2**

 **Case Study 1**

 **A customer can make a payment either by Card or by Wallet or by Cash or by Net banking.**

1. **Draw a Use Case Diagram.**



1. **Derive Boundary Classes, Controller classes, Entity Classes.**

To derive **Boundary**, **Controller**, and **Entity** classes from the scenario, we'll follow **Object-Oriented Analysis and Design (OOAD)** approach, particularly using the **Model-View-Controller (MVC)** pattern often used in UML and software design.

* + **Entity Classes**

These are the core **business objects** that represent the **data and rules** of the application.

| **Class Name** | **Attributes (Example)** | **Description** |
| --- | --- | --- |
| Customer | customerId, name, email | Represents the customer making the payment |
| Payment | paymentId, amount, date, paymentMethod | Represents a payment |
| Card | cardNumber, expiryDate, cvv, cardHolderName | Details for card payment |
| Wallet | walletId, balance, provider | Represents a digital wallet |
| Cash | receivedBy, cashAmount | For recording cash payments |
| NetBanking | bankName, accountNumber, ifscCode | Details for net banking |

* + **Controller Classes**

These handle **requests**, perform **logic**, and coordinate between the **Boundary** and **Entity** classes.

| **Class Name** | **Responsibilities** |
| --- | --- |
| PaymentController | Accepts payment requests, verifies input, determines the payment method, and processes the transaction using the correct method |
| CustomerController | Handles customer-related operations like lookup, registration (if needed) |

* + **Boundary Classes**

These are the **interfaces** through which users or external systems interact with your system (UI/API).

| **Class Name** | **Description** |
| --- | --- |
| PaymentUI or PaymentPage | UI component/form through which a customer selects a payment method and submits a payment |
| CustomerUI | Interface for customer login, registration, or profile info |
| PaymentGatewayAPI | If it's an external API interacting with your app (like a third-party gateway) |

1. **Place these classes on a three tier Architecture.**
* **User layer**
* PaymentMethod Selection Boundary
* Card PaymentBoundary
* WalletPaymentBoundary
* CashPaymentBoundary
* NetBanking PaymentBoundary
* **Business Logic**
* PaymentController
* Card PaymentController
* WalletPaymentController
* CashPaymentController
* NetBanking PaymentController
* **Data Tier**
* Customer (Entity Class)
* Payment (Entity Class)
* Card (Entity Class)
* Wallet (Entity Class)
* BankAccount (Entity Class)
1. **Explain Domain Model for Customer making payment through Net Banking.**

ERDs are focused on the data structure and relationships, making them essential for database design, whereas Domain Models provide a broader view of the system's functionality and behaviour, guiding the development process. A domain model represents the core concepts (entities), their attributes, and the relationships between them in a specific business domain—in this case, **online payments using Net Banking**.

**Process Flow (Simplified)**

1. **Customer** chooses **Net Banking** on the UI.
2. **Payment** object is created with payment Method = Net Banking.
3. **Net Banking Details** are captured and linked to the Payment.
4. The system authenticates and processes via **Bank** API.
5. The Payment and Net Banking Details are updated based on the transaction status.

|  |
| --- |
| **Customer** |
| Customer Id | Name | Email | Phone No |
|   |  |  |  |
|  |  |  |  |

|  |
| --- |
| **Bank** |
| Bank ID | Bank Name | Netbanking URL |
|  |  |  |
|  |  |  |

|  |
| --- |
|  **Payment** |
| Payment Id | Amount | Payment Date | Status | Payment method |
|  |  |  |  |  |
|  |  |  |  |  |

|  |
| --- |
| **Account** |
| AccountNo | Account type | Balance | Account Holder Name |
|   |  |  |  |
|  |  |  |  |

|  |
| --- |
|  **Net Banking** |
| Bank Name | Account Number | IFSC Code | Transaction Reference No | Authentication Status |
|  |  |  |  |  |
|  |  |  |  |  |

|  |
| --- |
| **Authentication** |
| Username | Password | OTP |
|   |  |  |
|  |  |  |

|  |
| --- |
| **Transaction** |
| Transaction Id | Recipient Details | Amount | Time and Date Records |
|   |  |  |  |
|  |  |  |  |

**5. Draw a sequence diagram for payment done by Customer Net Banking.**



1. **Explain Conceptual Model for this Case.**

A **conceptual model** is a high-level representation of the system—**focused on the what** rather than the *how*. It defines key **entities**, their **attributes**, and **relationships**, but **without implementation details** (like data types, APIs, or DB schemas). This **conceptual model** defines the core **business concepts** (like Customer, Payment, and Payment Method) and how they relate—without touching any technical implementation. Think of it as how a business analyst or domain expert would describe the system before a developer touches it.

**Conceptual Model Overview**

**Entities:**

1. **Customer**
	* A person who initiates a payment.
2. **Payment**
	* A financial transaction made by the customer.
	* Has general details like amount, date, and status.
3. **Payment Method** (Abstract / General Concept)
	* The strategy or option used to complete a payment.
	* Can be of different types.
4. **Card**, **Wallet**, **Cash**, **Net Banking** (Specializations of Payment Method)
	* Specific implementations or types of payment methods.
	* Each has its own details:
		+ **Card**: Card Number, Expiry, CVV (conceptual, not technical)
		+ **Wallet**: Wallet Provider, Wallet ID
		+ **Cash**: May not need extra info
		+ **Net Banking**: Bank Name, Transaction ID

**Relationships:**

* A **Customer** initiates one or more **Payments**.
* A **Payment** uses one **Payment Method**.
* **Payment Method** can be **Card**, **Wallet**, **Cash**, or **Net Banking**.

**Summary:**

A **Customer** makes a **Payment**. That **Payment** is completed using one of several **Payment Methods**—either **Card**, **Wallet**, **Cash**, or **Net Banking**. Each payment method has its own relevant details. We're only describing what exists in the system conceptually—not how it's coded or stored.

1. **What is MVC architecture? Explain MVC rules to derive classes from use case diagram and guidelines to place classes in 3-tier architecture.**

**MVC (Model-View-Controller)** is a **design pattern** used to separate the concerns of a software application, especially in GUI or web applications.

**MVC Breakdown:**

* **Model**
	+ Represents the **data** and **business logic** of the application.
	+ Notifies views/controllers of any state changes.
	+ Examples: Payment, Customer, Bank, Wallet
* **View**
	+ Represents the **UI or presentation** layer.
	+ Displays data from the model to the user.
	+ Examples: PaymentForm, PaymentSummaryPage, CustomerDashboard
* **Controller**
	+ Handles **user input and interaction**.
	+ Interprets UI actions, calls the appropriate model logic, and updates the view.
	+ Examples: PaymentController, CustomerController

**Rules to Derive Classes from Use Case Diagram (Using MVC)**

Let’s say you have a use case diagram like:
**"Make Payment"**, **"Select Payment Method"**, **"Confirm Transaction"**

Here’s how you extract and classify classes:

**1. Identify Classes from Use Case Steps:**

* Look at **nouns** in the use cases (e.g., Customer, Payment, Card).
* These typically become **Model** classes.

**2. Identify Roles from Actors and Use Case Logic:**

* Use case **actions and flow** → likely **Controller** responsibilities.
* Example: Clicking “Submit Payment” calls PaymentController.submit().

**3. User Interface Screens:**

* Every screen/form/view in a use case is part of the **View**.
* Example: PaymentSelectionView, PaymentStatusView

**Guidelines to Place Classes in 3-Tier Architecture**

Now let’s map the **MVC-derived classes** into a **3-Tier Architecture**:

| **MVC Component** | **3-Tier Layer** | **Description** |
| --- | --- | --- |
| **View** | **Presentation Layer** | UI components, forms, screens |
| **Controller** | **Application Layer** | Processes requests, interacts with business logic |
| **Model** | **Business + Data Layer** | Business rules and data access logic (can be split) |

**Example Mapping:**

| **Class Name** | **MVC Role** | **3-Tier Placement** |
| --- | --- | --- |
| PaymentForm | View | Presentation Layer |
| PaymentController | Controller | Application Layer |
| Payment, Customer | Model | Business Layer |
| PaymentRepository | Model (Data) | Data Access Layer |

**Summary:**

**MVC** separates UI, logic, and data handling.

* **Use case diagrams** help derive model (nouns), controller (actions), and view (screens).
* **3-Tier architecture** organizes your codebase into Presentation, Application (Business), and Data layers.
* MVC can live within a 3-Tier structure: **MVC is about responsibility; 3-Tier is about deployment/logical layers.**

**8. Explain BA contributions in project (Waterfall Model – all Stages)**

Here's a breakdown of **Business Analyst (BA) contributions** in each stage of a project that follows the **Waterfall Model**—a sequential software development methodology with distinct phases.

 **1. Requirement Gathering & Analysis**

This is where the BA plays the most critical role.

**BA Contributions:**

* **Stakeholder Analysis**: Identify stakeholders and understand their needs.
* **Elicit Requirements**: Use techniques like interviews, workshops, surveys, document analysis, etc.
* **Document Requirements**: Create:
	+ Business Requirements Document (BRD)
	+ Functional Requirements Specification (FRS)
	+ Use cases / User stories
* **Validate Requirements**: Ensure they are clear, complete, testable, and aligned with business goals.
* **Sign-off Process**: Facilitate stakeholder review and formal approval.

**2. System Design**

This is more of a technical phase, but BA still supports the translation of business needs into technical specs.

**BA Contributions:**

* **Bridge Between Business and Technical Teams**: Clarify any ambiguities from requirements.
* **Review Design Documents**: Ensure alignment with business goals.
* **Support UI/UX Discussions**: Provide input on user needs and behaviour.

**3. Implementation (Coding)**

BA has a lower day-to-day role here, but support is still important.

**BA Contributions:**

* **Support Developers**: Answer questions and clarify requirements.
* **Change Requests**: Manage any changes in scope or requirements.
* **Traceability**: Ensure that every requirement is being implemented (using a traceability matrix).

**4. Integration and Testing**

Testing is typically led by QA, but BA ensures that tests align with requirements.

**BA Contributions:**

* **Review Test Plans & Cases**: Confirm they cover all business scenarios.
* **Support UAT (User Acceptance Testing)**:
	+ Coordinate with end-users
	+ Provide test data
	+ Validate test results
* **Defect Triage**: Help prioritize more serious issues in comparison with other issues based on business impact.

**5. Deployment**

System goes live.

**BA Contributions:**

* **Readiness Check**: Ensure all business conditions are met for go-live.
* **Training & Documentation**: Help prepare user manuals, training materials, and conduct sessions.
* **Business Communication**: Notify stakeholders, coordinate rollout plans.

**6. Maintenance**

Post-deployment support phase.

**BA Contributions:**

* **Monitor Feedback**: Gather feedback from users.
* **Manage Enhancements**: Identify opportunities for improvements.
* **Support Issue Resolution**: Assist in analysing defects and identifying causes.
* **Change Management**: Help with managing change requests and scope updates.

**9. What is conflict management? Explain using Thomas – Kilmann technique.**

**Conflict management** refers to the practice of identifying and handling conflicts in a sensible, fair, and efficient manner. It's an essential skill in both personal and professional settings, as conflicts are a natural part of human interaction. The goal of conflict management is not necessarily to eliminate conflict but to manage it constructively so that it leads to positive outcomes like better decision-making, stronger relationships, and improved team dynamics.

**Thomas–Kilmann Conflict Mode Instrument (TKI)**

The **Thomas–Kilmann technique**, developed by Kenneth Thomas and Ralph Kilmann, is one of the most widely used tools for understanding and managing conflict. It categorizes conflict-handling styles based on two dimensions:

1. **Assertiveness** – the degree to which you try to satisfy your own concerns.
2. **Cooperativeness** – the degree to which you try to satisfy the other person's concerns.

These dimensions create **five conflict-handling styles**:

**1. Competing (High Assertiveness, Low Cooperativeness)**

* **"I win, you lose."**
* This style is power-oriented and used when quick, decisive action is necessary, such as in emergencies.
* **Example:** A manager insists on implementing a new process despite team objections because of a critical deadline.

**2. Collaborating (High Assertiveness, High Cooperativeness)**

* **"Win-win."**
* Both parties work together to find a solution that fully satisfies both sides.
* **Example:** Two department heads jointly develop a budget plan that supports both of their needs after extensive discussion.

**3. Compromising (Moderate Assertiveness, Moderate Cooperativeness)**

* **"Split the difference."**
* Each party gives up something to reach a mutually acceptable solution.
* **Example:** Two colleagues agree to rotate their preferred project roles rather than one dominating the decision.

**4. Avoiding (Low Assertiveness, Low Cooperativeness)**

* **"No winners, no losers."**
* The person does not immediately pursue their own or others' concerns; they sidestep or withdraw from the conflict.
* **Example:** An employee chooses not to bring up a minor disagreement in a meeting to avoid confrontation.

**5. Accommodating (Low Assertiveness, High Cooperativeness)**

* **"I lose, you win."**
* One party puts the other’s concerns above their own to maintain harmony.
* **Example:** A team member agrees to help with extra work despite their own busy schedule to keep peace.

| **Style** | **Best Used When...** |
| --- | --- |
| **Competing** | Quick decisions are needed; unpopular actions must be implemented. |
| **Collaborating** | The goal is to find a solution that fully satisfies both parties. |
| **Compromising** | Time is limited, and both parties hold equally important goals. |
| **Avoiding** | The conflict is trivial, or the cost of confrontation outweighs the benefit. |
| **Accommodating** | The relationship is more important than the issue; the other person has a better solution. |

**10. List down the reasons for project failure.**

 **1. Poor Planning**

* Unclear goals and objectives
* Inadequate project scope definition
* Missing timelines or unrealistic deadlines

**2. Lack of Clear Objectives**

* Uncertain deliverables
* No measurable success criteria

**3. Inadequate Risk Management**

* Failing to identify and mitigate risks early
* Not having contingency plans

**4. Ineffective Communication**

* Poor communication among team members or stakeholders
* Misunderstanding requirements or expectations

**5. Lack of Stakeholder Involvement**

* Ignoring input from key stakeholders
* Stakeholders not being engaged or aligned

**6. Inadequate Resources**

* Shortage of staff, tools, budget, or time
* Wrong skill sets assigned to tasks

**7. Scope Creep**

* Uncontrolled changes or continuous growth in project scope
* No change management process in place

**8. Weak Leadership or Project Management**

* Inexperienced project manager
* Poor decision-making or leadership style

**9. Unrealistic Expectations**

* Overpromising results within tight timelines or limited resources
* Setting unachievable goals

**10. Poor Team Dynamics**

* Lack of collaboration or trust
* Team conflicts that go unresolved

**11. Technology Failures**

* Choosing the wrong tools or platforms
* Integration issues or outdated technology

**12. Failure to Monitor and Evaluate**

* No performance tracking or reporting
* Lack of timely feedback loops

**13. Noncompliance with Regulations**

* Overlooking legal, regulatory, or industry standards

**14. Ignoring the End-User or Customer**

* Not understanding user needs
* Deliverables don’t meet user expectations

**11. List the Challenges faced in projects for BA.**

Business Analysts (BAs) play a crucial role in the success of projects, but they often face unique challenges that can impact project outcomes. Here's a list of the most common **challenges faced by Business Analysts in projects:**

**1. Unclear or Evolving Requirements**

* Stakeholders often don’t know exactly what they want.
* Requirements change frequently, leading to confusion and rework.

 **2. Stakeholder Conflicts**

* Conflicting priorities or interests among stakeholders.
* Difficulty in getting everyone aligned on project goals.

 **3. Limited Stakeholder Engagement**

* Stakeholders may be unavailable or unresponsive.
* Lack of input leads to incomplete or inaccurate requirements.

**4. Inadequate Communication**

* Miscommunication between business and technical teams.
* Jargon or unclear documentation causing misunderstandings.

**5. Ambiguous Business Processes**

* Lack of existing process documentation.
* Legacy processes that are complex or not standardized.

**6. Scope Creep**

* Gradual expansion of project scope without proper impact analysis.
* Difficulty managing expectations when new requirements keep getting added.

**7. Tight Timelines**

* Pressure to deliver quickly may lead to skipping critical analysis steps.
* Limited time for thorough requirement gathering and validation.

**8. Technology Constraints**

* Existing systems may not support new requirements.
* Technical limitations not identified early in the project.

**9. Resistance to Change**

* End-users or departments reluctant to adopt new processes or systems.
* Organizational culture not supportive of transformation.

**10. Insufficient BA Involvement in Decision-Making**

* BAs not included in key project decisions.
* Misalignment between project execution and actual business needs.

**11. Poorly Defined Roles and Responsibilities**

* Overlapping roles between BAs, PMs, developers, or testers.
* Lack of clarity leads to delays and duplication of work.

**12. Difficulty in Validating Requirements**

* Trouble confirming that requirements meet business needs.
* Limited access to real users or data for validation/testing.

To overcome many of these challenges, effective stakeholder management, clear documentation, proactive communication and using tools like **BA templates, requirement traceability matrices and modelling techniques (like BPMN or UML)** can make a big difference.

**12. Write about Document Naming Standards.**

**Document naming standards** are predefined rules and formats used to name files and documents consistently across an organization or project. They help in **organizing, identifying, retrieving, and managing documents** more efficiently.

**Purpose of Naming Standards:**

* Improve document **searchability** and **accessibility**
* Avoid **confusion** caused by duplicate or unclear names
* Support **version control** and **audit tracking**
* Enable **collaboration** across teams and departments
* Promote **consistency** and **professionalism**

**Key Elements of a Naming Standard:**

A standard name might include components like:

| **Element** | **Description** | **Example** |
| --- | --- | --- |
| **Project Code** | Identifies the project | PRJ001 |
| **Document Type** | Type of document (e.g., report, plan) | REQ for Requirements |
| **Department/Team** | Team that owns the document | IT, HR, FIN |
| **Date** | In YYYYMMDD format for sorting | 20250423 |
| **Version Number** | Indicates revision level | v1.0, v2.1 |
| **Title/Short Desc.** | Brief description of the content | LoginFunctionality |

**Example Filename:**

PRJ001\_REQ\_IT\_20250423\_LoginFunctionality\_v1.0.docx

**Best Practices:**

* Avoid spaces—use **underscores** (\_) or **camelCase**
* Be **concise** but **descriptive**
* Use **standardized date formats** (e.g., YYYYMMDD)
* Include a **version number** to track changes
* Stick to **approved abbreviations**

**13. What are the Do’s and Don’ts of a Business analyst?**

**✅ Do’s for a Business Analyst**

 **1. Understand the Business**

* Learn the domain, goals, and challenges of the organization.

**2. Communicate Clearly**

* Use simple language with stakeholders; bridge the gap between business and IT.

**3. Ask the Right Questions**

* Dig deeper to uncover real needs, not just stated wants.

**4. Document Requirements Thoroughly**

* Use clear, structured formats: BRD, user stories, use cases, etc.

**5. Engage Stakeholders Early**

* Involve the right people at the right time to validate and refine requirements.

**6. Stay Neutral**

* Remain objective and focus on what's best for the business, not individuals.

**7. Adapt to Change**

* Be flexible and open to evolving requirements, especially in Agile environments.

**8. Use Visual Tools**

* Diagrams (like BPMN, flowcharts, wireframes) make complex ideas easier to understand.

**❌ Don’ts for a Business Analyst**

**1. Don’t Assume—Always Validate**

* Never rely solely on what one stakeholder says; cross-check information.

**2. Don’t Skip Documentation**

* Verbal agreements can be forgotten—document everything.

**3. Don’t Use Jargon with Non-Tech Stakeholders**

* Use clear, non-technical language when needed.

**4. Don’t Ignore Risks or Constraints**

* Always identify and communicate limitations and challenges early.

**5. Don’t Overpromise**

* Set realistic expectations about timelines, features, and outcomes.

**6. Don’t Take Sides in Conflicts**

* Be a facilitator, not a participant in disputes.

**7. Don’t Forget the End User**

* Requirements should reflect actual user needs and behaviors.

**8. Don’t Work in Isolation**

* Regularly collaborate with developers, testers, and business teams.

**14. Write the difference between packages and sub-systems.**

| **Feature** | **Package** | **Sub-system** |
| --- | --- | --- |
| **Definition** | A logical grouping of related elements like classes, interfaces, etc. | A larger component that represents a major part of the system with its own functionality. |
| **Scope** | Smaller in scope; used to organize code or models. | Broader; often represents a whole module or functional area. |
| **Purpose** | Organize and manage complexity in a project. | Divide the system into manageable, independent sections. |
| **Dependency** | Can have dependencies on other packages. | May include multiple packages and define external interfaces. |
| **UML Representation** | Represented as a tabbed folder icon. | Represented similarly to packages, but typically with stereotypes like «subsystem». |
| **Example** | com.bank.account, com.bank.loan | PaymentSystem, InventorySystem |

**Summary :**

* **Package** is mainly for **organization and structure** within a model or codebase.
* **Sub-system** represents a **self-contained unit of functionality** that can operate independently or be integrated into a larger system.

**15. What is camel-casing and explain where it will be used.**

**Camel casing** (or **camelCase**) is a **naming convention** in which words are combined into a single phrase **without spaces**, and each word **after the first starts with a capital letter**.

**Example:**

* customerName
* totalAmountDue
* getUserDetails

It’s called **camel case** because the capital letters in the middle resemble the **humps of a camel** 🐫.

**Where Is Camel Case Used?**

* **Programming:**
* Used for **variable names**, **function names**, and sometimes **object names**.
* Common in **JavaScript**, **Java**, **C#**, and many other languages.

orderTotal = 99.99;

function calculateTax() { ... }

* **APIs and JSON Naming:**
* camelCase is often used for **JSON keys** to match coding style.

{

 "firstName": "John",

 "lastName": "Doe"

}

* **Document Naming (sometimes):**
* In document/file names when spaces are not allowed or discouraged.
	+ Example: projectPlanFinal.docx instead of Project Plan Final.docx

**Note:**

CamelCase starts with a **lowercase letter**. If it starts with an uppercase (e.g., CustomerName), that’s called **PascalCase**.

**16. Illustrate Development server and what are the accesses does business analyst has?**

A **development server** is a **dedicated environment** where developers write, test, and debug code during the software development lifecycle.

**Definition (Simple Terms):**

A **development server** is a **sandbox** where developers can safely build and test new features or updates **without affecting live users or production systems**.

**Key Features of a Development Server:**

* ✅ **Safe Testing Zone** – It lets developers try out code without crashing the real app.
* 🧪 **Early Testing Ground** – Used for unit tests and integration tests.
* 🛠️ **Coding Playground** – Developers write and run their code here.
* 🔄 **Frequent Updates** – It’s updated constantly with the latest code changes.
* 🧱 **Mock or Dummy Data** – Often uses fake or sample data to simulate real scenarios.

**Where It Fits:**

Business Needs ➝ Development ➝ Testing ➝ Staging ➝ Production

* **Development Server (DEV)**: For coding & initial testing.
* **Testing/Staging Server**: For QA and user acceptance testing.
* **Production Server**: The live system used by real users.

**Example Scenario:**

You're building an e-commerce app. A new "Dark Mode" feature is being developed:

* The developer adds the feature on the **development server**.
* They test to make sure it doesn’t break anything.
* Once it works, the code is sent to QA for deeper testing.

**What Access Does a Business Analyst Have?**

Business Analysts (BAs) typically focus on **gathering requirements, analysing processes, and communicating between business and tech teams**, rather than writing code or deploying software. So, their access in a development environment is **limited and non-technical**.

Typical BA Access:

* **Requirement Management Tools** (like Jira, Confluence, Azure DevOps)
* **Read-Only Access to DEV Environment**:
	+ To view data for validation
	+ To observe features implemented based on their specs
* **Testing Tools or Sandboxes (**testing environment in a computer system in which new or untested software or coding can be run securely) for validating functionality (UAT level)
* **Documentation Repositories for storage** (SharePoint, Confluence, Git repos for docs)
* **Access to Logs or Dashboards** (if needed to trace data flow or issues)
* **Stakeholder Communication Platforms** (MS Teams, Slack, etc.)

**What BAs Usually Don’t Have:**

* Admin access to the development server
* Code commits or deployment rights
* Deep database access (unless explicitly required)
* Access to production (in most environments)

**16. What is Data Mapping?**

**Data Mapping** is the process of **matching data fields** from one source to their corresponding fields in another system or database. In simple words, it's like creating a **translation guide. Eg :** When you see “First\_Name” in System A, it matches “FName” in System B.

**Data mapping is essential when:**

* Migrating data from one system to another
* Integrating multiple systems (like CRM to ERP)
* Ensuring data flows correctly in data pipelines or APIs

**Its’s used In:**

* Data migration
* API integration
* ETL (Extract, Transform, Load) processes
* Business Intelligence (BI) and reporting
* System upgrades

| **Old System (Source)** | **New System (Target)** |
| --- | --- |
| emp\_id | employeeNumber |
| emp\_name | fullName |
| dob | dateOfBirth |
| dept | departmentCode |

Example : When filling out a form in another language using a **dictionary**:

* Your dictionary (data map) tells you what word in your language matches the form’s field.
* Without the right map, you could enter your **last name** where it asks for your **birthday**

**18. What is API. Explain how you would use API integration in the case of your application Date format is dd-mm-yyyy and it is accepting some data from Other Application from US whose Date Format is mm-dd-yyyy.**

**API** stands for **Application Programming Interface**.

It’s a **bridge** that allows two different applications to **talk to each other** and **exchange data**.

**Example :** At a restaurant:

* **You** are the application that needs food (data).
* **The kitchen** is the system where the food (data) is prepared.
* **The waiter (API)** takes your order to the kitchen and brings back your food — correctly and efficiently.

**API Integration :**

* **Establish API Communication: Set-up API communication between your application and the other application (US application) to exchange data.**
* **API Receives Data** from the US system.
* **Middleware / Backend Code Converts the Format:**
* Detects incoming date format (Eg: 04-24-2025 in the format of mm-dd-yyyy)
* Parses it
* Converts to your required format (dd-mm-yyyy)
* Final format your system uses: Eg: "orderDate": "24-04-2025 in the format of dd-mm-yyyy)"
* **Data is saved or displayed correctly** in your application.
* **Business Analyst's Role in API Integration:**

As a BA, your job is to:

* Identify these kinds of data format differences
* Document **mapping rules** and **transformation logic**
* Ensure developers know the **expected formats**
* Coordinate **testing** to ensure correct data flow