Desafío Técnico — DynamoDB Single Table Design + AWS Lambda API

Node.js · TypeScript · AWS SDK v3

Resumen ejecutivo: Este documento describe el diseño completo del sistema: modelado de datos en Amazon DynamoDB utilizando el patrón Single Table Design, y la arquitectura de la API serverless implementada con AWS Lambda y Node.js/TypeScript. La solución resuelve los cuatro access patterns requeridos sin recurrir a Scans, garantizando performance O(1) o O(log n) en todas las operaciones de lectura. La idempotencia, la paginación nativa de DynamoDB y la validación JWT con `aws-jwt-verify` son pilares del diseño.

PARTE 1 — Diseño de DynamoDB

1.1 Nombre de la tabla

Campo	Valor
Nombre de la tabla	AppCore
Billing mode	PAY_PER_REQUEST (On-Demand)
Región sugerida	us-east-1 (o la región del tenant)

Se eligió el nombre `AppCore` porque en Single Table Design una sola tabla aloja todas las entidades del sistema. On-Demand elimina la necesidad de provisionar capacidad y es ideal durante el crecimiento inicial, ya que DynamoDB escala automáticamente sin intervención manual.

1.2 Definición de Primary Key (PK y SK)

Atributo	Tipo	Descripción
pk (Partition Key)	String	Prefijo + ID. Ejemplos: `USER#u-123`, `TX#tx-abc`, `IDE#req-uuid`
sk (Sort Key)	String	Descriptor jerárquico. Ejemplos: `PROFILE`, `TX#<timestamp>#<txId>`, `METADATA`

El uso de prefijos (`USER#`, `TX#`, `NOTIF#`, `IDE#`) cumple dos funciones esenciales en Single Table Design. Primero, previene colisiones de IDs entre entidades distintas: un usuario con id “123” y una transacción con id “123” conviven sin conflicto porque sus partition keys

son USER#123 y TX#123 respectivamente. Segundo, permite distinguir el tipo de ítem con solo leer la PK, lo que simplifica el código de los repositorios.

La Sort Key compuesta TX#<timestamp>#<txId> merece una explicación especial. DynamoDB ordena los ítems dentro de una partición lexicográficamente por SK. Como el timestamp es ISO 8601, su orden lexicográfico coincide exactamente con el orden cronológico (2024-01-15 < 2024-03-20). Esto significa que al hacer Query con ScanIndexForward: false obtenemos las transacciones más recientes primero sin ningún paso de ordenamiento adicional. El txId al final de la SK actúa como desempate cuando dos transacciones ocurren en el mismo milisegundo.

1.3 Global Secondary Index (GSI1)

Nombre	GSI1PK (Partition Key)	GSI1SK (Sort Key)	Proyección	Propósito
GSI1	GSI1PK (String)	GSI1SK = createdAt (ISO 8601)	ALL	Últimas N transacciones globales

El GSI1 es necesario exclusivamente para el access pattern 4 (“últimas 10 transacciones globales del sistema”). Sin un GSI, la única alternativa sería un Scan de tabla completa, lo cual es inaceptable en producción porque su costo y latencia crecen linealmente con el volumen de datos.

La estrategia consiste en que cada ítem de transacción almacena un atributo GSI1PK con el valor fijo "GLOBAL_TX" y un atributo GSI1SK con el timestamp de creación. El GSI los indexa automáticamente. Al hacer Query sobre el GSI con ScanIndexForward: false y Limit: 10, DynamoDB lee solamente los 10 ítems más recientes sin tocar el resto de la tabla.

Consideración de escalabilidad: En sistemas de alto volumen, la partition key fija "GLOBAL_TX" podría convertirse en una hot partition. La solución es el shard pattern: usar "GLOBAL_TX#0" a "GLOBAL_TX#9" y distribuir las transacciones entre shards con txId.hashCode() % 10. Para leer las últimas 10, se hacen 10 queries en paralelo (una por shard) y se fusionan los resultados.

1.4 Access Patterns — mapa completo

#	Descripción	Operación DynamoDB	Índice	PK / SK usados
1	Transacciones de un usuario	Query	Tabla principal	pk=USER#<id> · sk begins_with TX#
2	Transacción por ID	GetItem	Tabla principal	pk=TX#<txId> · sk=METADATA

#	Descripción	Operación DynamoDB	Índice	PK / SK usados
3	Notificaciones de usuario ordenadas por fecha	Query	Tabla principal	`pk=USER#<id>` · `sk begins_with NOTIF#`
4	Últimas 10 transacciones globales	Query	GSI1	`GSI1PK=GLOBAL_TX` · `GSI1SK=createdAt (desc)`

1.5 Ejemplos de ítems almacenados

Ítem 1 — Entidad Usuario

```
{
  "pk": "USER#u-123",
  "sk": "PROFILE",
  "entityType": "USER",
  "userId": "u-123",
  "email": "ana.garcia@mail.com",
  "name": "Ana García",
  "createdAt": "2024-01-15T10:00:00.000Z"
}
```

Ítem 2 — Transacción bajo el usuario (access pattern 1)

Este ítem vive en la misma partición que el usuario. La SK compuesta garantiza orden cronológico nativo.

```
{
  "pk": "USER#u-123",
  "sk": "TX#2024-01-15T10:05:00.000Z#tx-abc",
  "entityType": "TRANSACTION",
  "txId": "tx-abc",
  "userId": "u-123",
  "amount": 500,
  "currency": "ARS",
  "status": "PENDING",
  "createdAt": "2024-01-15T10:05:00.000Z",
  "GSI1PK": "GLOBAL_TX",
  "GSI1SK": "2024-01-15T10:05:00.000Z"
}
```

Los atributos `GSI1PK` y `GSI1SK` habilitan el access pattern 4 sin operaciones extra.

Ítem 3 — Lookup de transacción por ID (access pattern 2)

Este ítem permite un GetItem O(1) directamente por `txId`, sin necesidad de conocer el `userId`.

```
{
  "pk": "TX#tx-abc",
  "sk": "METADATA",
  "entityType": "TRANSACTION",
  "txId": "tx-abc",
  "userId": "u-123",
  "amount": 500,
  "currency": "ARS",
  "status": "COMPLETED",
  "createdAt": "2024-01-15T10:05:00.000Z",
  "GSI1PK": "GLOBAL_TX",
  "GSI1SK": "2024-01-15T10:05:00.000Z"
}
```

Ítem 4 — Notificación del usuario (access pattern 3)

La SK sigue el mismo patrón de timestamp que las transacciones, garantizando orden cronológico descendente con `ScanIndexForward: false`.

```
{
  "pk": "USER#u-123",
  "sk": "NOTIF#2024-01-15T10:06:00.000Z#notif-xyz",
  "entityType": "NOTIFICATION",
  "notifId": "notif-xyz",
  "userId": "u-123",
  "message": "Tu transacción TX#tx-abc fue aprobada.",
  "read": false,
  "createdAt": "2024-01-15T10:06:00.000Z"
}
```

Ítem 5 — Control de idempotencia

Este ítem se escribe junto con la transacción en una sola operación atómica (`TransactWrite`). Si el mismo `Idempotency-Key` llega dos veces, la `ConditionExpression: attribute_not_exists(pk)` rechaza el segundo Put y se devuelve la transacción original.

```
{
  "pk": "IDE#req-uuid-xxxx",
  "sk": "METADATA",
  "entityType": "IDEMPOTENCY",
  "txId": "tx-abc",
  "userId": "u-123",
  "createdAt": "2024-01-15T10:05:00.000Z"
}
```

1.6 Justificación de decisiones de diseño

¿Por qué Single Table Design? En DynamoDB, cada operación que cruza particiones requiere múltiples round-trips a la red. Con Single Table Design, una operación que en SQL requeriría un JOIN puede ejecutarse como una sola Query si los datos están en la misma partición. Esto reduce latencia y simplifica el código de acceso a datos.

¿Por qué TransactWrite para crear transacciones? Al crear una transacción se necesita escribir dos ítems atómicamente: el ítem bajo `USER#` (para el access pattern 1) y el ítem lookup `TX#` (para el access pattern 2). Si se usaran dos PutItem separados y la Lambda fallara entre ellos, el sistema quedaría en un estado inconsistente. `TransactWrite` garantiza que ambas escrituras ocurren juntas o ninguna.

¿Por qué incluir GSI1PK y GSI1SK en ambos ítems de transacción? DynamoDB proyecta los ítems de la tabla base al GSI en el momento de escritura. Si el ítem lookup `TX#` no tuviera `GSI1PK`, no aparecería en el GSI y el access pattern 4 mostraría información incompleta. Proyectar en ambos garantiza que cualquier lectura del GSI devuelva ítems con todos los datos necesarios.

PARTE 2 — Diseño de la API en AWS Lambda

2.1 Estructura del proyecto

```
dynamodb-lambda/
├── src/
│   ├── handlers/              ← Lambda entry points (1 archivo = 1 func
ión)
│   │   ├── transactionsGet.ts
│   │   ├── transactionGetById.ts
│   │   ├── transactionsPost.ts
│   │   ├── transactionsGlobal.ts   ← AP4: últimas 10 del sistema
│   │   └── notificationsGet.ts
│   ├── services/              ← Lógica de negocio y acceso a DynamoDB
│   │   ├── auth.ts            ← Validación JWT con aws-jwt-verify
│   │   ├── dynamoClient.ts    ← Cliente DynamoDB compartido
│   │   ├── transactions.ts    ← Repositorio de transacciones
│   │   └── notifications.ts   ← Repositorio de notificaciones
│   ├── types/
│   │   └── index.ts           ← Tipos TypeScript compartidos (TxItem, e
tc.)
│   └── utils/
│       └── errors.ts          ← Jerarquía de errores HTTP tipados
├── test/
│   ├── transactions.test.ts
│   └── transactionsPost.test.ts
└── docs/                      ← Este documento
```

La separación entre `handlers` y `services` es fundamental. Los handlers son responsables únicamente del protocolo HTTP: leer headers, parsear body, retornar status codes y manejar errores. Los services contienen la lógica de negocio real y las queries a DynamoDB. Esta separación hace que los services sean 100% testeables sin invocar una Lambda real, como demuestran los tests con Vitest que mockean `ddb.send` directamente.

Tipos compartidos: El directorio `types/` centraliza las definiciones TypeScript que se comparten entre capas. El tipo `TxItem` modela cualquier ítem de la tabla con sus atributos mínimos garantizados (`pk`, `sk`, `entityType`) y un índice dinámico `[k: string]: any` que permite almacenar atributos adicionales sin perder type-safety en los campos conocidos. El union type `'TRANSACTION' | 'USER' | 'NOTIFICATION'` en `entityType` actúa como discriminador en runtime, el mismo rol que cumplen los prefijos en las partition keys: identificar el tipo de entidad sin ambigüedad.

2.2 Endpoints y organización de Lambdas

Método	Endpoint	Handler	Descripción
GET	/transactions	handlers/transactionsGet.ts	Transacciones del usuario autenticado (paginadas)
GET	/transactions/{id}	handlers/transactionGetById.ts	Transacción por ID con validación de ownership
GET	/transactions/global	handlers/transactionsGlobal.ts	Últimas 10 transacciones del sistema (GSI1)
POST	/transactions	handlers/transactionsPost.ts	Crear transacción con idempotencia
GET	/notifications	handlers/notificationsGet.ts	Notificaciones del usuario, orden descendente

Cada endpoint corresponde a una Lambda individual (arquitectura one function per endpoint). Esto permite que AWS escale cada función de forma independiente: si `GET /transactions` recibe 1000 req/s pero `POST /transactions` solo recibe 10, AWS despliega 1000 instancias de la primera y solo 10 de la segunda. Con una Lambda monolítica, toda la función escalaría igual, desperdiciando recursos.

2.3 Validación del JWT de Cognito

La función `validateToken` en `services/auth.ts` utiliza la librería oficial `aws-jwt-verify` publicada por el equipo de AWS. Esta librería realiza automáticamente las siguientes verificaciones al llamar a `verifier.verify(token)`:

Verifica la firma digital del token contra las claves públicas del User Pool (JWKS endpoint de Cognito), garantizando que el token no fue falsificado. Valida que el campo `tokenUse` sea `"access"` y no `"id"`, evitando que se use el token de identidad como token de acceso. Comprueba que el `issuer` coincida exactamente con el URL del User Pool configurado en `COGNITO_USER_POOL_ID`. Verifica que el token no haya expirado inspeccionando el claim `exp`, sin depender del reloj del cliente. Valida que el `clientId` del token corresponda al valor configurado en `COGNITO_CLIENT_ID`.

Las claves JWKS se cachean automáticamente por la librería, por lo que no se realiza una llamada HTTP a Cognito en cada request. Esto es crítico en Lambda porque las llamadas externas en el hot path aumentan la latencia considerablemente. Una vez que la instancia Lambda está "caliente", la verificación es completamente local.

El `userId` (claim `sub` del token) se extrae del payload verificado y se usa para construir las partition keys de DynamoDB (`USER#<sub>`). De esta forma, cada usuario solo puede acceder a sus propios datos porque el `userId` viene del token, no del body del request.

2.4 Manejo de errores

El manejo de errores se apoya en una jerarquía de clases tipadas definida en `utils/errors.ts`. La clase base `HttpError` extiende `Error` agregando un campo `statusCode`, y subclases como `UnauthorizedError` (401), `ForbiddenError` (403), `NotFoundError` (404) y `BadRequestError` (400) proveen mensajes por defecto semánticos. Esto elimina los "magic numbers" de status code dispersos por el código: en lugar de escribir `return { statusCode: 401 }` manualmente en cada handler, cualquier capa de la aplicación puede lanzar `throw new UnauthorizedError()` y el bloque `catch` del handler lo convierte automáticamente en la respuesta HTTP correcta a través de `err.statusCode || 500`.

```
// utils/errors.ts
export class HttpError extends Error {
  statusCode: number;
  constructor(statusCode: number, message: string) {
    super(message);
    this.name = this.constructor.name; // facilita identificación en logs
    this.statusCode = statusCode;
  }
}

export class UnauthorizedError extends HttpError {
  constructor(message = 'Unauthorized') { super(401, message); }
}

export class ForbiddenError extends HttpError {
  constructor(message = 'Forbidden') { super(403, message); }
}

export class NotFoundError extends HttpError {
```

```
  constructor(message = 'Not found') { super(404, message); }
}

export class BadRequestError extends HttpError {
  constructor(message = 'Bad request') { super(400, message); }
}
```

La tabla siguiente resume todos los códigos de error que el sistema puede devolver y su causa:

HTTP Code	Causa	Ejemplo
401	Token ausente o inválido	Authorization header no enviado
403	Recurso de otro usuario	GET /transactions/{id} de otra persona
404	Recurso no encontrado	txId inexistente
400	Body inválido	JSON malformado o falta el campo `amount`
409	Idempotency conflict	Mismo Idempotency-Key procesado dos veces
500	Error interno inesperado	Falla de DynamoDB o bug no controlado

Consistencia entre la jerarquía de errores y los handlers: Los handlers utilizan la jerarquía `HttpError` de dos formas complementarias. Para errores de negocio que se detectan dentro de los services (por ejemplo, un recurso no encontrado o un token inválido), cualquier capa puede lanzar `throw new NotFoundError()` o `throw new UnauthorizedError()` y el bloque `catch` del handler lo convierte automáticamente en la respuesta HTTP correcta mediante `err.statusCode || 500`, sin necesidad de conocer el código numérico en el lugar donde se lanza el error. Para validaciones de entrada que ocurren directamente en el handler antes de llamar al service (como ausencia del `Authorization` header o JSON malformado), se lanza también el error tipado correspondiente para que el mismo `catch` central lo resuelva de forma uniforme. Esta estrategia garantiza que el bloque `catch` de cada handler sea idéntico y no cambie entre funciones: toda la lógica de "qué status code corresponde" vive en la jerarquía de errores, no dispersa en condicionales.

Un detalle importante de seguridad está en `transactionGetById.ts`: después de obtener el ítem de DynamoDB, se verifica que `item.userId === userId` antes de devolverlo. Esto previene que un usuario autenticado acceda a las transacciones de otro usuario, incluso si conoce el `txId`. Sin esta verificación, el sistema sería vulnerable a un ataque de enumeración de IDs (IDOR — Insecure Direct Object Reference).

2.5 Estrategia de testing

El proyecto utiliza Vitest como framework de testing, con una arquitectura de mocks que refleja la separación entre capas del código. Los tests de handlers mockean tanto `auth` como `services`, permitiendo verificar el comportamiento HTTP de forma aislada sin tocar DynamoDB ni Cognito. Los tests de services mockean directamente `ddb.send`, lo que permite simular secuencias de respuestas de DynamoDB con precisión quirúrgica.

El test de idempotencia en `transactions.test.ts` es un ejemplo representativo de esta estrategia. Mockea una secuencia de tres llamadas consecutivas a `ddb.send`: primero el `TransactWrite` que lanza `ConditionalCheckFailedException`, luego el `GetCommand` que devuelve el ítem de idempotencia con el `txId` original, y finalmente el `GetCommand` que recupera la transacción existente. Esto valida el flujo de recuperación completo ante un reintento, que es precisamente el camino más difícil de testear y el más crítico para la corrección del sistema.

Los casos cubiertos incluyen: creación exitosa de una transacción (201), token ausente (401), JSON inválido en el body (400), error inesperado del servicio (500) y recuperación idempotente ante un reintento con la misma clave.

2.6 Estrategia de escalabilidad

AWS Lambda — concurrencia. Lambda escala horizontalmente de forma automática. El límite por defecto es 1000 instancias simultáneas por región, ampliable bajo pedido. Las funciones críticas como `POST /transactions` pueden tener Reserved Concurrency para garantizarles capacidad en picos, sin que funciones de menor prioridad los consuman.

DynamoDB — particiones y hot keys. Con el diseño actual, cada usuario tiene su propia partición (`USER#<id>`), lo que garantiza una distribución natural de la carga. La única partition key compartida es `GLOBAL_TX` en el GSI1. Para volúmenes muy altos (más de 3000 RCU/s sobre esa partición), se aplicaría el shard pattern mencionado en la sección 1.3.

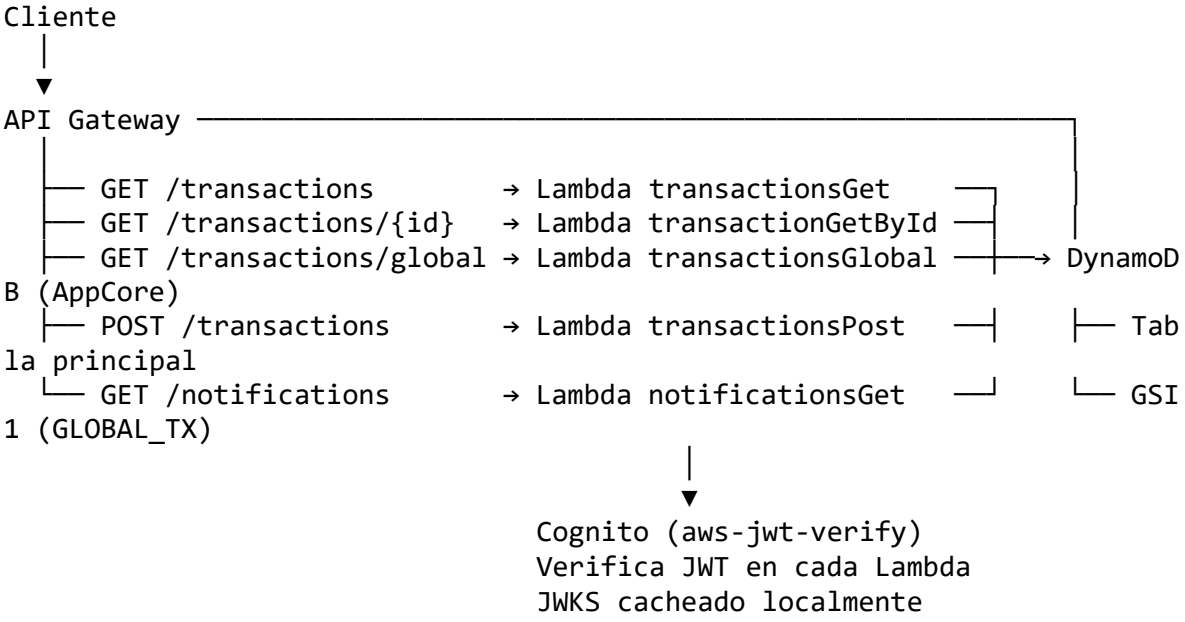
API Gateway — throttling. API Gateway soporta hasta 10.000 req/s por defecto. Se configuran Rate Limiting y Burst Limit a nivel de stage y por API key para proteger el sistema de picos abusivos. Los endpoints de solo lectura como `GET /transactions/global` pueden cachearse en API Gateway y reducir el número de Lambdas invocadas a cero durante el TTL configurado.

Paginación nativa. Todos los endpoints de listado implementan paginación basada en `LastEvaluatedKey` de DynamoDB. Esto garantiza que ninguna Lambda necesita cargar en memoria listas de longitud arbitraria, manteniendo el consumo de memoria predecible independientemente del volumen de datos. El cliente recibe el `lastKey` en la respuesta y lo envía como query parameter en la siguiente request.

Idempotencia — seguridad ante reintentos. Los clientes móviles suelen reintentar requests fallidas. Sin idempotencia, un reintento de `POST /transactions` podría crear la

misma transacción dos veces. El `Idempotency-Key` header junto con el ítem `IDE#` en DynamoDB y la `ConditionExpression attribute_not_exists` garantizan que la segunda request devuelva el mismo resultado que la primera sin crear un duplicado, incluso bajo concurrencia.

2.7 Diagrama de arquitectura



Todas las decisiones de diseño de este sistema priorizan el rendimiento en producción: queries O(1) o O(log n), escrituras atómicas con `TransactWrite`, jerarquía de errores tipada con `HttpError`, escalado automático sin configuración manual, y seguridad basada en tokens verificados criptográficamente por `aws-jwt-verify`.