Section A: General Knowledge

1.

Security Considerations	
Encrypt Data	While data is In Transit Use TLS (Transport Layer Security) to protect data as it moves between users and servers
	Encrypt sensitive data stored in databases (like… customer details, transactions) using strong encryption like AES-256.
Strong Authentication	Implement multi-factor authentication (MFA) to verify user identities.
	Enforce strong password policies and require regular updates.
Access Control	Use Role-Based Access Control (RBAC) to ensure users only access data and features relevant to their roles.
	Regularly review and update permissions to prevent unauthorized access.
Secure APIs	Validate and sanitize all inputs to prevent attacks like SQL injection.
	Use API gateways and rate limiting to control access and prevent abuse.
Comply with Regulations	Follow industry standards like GDPR (data privacy), PCI DSS (payment security), and PSD2 (EU payment rules).
	Stay updated with local regulations, such as CBN guidelines for fintechs in Nigeria.
Fraud Detection	Facilitate real-time monitoring and anomaly detection to identify suspicious activity.
	Use machine learning models to recognise patterns that indicate fraud. Set up alerts for anomalous transactions (e.g., large transfers, multiple failed login attempts).
Secure Coding Practices	Follow secure coding guidelines to avoid vulnerabilities like XSS and CSRF.
	Regularly review code and conduct security testing (e.g., static and dynamic analysis). Use dependency scanning tools to identify vulnerabilities in third-party libraries.
Data Privacy	Only collect data to the minimum extent required.
	Anonymise or pseudonyms sensitive information where possible. Provide users with transparency and control over their data (e.g., opt-in/opt-out).
Logging and Monitoring	Keep meticulous records of all user activity and transactions for auditing and forensic examination.
	Watch logs in real-time for detection and response to security breaches in a timely manner.
Backup and Recovery	Regularly back up critical data and test recovery processes.
	Have a business continuity plan to ensure operations continue during outages or breaches.

Security Considerations	
Third-Party Security	Vet third-party vendors and partners for security compliance. Require third-party integrations to employ secure APIs and best practices.
	Refresh contracts regularly to include security requirements.
User Education	Train users on security best practices, including phishing attack detection and strong password use.
	Give clear instructions on reporting suspicious activity.
Penetration Testing and Audits	Perform regular penetration testing to detect and remediate vulnerabilities.
	Conduct security audits for compliance with internal policy and external regulation.
Secure Infrastructure	Deploy firewalls, intrusion detection/prevention systems (IDS/IPS), and secure network configurations.
	Regularly patch and update software to address known vulnerabilities.

2.

- **PCI-DSS**
  - **Purpose**: Protects payment card information.
  - **Why It Matters**:
    - Avoids breaches through encryption, access controls, and testing.
    - Builds trust; avoids fines and loss of payment functionality.
    - Required for global operations.

- **GDPR**
  - **Purpose**: Protects personal data of EU/EEA residents.
  - **Why It Matters**:
    - Ensures open, secure data processing and user control.
    - Avoids hefty fines (up to 4% of global turnover).
    - Applies globally if processing EU/EEA data.

- **Similarities:** Increases security, builds trust, avoids fines, and enables global reach.

- **Differences**

PCI-DSS	GDPR
Secures payment card data (e.g., credit/debit card numbers)	Protects all personal data (e.g., names, addresses, financial info) of EU/EEA residents.
Applies globally to any organization handling card payments.	Primarily EU/EEA-focused but impacts any global business processing EU/EEA data.
Prevents fraud, builds customer trust, and avoids fines or loss of payment capabilities.	Ensures data privacy, gives users control over their data, and avoids heavy fines (up to 4% of global revenue).

3. What is Idempotency?
The concept of Idempotency entails that performing the same operation multiple times is the same as performing it once. In banking, re-sending a transaction request (incase of retries) will never cause double charges or duplicates. It is crucial for several reasons some of which includes.

- Prevents Duplicates: Avoids double charges or transfers.
- Ensures Consistency: Keeps account balances accurate.
- Improves UX: Users can retry failed transactions safely.
- Simplifies Errors: Handles retries without reprocessing.
- Builds Trust: Ensures reliability and compliance.

4.

- **Risks of Handling Sensitive Customer Data**

  - **Data Breaches:** Unauthorised access to sensitive info (e.g., credit card numbers, personal data).
  - **Financial Fraud:** Stolen data used for identity theft or unauthorised transactions.
  - **Repetitional Damage:** Loss of customer trust and brand credibility.
  - **Regulatory Penalties:** Fines for non-compliance with laws like GDPR or PCI-DSS.
  - **Operational Disruption:** Breaches can cause downtime and revenue loss.

- **Mitigation Strategies**

  - **Encrypt Data:** Use strong encryption (e.g., AES-256) for data in transit (TLS) and at rest (databases).
  - **Access Controls:** Implement Role-Based Access Control (RBAC) to restrict data access.
  - **Security Audits:** Regularly test systems for vulnerabilities and fix weaknesses.
  - **Compliance:** Follow regulations like GDPR and PCI-DSS to avoid fines.
  - **Employee Training:** Teach staff to recognise phishing and handle data securely.
  - **Multi-Factor Authentication (MFA):** Add an extra layer of security for system access.
  - **Monitoring and Logging:** Track data access in real-time to detect suspicious activity.
  - **Data Backups:** Regularly back up data and test recovery processes.
  - **Third-Party Security:** Ensure vendors comply with security standards.
  - **Incident Response Plan:** Prepare and test a plan to respond to breaches quickly.

Section B: Backend Development

1. ACID which when extended is “ Atomicity, Consistency, Isolation, Durability ” are critical for ensuring the reliability and integrity of financial transactions in databases. ACID properties must be ensured for financial applications so that transactions are accurate, dependable, and secure. ACID avoids errors and fosters trust and hence becomes the backbone of database design for financial applications. I’ve drawn a table below to describe how acid properties interact with the database.

ACID PROPERTIES	What It Means	What it does for you
Atomicity	Transactions are all-or-nothing. If one part fails, the whole transaction is rolled back.	Prevents partial updates (e.g., money debited but not credited). Ensures transactions complete fully or not at all.
Consistency	Transactions keep data valid and accurate, following rules (e.g., no negative balances).	Ensures financial data stays correct and reliable after every transaction.
Isolation	Transactions run independently, even when happening at the same time.	Stops issues like double-spending or incorrect reads during high activity (e.g., multiple transfers).
Durability	Once a transaction is done, it's permanent—even if the system crashes.	Guarantees completed transactions (e.g., payments) aren't lost, building trust in the system.

2. Encryption plays a significant role in protecting sensitive information during bank transactions and is an important security feature to safeguard the secure and confidential transmission of important information.

ESSENTIALS OF ENCRYPTION	What It Means	What it does for you
Protects Data in Transit	Encrypts data (like.. account numbers, transaction details) as it travels between the user's device and the bank's servers.	Prevents hackers from intercepting and stealing sensitive information during transmission.
Secures Data at Rest	Encrypts data stored in databases, servers, or backups (e.g., customer details, transaction records).	Ensures that even if hackers gain access to storage systems, they cannot read or misuse the data.
Prevents Unauthorised Access	Ensures only authorized parties (e.g., the bank and the customer) can decrypt and access transaction data.	Reduces the risk of insider threats or external breaches.
Ensures Compliance	Helps banks meet regulatory requirements (e.g., PCI-DSS, GDPR) for data protection.	Avoids fines, legal issues, and reputation damage from non-compliance.
Builds Customer Trust	Demonstrates a commitment to safeguarding customer data.	Customers are more likely to trust and use banks that prioritise security.
Mitigates Fraud	Makes it extremely difficult for attackers to manipulate or misuse transaction	Reduces the risk of fraudulent activities like identity theft or unauthorised transactions.

My Approach

My method of bank transaction security encryption would be to make an in-house package or SDK. It would manage the data and encryption, as well as convert it into the format that is needed at session time. It would also encrypt all responses prior to sending them. This would be helpful to make sure in a case of multiple usage (Micro-services etc..), it would be consistent and reusable.

3. In efforts to implement a secure login system using JWT tokens, my thought process would be as follows:

- **User Logs In:** The user enters their credentials (say, a username and password) to the server, where they're matched against its database or third-party service (Firebase, Supabase, etc..,).
- **JWT is Created:** If credentials are legit, the server generates a JWT. The token has the user data (e.g., ID or username) and is signed with a secret key so that others can't mimic it.
- **Token is Sent Back:** The server sends back the JWT to the client as a response to the login request.
- Client Saves the Token: The client stores this token securely, most often in local storage or cookies.
- **Token is Used for Future Requests:** For each subsequent request, the client sends the JWT in the Authorisation header.
- **Server Verifies the Token:** When the server receives each request, it checks the token's signature and if the token has expired.
- **Access Granted:** The server grants access to the requested resources when the token is valid.

4. To handle simultaneous transactions and prevent double spending or data inconsistency, I'd most likely approach as per the specific need of the system:

	Use Case	**Mechanism:**
Optimistic Concurrency Control (OCC)	Suitable for scenarios with low contention.	Add a version number or timestamp column to the database table.
		When updating a record, verify that the version or timestamp matches the current value in the database.
		If the value has changed, reject the transaction and prompt the user to retry.
Pessimistic Locking	Ideal for high contention scenarios where updates need to be tightly controlled.	Use database row-level locks to prevent other transactions from accessing the same record until the lock is released.
		I could also use the SQL keywords. Example: "SELECT ... FOR UPDATE".
Idempotency for Transactions	For systems that require external requests (e.g., payment APIs).	Generate a unique transaction ID for each operation.
		Store and verify the transaction ID to ensure it is not reused.

	Use Case	Mechanism:
Atomic Operations	For simple, single-step updates.	Use database atomic operations (e.g., UPDATE ... WHERE ...) to ensure that the operation completes as a single unit of work.