**Q1. Draw a Use Case Diagram.**

Ans.

A **use case diagram** is a type of **UML (Unified Modeling Language)** diagram that visually represents the functional requirements of a system. It shows **actors** (users or other systems), **use cases** (interactions or services the system provides), and the **relationships** between them.

 **Components of a Use Case Diagram:**

1. **Actors**:
	* Represent users or other systems interacting with your system.
	* Can be primary (initiating the interaction) or secondary (responding).
2. **Use Cases**:
	* Represent the functionality or services provided by the system.
	* Shown as ovals with the name of the use case inside.
3. **System Boundary**:
	* A rectangle that defines the scope of the system.
4. **Relationships**:
	* **Association** (solid line): Connects actors to use cases.
	* **Include** (dashed arrow with label): One use case always includes another.
	* **Extend** (dashed arrow with label): One use case optionally extends another.
	* **Generalization** (hollow triangle): Inheritance between actors or use cases.

Given below is the Use Case Diagram for Customer Making Payment:



**Q2. Derive Boundary Classes, Controller classes, Entity Classes.**

Ans.

* **Boundary Classes** – Interface between the system and actors (e.g., UI or communication interface).
* **Controller Classes** – Handle the flow of control from the boundary to the entity.
* **Entity Classes** – Represent the business logic/data of the system.

##  **1. Actors and Use Case Summary**

* **Actor**: Customer
* **Use Case**: Make Payment
	+ by Card
	+ by Wallet
	+ by Cash
	+ by Net Banking

## **Entity Classes**

These contain core business logic/data.

| **Class** | **Attributes** | **Responsibilities** |
| --- | --- | --- |
| Payment | amount, date, status, paymentMethod | Base class or interface for all payments |
| CardPayment | cardNumber, expiryDate, cvv | Process card payment |
| WalletPayment | walletId, walletBalance | Deduct from digital wallet |
| CashPayment | receiptNo | Handle cash transactions |
| NetBankingPayment | bankName, accountNumber, ifscCode | Process net banking payment |

## **Controller Classes**

Coordinate between boundary and entity classes.

| **Class** | **Responsibilities** |
| --- | --- |
| Payment Controller | Handles user requests, selects payment method, triggers business logic |
| Card Payment Controller, Wallet Payment Controller, etc. | Handle specific method logic if needed separately |

## **Boundary Classes**

These are user interfaces or communication interfaces.

| **Class** | **Responsibilities** |
| --- | --- |
| Payment UI | Collects payment input from user |
| Card Payment Form, Wallet Payment Screen, etc. | Specific forms/screens for each method |

**Q3. Place these classes on a three tier Architecture**

Ans.

**Three-Tier Architecture** is a software architecture pattern that organizes applications into **three logical layers**:

### 1. ****Presentation Tier (UI Layer)****

* Directly interacts with the user.
* Responsible for displaying information and collecting input.

### 2. ****Application Tier (Business Logic / Controller Layer)****

* Processes user input from the presentation layer.
* Makes logical decisions and calculations.
* Calls data access or business rules in the data layer.

### 3. ****Data Tier (Persistence Layer)****

* Manages the application's data.
* Stores and retrieves data from databases or external storage.

### ****1. Presentation Tier (UI Layer)****

Handles all interactions with the user (Customer)

* PaymentUI
* CardPaymentForm
* WalletPaymentScreen
* NetBankingPaymentForm

These display forms to users, take inputs like amount, card details, etc.

### ****2. Application Tier (Business Logic Layer)****

Orchestrates processing of the payment logic

* Payment Controller
* (Optional) Card Payment Controller, Wallet Payment Controller if logic is separated

### ****3. Data Tier (Persistence Layer / Entity Layer)****

Manages and processes core business objects and persistence

* Payment
* CardPayment
* WalletPayment
* CashPayment
* NetBankingPayment

These classes represent actual payment data and contain methods for processing the payment and interacting with the database (e.g., saving transaction details).

**Q4. Explain Domain Model for Customer making payment through Net Banking**

**Ans.**

A Domain Model is a conceptual representation that defines the Structure, relationships and Behaviours of entities within a specific problem domain.

Below given is the Domain Model for a customer making a payment through Net Banking:



**Q5. Draw a sequence diagram for payment done by Customer Net Banking.**

Ans.

A **Sequence Diagram** is a type of diagram used in **UML (Unified Modelling Language)** to show how **objects or components in a system interact with each other over time**. It’s especially useful for modelling the flow of logic or communication in a use case or system function.

**Key Elements of a Sequence Diagram:**

1. **Actors/Objects:** Represented as vertical lines (lifelines), showing participants like users or system components.
2. **Lifelines:** Vertical dashed lines showing the object's life during the interaction.
3. **Messages:** Horizontal arrows between lifelines, representing the communication or method calls.
	* Solid arrows: synchronous messages (calls)
	* Dashed arrows: return messages
4. **Activation Bars:** Thin rectangles on lifelines showing when an object is active/performing an action.

Below given is the Sequence Diagram for a Customer using Net Banking:



**Q6. . Explain Conceptual Model for this Case.**

Ans.

A **Conceptual Model** represents the high-level structure of the system involved in the Net Banking payment process. It identifies **key entities**, their **relationships**, and **attributes** without going into implementation or database details. It's a crucial part of system design in object-oriented analysis.

 **Key Entities in the Conceptual Model:**

1. Customer
2. Merchant
3. Payment Gateway
4. Bank (Customer Bank & Merchant Bank)
5. Order
6. Transaction

**Entity Descriptions and Attributes:**

| **Entity** | **Attributes** |
| --- | --- |
| **Customer** | Customer ID, Name, Email, Phone Number, Bank Account Details |
| **Merchant** | Merchant ID, Name, Website, Merchant Bank Details |
| **Order** | Order ID, Order Date, Amount, Status, Customer ID, Merchant ID |
| **Payment Gateway** | Gateway ID, Name, API Key, Status |
| **Bank** | Bank ID, Bank Name, IFSC Code, Account Details |
| **Transaction** | Transaction ID, Amount, Date Time, Status, Payment Mode, Linked Bank |

**Relationships Between Entities:**

* Customer places Order
* Order is processed by Merchant
* Merchant uses Payment Gateway
* Payment Gateway interacts with both Customer's and Merchant's Bank
* Transaction is created for every payment process

**Q7. What is MVC architecture? Explain MVC rules to derive classes from use case diagram and guidelines to place classes in 3-tier architecture**

Ans.

**MVC (Model-View-Controller)** is a **software design pattern** that separates an application into **three main logical components**, each with specific responsibilities:

### ****1. Model****

* **Represents the business logic and data** of the application.
* It directly manages the data, logic, and rules of the application.
* Examples: User, Order, Transaction, etc.

###  ****2. View****

* **The UI (User Interface)** component.
* Displays data from the Model to the user.
* Notifies the controller of user input.
* Examples: HTML page, GUI screen, mobile app UI.

### ****3. Controller****

* **Acts as an interface** between Model and View.
* Handles user input and updates the Model and View accordingly.
* Example: Login Controller, Payment Controller.

##  **MVC Rules to Derive Classes from Use Case Diagram**

1. **Identify Controllers:**
	* Each use case generally maps to a **Controller** class.
	* Example: "Make Payment" → PaymentController.
2. **Identify Models (Business Entities):**
	* Extract nouns and domain objects from use case descriptions.
	* These become **Model** classes.
	* Example: "Customer logs in" → Customer, LoginCredential.
3. **Identify Views:**
	* For each user interaction, derive a **View**.
	* These are typically UI screens/pages.
	* Example: "Display confirmation page" → PaymentSuccessView.

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

The 3-tier architecture maps well with MVC:

###  1. ****Presentation Tier****

* Handles UI logic.
* Interacts with the Controller to send/receive user data.
* Contains: JSP/HTML pages, React/Angular components, etc.

### 2. ****Business Logic Tier****

* Manages application logic, workflows, and rules.
* Controllers process user input.
* Models manage business data.

###  3. ****Data Access Tier (Persistence Layer)****

* Interacts with the database.
* Responsible for CRUD operations.
* Contains: DAOs, Repositories, ORM classes

**Q8. Explain BA contributions in project (Waterfall Model – all Stages)**

Ans.

**Business Analyst (BA) Contributions in a Project Using the Waterfall Model**

The **Waterfall Model** is a **linear and sequential software development model**, where each phase must be completed before the next begins. A **Business Analyst (BA)** plays a crucial role at **every stage** of the Waterfall lifecycle to ensure requirements are clearly defined, communicated, and implemented correctly.

**1. Requirements Gathering & Analysis**

* **Responsibilities:**
	+ Meet stakeholders to understand business needs.
	+ Elicit and document requirements (Business Requirements Document – BRD).
	+ Analyze feasibility and scope.
	+ Define use cases, process flows, and high-level functional requirements.
* **Deliverables:**
	+ BRD, Use Case Diagrams, Requirement Traceability Matrix (RTM), Stakeholder Analysis.

 **2. System Design**

* **Responsibilities:**
	+ Collaborate with solution architects and designers to ensure the design aligns with requirements.
	+ Clarify business needs and constraints during design.
	+ Help translate business requirements into functional specifications.
* **Deliverables:**
	+ Functional Requirement Specifications (FRS), User Interface Wireframes (in collaboration with UX).

 **3. Implementation (Development)**

* **Responsibilities:**
	+ Act as a liaison between developers and business users.
	+ Resolve ambiguities in requirements.
	+ Provide support to developers on business logic and flows.
	+ Participate in review meetings to track progress.

 **4. Testing**

* **Responsibilities:**
	+ Assist QA in understanding the requirements.
	+ Validate test cases against requirements.
	+ Perform UAT (User Acceptance Testing) coordination.
	+ Report and track defects from a business perspective.
* **Deliverables:**
	+ UAT Test Plan, Test Case Review, Defect Reports.

 **5. Deployment**

* **Responsibilities:**
	+ Assist in go-live planning and risk identification.
	+ Ensure that business is prepared for changes (documentation, training).
	+ Coordinate with stakeholders for final sign-off.

 **6. Maintenance**

* **Responsibilities:**
	+ Gather feedback and new enhancement requests.
	+ Analyze defects or issues in the live system.
	+ Update documentation and requirements for changes.

**Q9. What is conflict management? Explain using Thomas – Kilmann technique**

Ans.

**Conflict Management** refers to the process of identifying, addressing, and resolving disagreements or disputes between individuals or groups in a constructive way. The goal is to minimize negative outcomes and promote healthy collaboration, especially in teams or projects.

Conflicts can arise due to:

* Differing priorities or goals
* Miscommunication
* Role ambiguity
* Resource limitations
* Personality clashes

##  **Thomas-Kilmann Conflict Management Technique**

The **Thomas-Kilmann Conflict Mode Instrument (TKI)** is a widely used model that identifies **five conflict-handling styles**, based on two dimensions:

* **Assertiveness** (the degree to which you try to satisfy your own concerns)
* **Cooperativeness** (the degree to which you try to satisfy others’ concerns)

###  ****Five Conflict Management Styles (TKI)****:

| **Style** | **Assertiveness** | **Cooperativeness** | **Description** |
| --- | --- | --- | --- |
| **1. Competing** | High | Low | Win-lose approach, assertive but uncooperative. Used in emergencies or when quick, decisive action is needed. |
| **2. Collaborating** | High | High | Win-win approach, assertive and cooperative. Seeks solutions that satisfy all parties. Ideal but time-consuming. |
| **3. Compromising** | Medium | Medium | Both parties give up something to reach a middle ground. Acceptable solution but may not fully satisfy anyone. |
| **4. Avoiding** | Low | Low | Withdraws from the conflict, neither assertive nor cooperative. Useful when the issue is trivial or needs time. |
| **5. Accommodating** | Low | High | Yield to others' needs, unassertive but cooperative. Maintains harmony but may suppress own needs. |

**Q10. List down the reasons for project failure**

Ans.

Projects can fail for a variety of reasons, ranging from poor planning to lack of stakeholder engagement. Below is a comprehensive list of common causes:

**1. Unclear or Incomplete Requirements**

Vague or constantly changing requirements lead to confusion and scope creep.

 **2. Lack of Stakeholder Involvement**

If key users, sponsors, or stakeholders are not engaged, the solution may not meet real needs.

 **3. Poor Project Planning**

Inaccurate estimates, no clear milestones, or lack of risk planning can derail a project.

 **4. Inadequate Communication**

Miscommunication between team members, stakeholders, or vendors causes delays and errors.

 **5. Unrealistic Deadlines**

Setting overly aggressive timelines leads to rushed work and poor quality.

 **6. Inadequate Risk Management**

Failure to identify and prepare for risks results in unanticipated problems.

 **7. Poor Leadership or Project Governance**

Weak project sponsorship or leadership leads to misaligned priorities and decision-making delays.

 **8. Lack of Resources**

Insufficient budget, skilled staff, or tools make it hard to complete tasks effectively.

 **9. Team Issues**

Low morale, poor collaboration, or unclear roles/responsibilities reduce productivity.

 **10. Scope Creep**

Uncontrolled changes or continuous addition of features without time or budget adjustments.

 **11. Technical Failures**

Choosing inappropriate technology, integration failures, or lack of testing can cause breakdowns.

 **12. Poor Quality Assurance**

Inadequate testing leads to undetected bugs and defects in the final product.

 **13. Inadequate Change Management**

Users resist adopting the new system due to lack of training or communication.

**14. Vendor or Third-party Issues**

External dependencies or vendors fail to deliver as expected.

 **15. No Clear Definition of Success**

Without agreed-upon goals and KPIs, it’s hard to measure progress or success.

**Q11. List the Challenges faced in projects for BA.**

Ans.

A **Business Analyst (BA)** plays a key role in bridging the gap between stakeholders and technical teams. However, they often face several challenges during the project lifecycle:

 **1. Unclear or Evolving Requirements**

* Stakeholders often don’t know what they want initially.
* Frequent changes can lead to scope creep and rework.

 **2. Communication Gaps**

* Misunderstandings between business users, developers, and other stakeholders.
* Multiple stakeholders with conflicting priorities.

 **3. Stakeholder Unavailability or Non-Cooperation**

* Key users may not be available for meetings, validations, or feedback.
* Difficult to gather accurate and complete requirements.

 **4. Lack of Domain Knowledge**

* If the BA is new to the industry or project, understanding processes and terminology can be difficult.

 **5. Resistance to Change**

* Users may resist adopting new systems or processes due to comfort with existing ones.

 **6. Ambiguity in Requirements**

* Vague or poorly defined requirements lead to misinterpretation by development and QA teams.

 **7. Time Constraints**

* Pressure to gather and document requirements quickly, leading to oversight or errors.

 **8. Managing Scope Creep**

* Stakeholders may request new features mid-project without adjusting timelines or budget.

 **9. Difficulty in Prioritizing Requirements**

* Stakeholders may consider all requirements equally critical, making prioritization difficult.

 **10. Conflicting Stakeholder Interests**

* Different departments or users may have conflicting needs or expectations.

**Q12. Write about Document Naming Standards**

Ans.

**Document Naming Standards** refer to a consistent and structured method for naming documents within a project or organization. These standards ensure that documents are easy to identify, retrieve, manage, and track across teams and stages of the project lifecycle.

 **Purpose of Document Naming Standards**

* Improve clarity and consistency
* Enable quick search and retrieval
* Avoid duplication and confusion
* Facilitate version control
* Support collaboration among stakeholders

 **Key Elements of a Document Naming Standard**

| **Element** | **Description** | **Example** |
| --- | --- | --- |
| **Project Code** | Short code identifying the project | BANKSYS |
| **Document Type** | Abbreviation of the type of document | BRD, FRS, UAT, SRS |
| **Module/Feature** | Optional – name of the feature or module | LOGIN, PAYMENT |
| **Version Number** | Indicates the revision of the document | v1.0, v2.1 |
| **Date** | (Optional) Format YYYYMMDD or DDMMYYYY for tracking | 20250428 |
| **Author/Team Initials** | (Optional) Who created or modified the document | JD, BA\_Team |

 **Best Practices for Naming Standards**

* Use **consistent separators** (e.g., underscore \_ or hyphen -)
* Avoid **spaces and special characters**
* Always include **version numbers**
* Keep names **short but meaningful**
* Maintain a **document control log** to track versions and updates

**Q13. What are the Do’s and Don’ts of a Business analyst.**

Ans.

A **Business Analyst** plays a crucial role in bridging the gap between business needs and technical solutions. To succeed, BAs must follow certain best practices while avoiding common pitfalls.

## **Do’s of a Business Analyst:**

| **Do** | **Explanation** |
| --- | --- |
| **1. Communicate Effectively** | Listen actively, clarify doubts, and ensure clear communication between stakeholders and the technical team. |
| **2. Document Clearly** | Maintain thorough, organized, and understandable documentation (BRD, FRS, use cases, etc.). |
| **3. Understand the Business Domain** | Learn the client's business processes, terminology, and objectives. |
| **4. Be Detail-Oriented** | Carefully capture all requirements, exceptions, and edge cases to avoid ambiguity. |
| **5. Ask Questions** | Never assume—clarify everything to ensure all requirements are fully understood. |
| **6. Manage Stakeholder Expectations** | Set realistic timelines, clarify scope, and communicate changes proactively. |
| **7. Facilitate Collaboration** | Act as a bridge between business, developers, QA, and UX teams. |
| **8. Stay Flexible and Adaptive** | Be open to changing requirements and agile workflows. |
| **9. Use Visual Aids** | Support documents with wireframes, flowcharts, or mockups to enhance clarity. |
| **10. Maintain Traceability** | Use tools and methods (like RTM) to track requirements from start to finish. |

##  **Don’ts of a Business Analyst:**

| **Don't** | **Explanation** |
| --- | --- |
| **1. Don't Make Assumptions** | Always validate information with stakeholders instead of guessing. |
| **2. Don't Ignore Stakeholders** | Every voice matters. Ignoring users or SMEs can lead to major requirement gaps. |
| **3. Don’t Overcomplicate Requirements** | Use simple language and avoid jargon unless necessary. |
| **4. Don’t Delay Documentation** | Postponing documentation can result in missed or forgotten requirements. |
| **5. Don’t Avoid Conflict** | Address and resolve conflicts early rather than avoiding them. |
| **6. Don't Be Rigid** | Requirements and processes may evolve; adaptability is key. |
| **7. Don't Skip Validation** | Never finalize requirements without validating them with users and stakeholders. |
| **8. Don’t Ignore Technical Feasibility** | Understand system constraints and collaborate with developers early. |
| **9. Don’t Work in Isolation** | Engage continuously with the team and stakeholders. |
| **10. Don’t Focus Only on Features** | Understand and prioritize based on business value, not just quantity of features. |

**Q14. Write the difference between packages and sub-systems.**

Ans.

In the context of software development and system design, both **Packages** and **Sub-systems** are used to organize and manage components of a system. However, they have different purposes, scope, and characteristics.

| **Feature** | **Package** | **Sub-system** |
| --- | --- | --- |
| **Definition** | A grouping of related classes or modules in a project | A self-contained functional unit within a larger system |
| **Scope** | Smaller, code-level organization | Larger, functional component |
| **Functionality** | Code structure and maintainability | A specific domain or service within the system |
| **Dependency** | Dependencies are usually at the module/class level | Dependencies involve interaction with other sub-systems |
| **Independence** | Not independent, part of a larger structure | Can operate as a standalone functional unit |
| **Example** | transactions, user-management packages | Transaction Sub-system, User Authentication Sub-system |

**Q15. What is camel-casing and explain where it will be used.**

Ans.

**Camel casing** is a **convention** for writing compound words or phrases where each word begins with a capital letter except the first word. This style of writing is typically used in **programming languages** to name variables, functions, and other identifiers in a way that improves readability.

* **Camel Casing** refers to the practice of writing multi-word identifiers where the first word is in **lowercase**, and all subsequent words have their first letter in **uppercase** with no spaces or underscores between words.
* The term **"camel case"** comes from the **humps** of uppercase letters, which resemble the humps of a camel.

 **Where Camel Casing Is Used**

Camel casing is commonly used in various places in software development, especially in programming languages like Java, C#, JavaScript, Python, and others.

1. **Variables**:
	* **Lower Camel Case** is used for variable names.
	* Example: user Age, total Amount, account Balance
2. **Functions or Methods**:
	* **Lower Camel Case** is used for function and method names.
	* Example: get User Info(), calculate Total(), is User Authenticated()
3. **Class Names**:
	* **Upper Camel Case** (Pascal Case) is used for class names.
	* Example: User Account, Product Details, Order Processor
4. **Object Properties/Fields**:
	* **Lower Camel Case** is typically used for properties/fields of objects.
	* Example: user Name, account Number, product Price
5. **File Names** (for some programming languages and environments):
	* **Camel Case** can also be used for naming files, though some languages might use other conventions.
	* Example: UserProfile.js, Employee Details.cs
6. **Constants (in some languages)**:
	* In some languages, **UPPERCASE\_WITH\_UNDERSCORES** is preferred for constants, but **Camel Case** might also be used in other contexts.
	* Example: MAX\_RETRIES (common for constants), but for variables: max Retries

**Q16. Illustrate Development server and what are the accesses does business analyst has?**

Ans.

A **Development Server** is an environment where developers write, test, and deploy code during the software development lifecycle (SDLC). It is separate from the production environment and typically includes tools, databases, and configurations specifically for development, testing, and debugging.

 **Development Server Overview**

A **Development Server** can have various configurations depending on the development process, the technology stack, and the organization’s needs. Here's a typical illustration:

1. **Purpose**:
	* The development server is used for **coding, testing, debugging**, and performing **integration** of features during the software development phase.
	* It's an isolated environment where new code is tested before it's moved to staging or production servers.
2. **Components**:
	* **Source Code**: The application code that developers write (e.g., in Java, Python, JavaScript).
	* **Web Server**: A server that handles HTTP requests, such as Apache, Nginx, or integrated web servers in frameworks.
	* **Database**: A separate database instance or a copy of the production database, often with test data for developers to work on.
	* **Application Server**: Server running the app to execute backend business logic.
	* **Version Control**: Tools like Git to manage different versions of the code.
	* **CI/CD Tools**: Continuous integration and continuous deployment tools (e.g., Jenkins) to automatically test and deploy changes to the development environment.
	* **Testing Tools**: Automated unit tests, integration tests, and test case scripts for QA.

 **Accesses a Business Analyst (BA) Might Have on a Development Server**

While a **Business Analyst (BA)** typically focuses on the requirements, documentation, and communication between stakeholders, they may still need to interact with the development server in certain contexts. However, BAs usually don’t have as much direct access to code or configuration settings as developers or system admins.

**1. Access to Test Environments**

* **Reason**: The BA might need to validate the functionality of a feature or perform **UAT (User Acceptance Testing)** to verify if the software meets business requirements.

**2. Viewing and Reviewing Logs and Reports**

* **Reason**: The BA may need to monitor the **progress** of specific features or issues (e.g., bug tracking, feature testing).

**3. Access to APIs and Documentation**

* **Reason**: If the business analyst is involved in defining integration points between systems, they may need access to **API documentation** and endpoints exposed by the development server.

**4. Collaboration with Developers and QA Team**

* **Reason**: BAs are often involved in **regular meetings** with developers and QA to provide feedback or clarification on requirements.

**5. Verifying Data Inputs and Outputs**

* **Reason**: The BA may need to check if **data flow** (inputs, processes, outputs) adheres to business rules.

 **Q17: What is Data Mapping?**

**Data Mapping** refers to the process of linking or associating data elements from one database, data model, or system to another. It involves creating a relationship between two different data structures or schemas, ensuring that data is correctly transferred, transformed, or integrated across systems or processes.

 **Purpose of Data Mapping:**

* **Data Transformation**: Ensures that data is accurately converted between formats or systems.
* **Data Integration**: Facilitates the integration of data from multiple sources by mapping fields from one system to another.
* **ETL Processes**: Data mapping is often used in **ETL (Extract, Transform, Load)** processes to extract data from source systems, transform it according to business rules, and load it into a target system.
* **Data Migration**: Helps in migrating data from one platform to another by establishing a clear relationship between old and new data formats.

 **Types of Data Mapping**

1. **One-to-One Mapping**:
	* A **single data element** in the source system corresponds to a **single data element** in the target system.
	* Example: Mapping a field **first\_name** in the source system to **firstName** in the target system.
2. **One-to-Many Mapping**:
	* A **single data element** in the source system corresponds to **multiple data elements** in the target system.
	* Example: A single **address** field in the source system might be mapped to multiple fields in the target system like **street**, **city**, **zipcode**.
3. **Many-to-One Mapping**:
	* Multiple source elements are mapped to a **single target element**.
	* Example: Several fields like **first\_name**, **middle\_name**, and **last\_name** in the source system are combined and mapped to a single **full\_name** field in the target system.
4. **Many-to-Many Mapping**:
	* Multiple source elements correspond to multiple target elements.
	* Example: A source system's **orders** table could map to multiple related tables in the target system such as **order\_items**, **shipping\_details**, and **customer\_info**.

**Q18. 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.**

Ans.

An **API (Application Programming Interface)** is a set of **rules, protocols, and tools** that allows different software applications to **communicate** with each other. It specifies how different software components should interact and enables data exchange and functionality sharing between systems.

APIs are used to allow third-party applications to access certain features or data from your application, system, or service in a secure and controlled way.

 **Steps to Use API Integration for Date Format Conversion**

**1. Define the API Integration Process**

* The first step is to **establish the connection** between your application and the US-based application through the **API**.
* The API can either be a **RESTful API** (using HTTP methods) or another format (such as SOAP or GraphQL).

For this example, we assume a **RESTful API** where the US application sends data through an HTTP POST request, and your application receives the data via an HTTP GET or POST request.

**2. Extract the Date from the API Response**

* When your application receives the data from the US application, you’ll get it in the format mm-dd-yyyy.

**3. Convert the Date Format**

To handle the conversion, your application needs to **parse the incoming date string** and then **reformat** it into the desired format (dd-mm-yyyy).

In many programming languages, there are built-in **date parsing** and **formatting** functions to facilitate this.

**4. Send the Data Back to the Application**

Once the date has been converted, you can then **send the correctly formatted date** back to the application or database, or proceed with other operations as required.