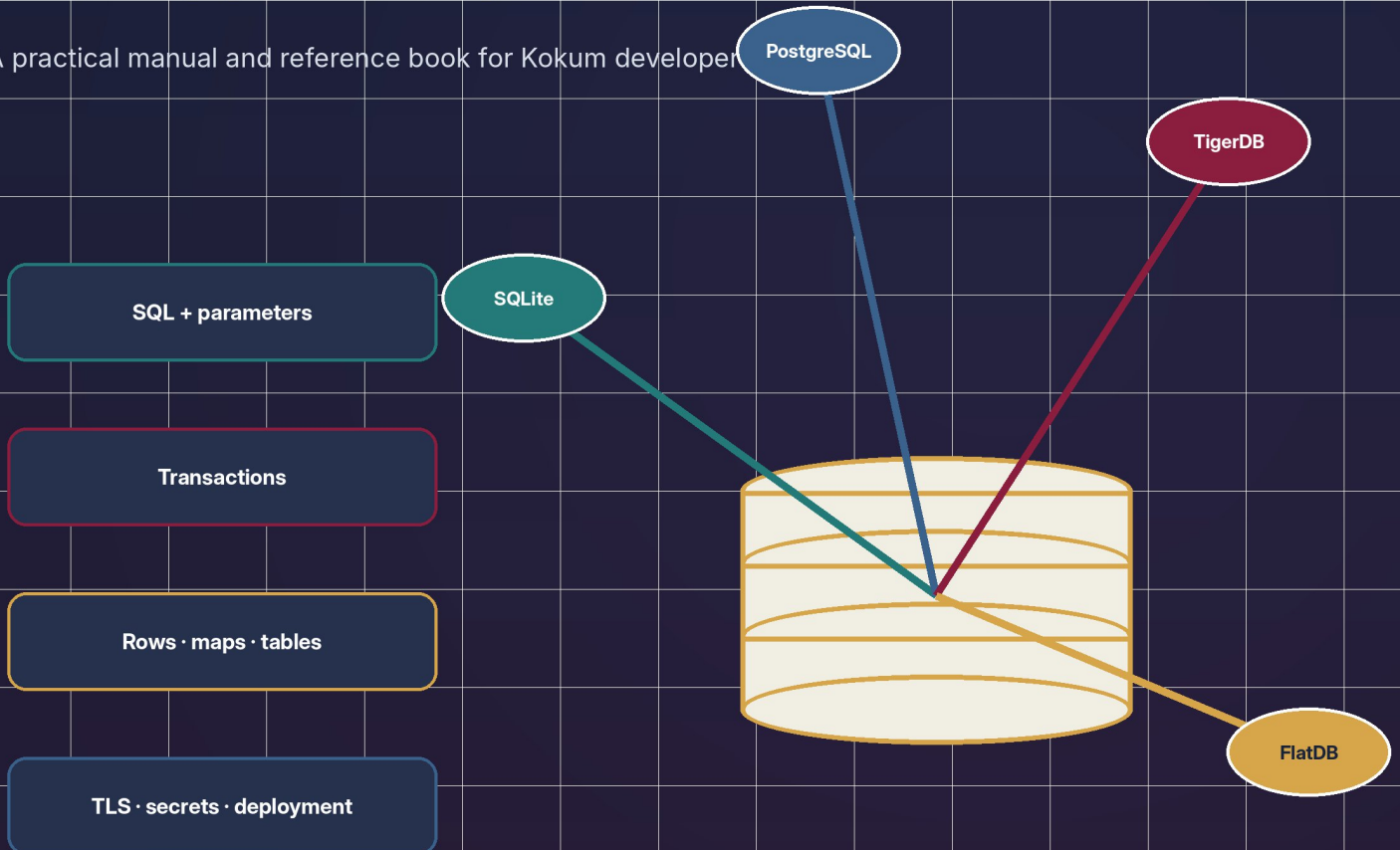


DATABASE PROGRAMMING

Connectivity, SQL Queries and Result Sets

A practical manual and reference book for Kokum developer



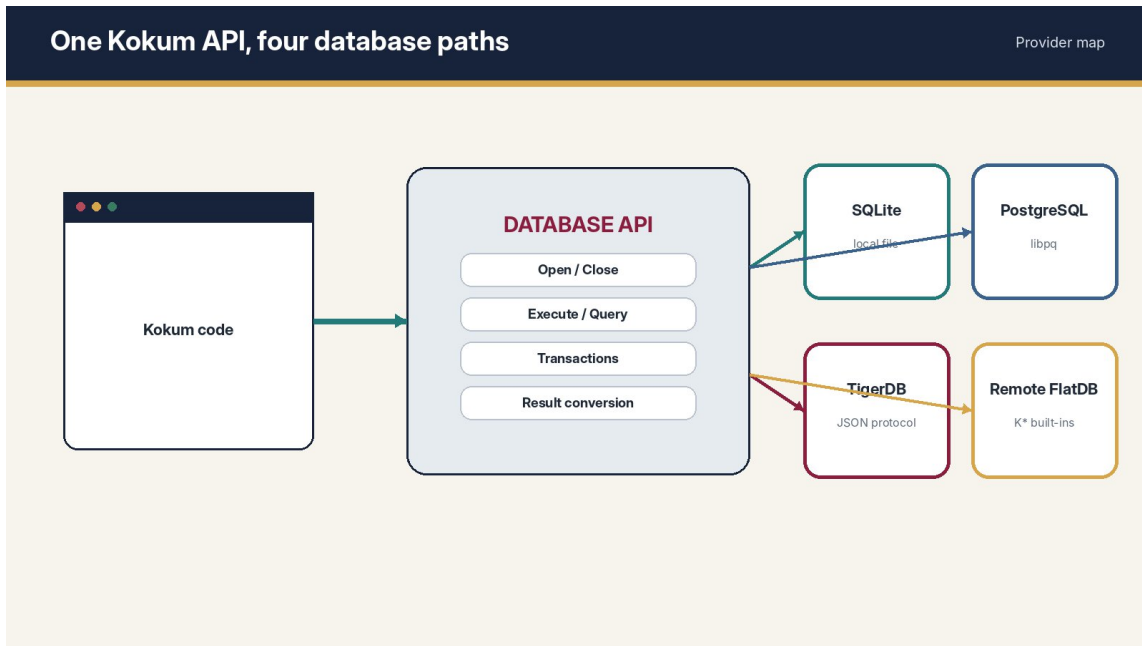
Kokum 0.29.0 · Phase 4.16.7

Database ABI 1.3 · SQLite · PostgreSQL · TigerDB · Remote FlatDB

TigerDB Solutions Private Limited

Kokum Database Programming

Connectivity, SQL Queries and Result Sets



Edition	Kokum 0.29.0 - Phase 4.16.7
Scope	SQLite, PostgreSQL, TigerDB and Remote FlatDB
Publisher	TigerDB Solutions Private Limited

A practical manual, tutorial and desk reference

EDITION NOTE

About this book

This volume is the database-programming companion to the Kokum Core Language Programming Manual. It concentrates on connection configuration, SQL execution, parameter binding, transactions, result-set handling, provider-specific capabilities, and production deployment. It deliberately avoids visual form design; database components are discussed only as nonvisual application resources.

VERSION BOUNDARY

The examples and contracts in this book describe Kokum 0.29.0 through Phase 4.16.7. The public DATABASE ABI is 1.3. Always check release notes before applying provider-specific details to a later version.

Copyright and responsibility

Copyright 2026 TigerDB Solutions Private Limited. Kokum and TigerDB are products of TigerDB Solutions Private Limited. Database administrators remain responsible for server configuration, backups, access control, certificates, schema change review, and legal or regulatory requirements that apply to their data.

Typographic conventions

Table 1. Conventions used throughout the book

Convention	Meaning
Monospace listing	Kokum source, SQL, configuration, paths or shell commands.
DATABASE	The provider-neutral Kokum native resource type.
dbLocal / dbPostgres / dbTiger	Example component identifiers generated into a slots module.
? placeholder	A value position supplied through a separate Kokum ARRAY.
NULL	No value; distinct from zero, .F. and an empty string.

Source of truth

The book was prepared from the Phase 4.16.7 source tree, public headers, provider documentation, example projects and executable regression suites. Where documentation and implementation differed, the current implementation and tests were treated as authoritative.

PREFACE

Why a separate database book?

Database programming looks simple when reduced to a connection string and a SELECT statement. Real applications must also decide where secrets live, how values are bound safely, what shape a query should return, when a transaction begins and ends, how provider-specific types become language values, and how connections behave across tasks, service workers and deployments.

Kokum intentionally presents a compact common API for SQLite, PostgreSQL and TigerDB. Remote FlatDB has a smaller direct API designed for authenticated, portable flat-file services. The common surface makes ordinary code easy to move, while explicit provider chapters keep important differences visible rather than pretending every database behaves identically.

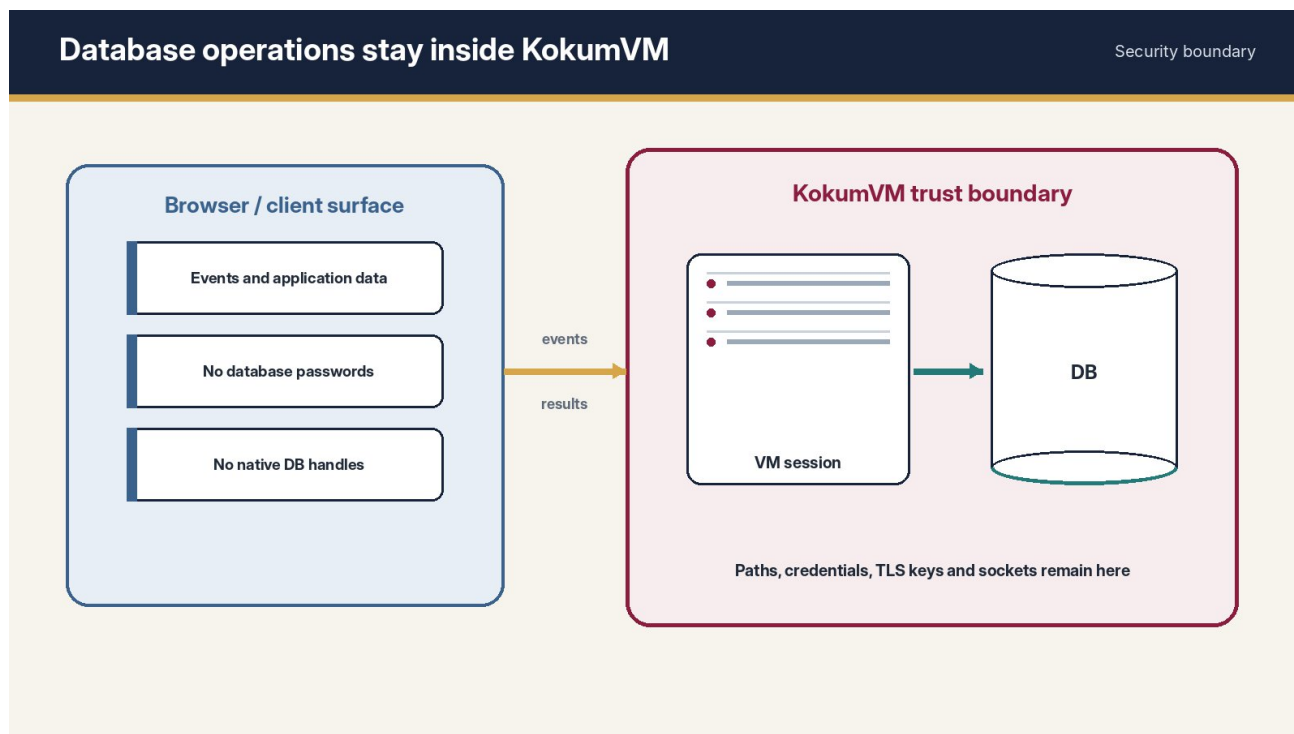


Figure 1. Database credentials and native handles remain inside the KokumVM trust boundary

What you will be able to do

- Configure nonvisual database resources without embedding production secrets in a bundle.
- Execute parameterized SQL and select the correct result shape for each use case.
- Process arrays of row maps, single rows, scalars and two-dimensional tables safely.
- Design reliable transactions, error handling, retries and deterministic cleanup.
- Use SQLite, PostgreSQL, TigerDB and Remote FlatDB with their real operational differences.
- Build repository modules, asynchronous database tasks and database-backed microservices.

HOW TO USE THIS BOOK

Reading paths and verification marks

Readers new to Kokum database programming should complete Parts I and II in order, then select the server provider they use. Experienced developers can treat the appendices as a desk reference and jump directly to a provider chapter.

Table 2. Verification legend

Mark	What was verified
EXECUTED	The example ran through a real provider path, bundle or local server in the Phase 4.16.7 tree.
PROTOCOL TESTED	The provider adapter, framing, parameter conversion, TLS or deterministic mock-server behavior passed its automated suite.
PROJECT CHECKED	The exact project or module passed semantic and project checks and, where stated, bundle verification.
CONFIG VALIDATED	The configuration was generated or consumed by the compiler/provider tests.

IMPORTANT

A verified example demonstrates language and provider behavior; it does not replace a staging test against your schema, server version, collation, permissions, network and production data volume.

A practical study sequence

1. Run the SQLite customer examples locally to learn the common API without a server.
2. Repeat the same operations against PostgreSQL or TigerDB and record any SQL dialect differences.
3. Add explicit transaction tests, failure tests and result-shape assertions.
4. Move SQL into repository modules before exposing it through an HTTP service.
5. Deploy with external configuration, least-privilege credentials and a tested backup/restore plan.

CONTENTS

Table of contents

Part 1: Common database foundations	7
Chapter 1: The Kokum database model	8
Chapter 2: Database components and project artifacts	9
Chapter 3: External configuration and secrets	11
Chapter 4: Connection lifecycle and health	12
Chapter 5: SQL execution and parameter binding	14
Chapter 6: Choosing a result-set method	16
Chapter 7: Working with rows, maps and tables	18
Chapter 8: NULL and provider type conversion	19
Chapter 9: Affected rows and generated identifiers	20
Chapter 10: Transactions and deterministic cleanup	21
Part 2: SQLite: embedded relational storage	23
Chapter 11: SQLite configuration and open modes	24
Chapter 12: Schema creation, initialization and migrations	26
Chapter 13: SQLite CRUD in practice	27
Chapter 14: Joins, aggregates, sorting and pagination	28
Chapter 15: SQLite values, dates, decimals and BLOB results	29
Chapter 16: WAL, contention and deployment	30
Part 3: PostgreSQL: shared transactional services	32
Chapter 17: PostgreSQL build and connection configuration	33
Chapter 18: TLS, schema identity and read-only sessions	34
Chapter 19: Portable parameters, casts and placeholder scanning	36
Chapter 20: PostgreSQL result types and JSONB	38
Chapter 21: RETURNING, transactions and diagnostics	40
Part 4: TigerDB: server and router connectivity	41
Chapter 22: TigerDB server and router architecture	42
Chapter 23: Secure TigerDB connection configuration	44
Chapter 24: TigerDB SQL and result sets	45
Chapter 25: Ping, UseDatabase, Raw and runtime diagnostics	46
Chapter 26: TLS, mTLS, timeouts and retry boundaries	47
Part 5: Remote FlatDB: authenticated portable storage	48
Chapter 27: Remote FlatDB server, key authentication and kdb.conf	49
Chapter 28: Remote FlatDB CRUD and result rows	51
Chapter 29: Path resolution, concurrency and persistence	52
Part 6: Advanced application architecture	54
Chapter 30: Creating tables from JSON	55
Chapter 31: Repository modules and reusable data access	57
Chapter 32: Async tasks, threads and connection ownership	59
Chapter 33: Database-backed HTTP microservices	61
Chapter 34: Testing, observability, security and deployment	63

Reference appendices	65
Appendix A: DATABASE API quick reference	65
Appendix B: Configuration key reference	66
Appendix C: Result and type mapping reference	68
Appendix D: Remote FlatDB direct API	69
Appendix E: Troubleshooting guide	70
Appendix F: Verification record for this edition	71
Closing note: Build the data boundary deliberately	72

PART 1

Common database foundations

The first part establishes the provider-neutral mental model. The same DATABASE value can expose SQLite, PostgreSQL or TigerDB while preserving a stable set of lifecycle, execution, query and transaction operations.

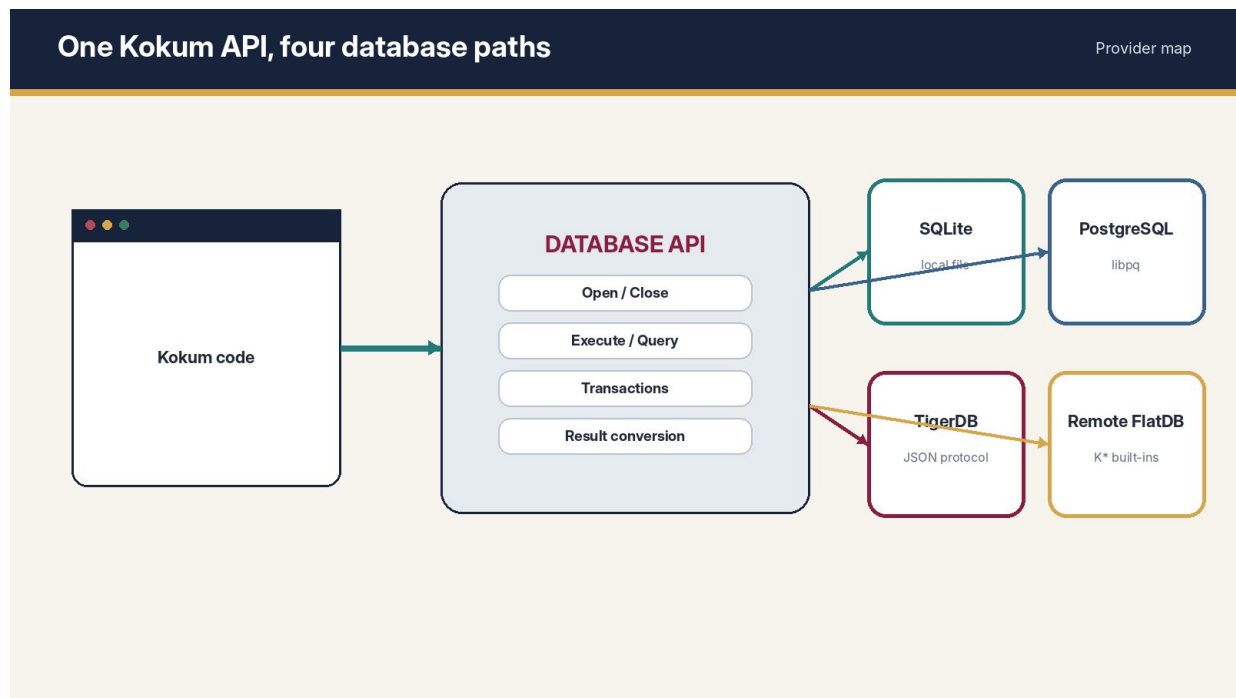


Figure 2. Part 1 overview: Common database foundations

Chapters in this part

- 1. The Kokum database model
- 2. Database components and project artifacts
- 3. External configuration and secrets
- 4. Connection lifecycle and health
- 5. SQL execution and parameter binding
- 6. Choosing a result-set method
- 7. Working with rows, maps and tables
- 8. NULL and provider type conversion
- 9. Affected rows and generated identifiers
- 10. Transactions and deterministic cleanup

CHAPTER 1

The Kokum database model

One language surface, several execution paths

LEARNING OBJECTIVES

identify the four supported database paths; separate common behavior from provider-specific behavior; inspect a DATABASE resource safely.

Kokum 0.29.0 has two related database programming surfaces. SQLite, PostgreSQL and TigerDB use the native DATABASE resource and its provider-neutral methods. Remote FlatDB uses KCONNECT, KEXECUTE, KQUERY and KDISCONNECT because its authentication, storage and deployment model is intentionally smaller and independent.

A DATABASE variable is not a connection string and not a map. It is an owner-affine native resource whose provider configuration is produced by the project. The provider is selected outside ordinary business logic, allowing service and repository functions to accept DATABASE parameters without knowing how the connection is implemented.

Table 3. Database paths in Kokum 0.29.0

Path	Typical use	Primary API
SQLite	Embedded application data, local cache, single-node service	DATABASE
PostgreSQL	Transactional server applications and shared relational data	DATABASE
TigerDB	TigerDB server or router, including TLS and mTLS	DATABASE
Remote FlatDB	Authenticated remote single-file database service	KCONNECT / KQUERY

Listing 1. Inspect a provider-neutral DATABASE resource

```
PROCEDURE DescribeDatabase(db AS DATABASE)
  ? "Id: " + db.Id
  ? "Provider: " + db.Provider
  ? "Open: " + STR(db.IsOpen)
ENDPROC
```

CHECKED - common DATABASE properties validated by the API suite

DESIGN RULE

Write ordinary CRUD and result processing against the common methods. Isolate dialect-specific SQL and provider-only methods in small repository functions.

Chapter summary

Kokum offers a shared DATABASE API for three providers and a direct API for Remote FlatDB. The common surface improves reuse, but provider differences remain explicit and testable.

Practice and review

- List the database path best suited to a desktop cache, a shared accounting system and a small authenticated remote file service.
- Write a procedure that logs Id, Provider and IsOpen without opening the connection.

CHAPTER 2

Database components and project artifacts

How a nonvisual database resource becomes usable source

LEARNING OBJECTIVES

understand generated DATABASE declarations; recognize build artifacts; avoid editing managed declaration blocks.

In a Kokum application project, a database component is nonvisual and application-scoped. Its stable ID becomes a PUBLIC DATABASE declaration in the configured module. This book does not teach form layout; the relevant point is that the project owns the provider declaration while your source owns the operations performed through it.

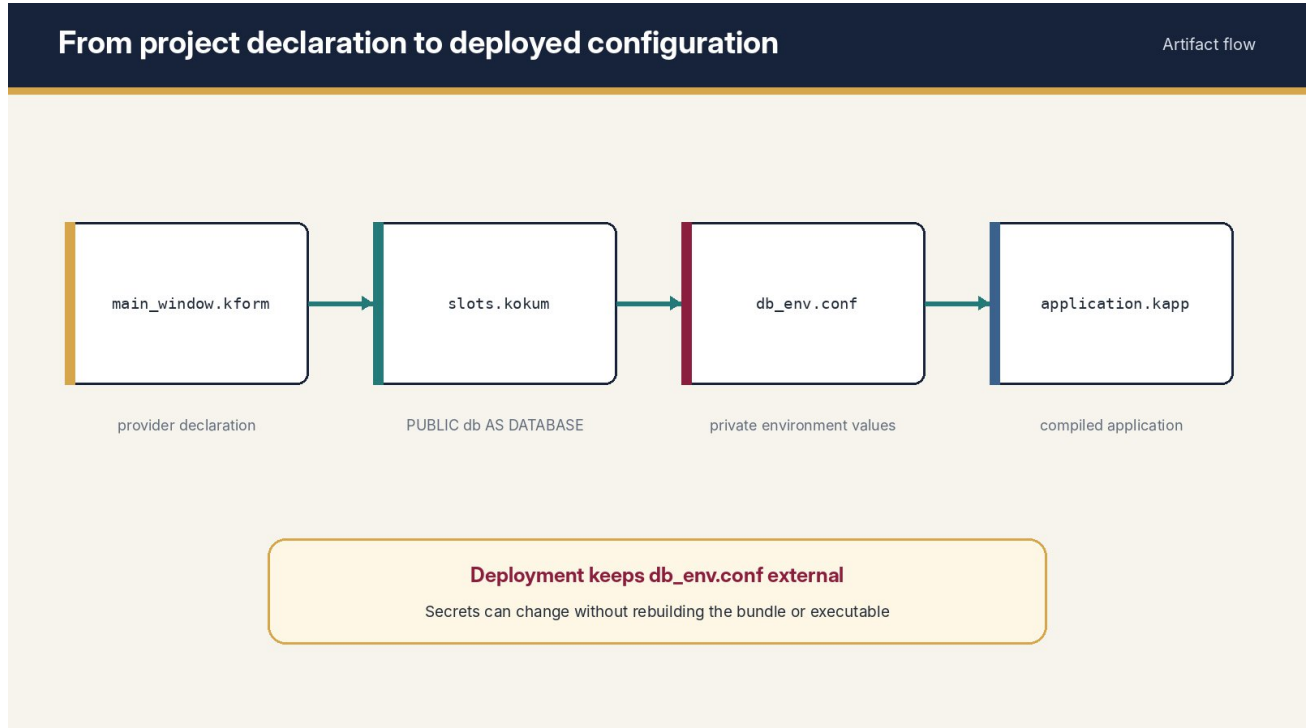


Figure 3. Database component declarations, source variables, private configuration and compiled applications are separate artifacts

The compiler may regenerate the managed block during project check or build. Programmer-written code belongs outside the markers. Renaming the component changes the generated variable name and should be treated as an API change within the application.

Listing 2. Generated database declarations in a slots module

```

&& <KOKUM-AUTO-DATABASE-COMPONENTS-BEGIN>
&& Generated from Database components. Do not edit this block.
PUBLIC dbLocal AS DATABASE
PUBLIC dbPostgres AS DATABASE
PUBLIC dbTiger AS DATABASE
&& <KOKUM-AUTO-DATABASE-COMPONENTS-END>
  
```

PROJECT CHECKED - managed declarations are regenerated from project metadata

Table 4. Project and deployment artifacts

Artifact	Purpose	Deployment status
.kform database declaration	Stable component identity and provider property defaults	Compiled into project resources
slots.kokum declaration	Source-visible DATABASE variable	Compiled
build/database/components.conf	Generated provider inventory	Build artifact
db_env.conf	Environment-specific values and secret references	External private file
.kapp or standalone executable	Code and provider declarations	Deployable application

Chapter summary

Database components are nonvisual resources. Generated declarations provide type-safe access, while environment-specific values remain outside the application bundle.

Practice and review

8. Identify which artifacts belong in source control and which must remain private.
9. Explain why editing the managed PUBLIC block directly is unsafe.

CHAPTER 3

External configuration and secrets

Keep deployment values outside compiled code

LEARNING OBJECTIVES

use db_env.conf lookup rules; apply supported substitutions; protect credentials and certificates.

Kokum keeps database environment values in db_env.conf beside the project, bundle or deployed executable. This allows the same compiled application to move between development, staging and production without recompilation. The compiler also creates db_env.example.conf as a shareable template.

The runtime first checks KOKUM_DB_ENV, then the legacy TLANG_DB_ENV, and finally looks for db_env.conf beside the active bundle or executable. Substitution is deliberately limited to APP_DIR, PROJECT_DIR, USER_DATA_DIR and ENV references; the file is not a shell script.

Listing 3. Production-style external PostgreSQL configuration

```
# db_env.conf - keep private and mode 0600 on Linux
[database:dbPostgres]
provider=postgresql
module=company.orders.data
host=db.internal.example
port=5432
database=orders
username=kokum_orders
password=${ENV:KOKUM_DB_PASSWORD}
sslmode=verify-full
sslrootcert=${APP_DIR}/tls/ca.crt
auto_open=false
```

CONFIG VALIDATED - generated configuration schema and environment substitution tested

SECURITY BOUNDARY

Do not place real passwords in source, .kform files, example configuration or application bundles. Prefer environment references or an operating-system secret injection mechanism. Restrict db_env.conf and private keys to the application account.

Table 5. Supported configuration substitutions

Substitution	Meaning
\${APP_DIR}	Directory containing the active bundle or executable.
\${PROJECT_DIR}	Project root during development and build workflows.
\${USER_DATA_DIR}	Platform-appropriate user data directory.
\${ENV:NAME}	Value of environment variable NAME.

Chapter summary

External configuration makes deployments portable and keeps secrets outside code. Lookup and substitution are intentionally deterministic and non-shell-like.

Practice and review

- 10. Create a db_env.conf that obtains its password from an environment variable.
- 11. Explain why a relative certificate path should normally use APP_DIR.

CHAPTER 4

Connection lifecycle and health

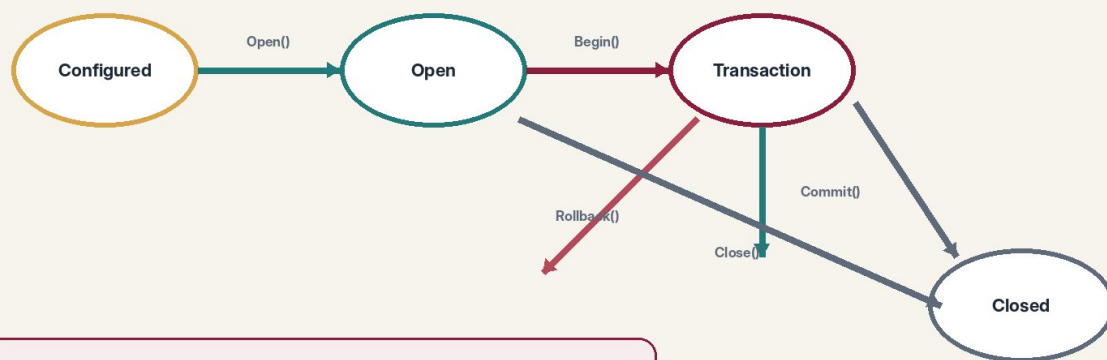
Open late, close deterministically, observe state

LEARNING OBJECTIVES

open and close a connection; use IsOpen and Ping correctly; guarantee cleanup with FINALLY.

Connection lifecycle and deterministic cleanup

State machine



Close() rolls back an unfinished transaction

Use TRY / CATCH / FINALLY for every manually opened connection

Figure 4. DATABASE lifecycle including the implicit rollback performed during close

A component with `auto_open=true` opens when its owning application session is initialized. With `auto_open=false`, call `Open` explicitly. `Open` returns logical true on success and raises a catchable error on failure. `Close` releases the provider resource; closing a connection with an unfinished transaction rolls that transaction back.

`IsOpen` answers whether the resource currently owns an open provider connection. PostgreSQL and TigerDB additionally implement `Ping`. A successful TCP connection is not the same as a successful authenticated database session, so service readiness checks should exercise the provider rather than merely inspect a socket.

Listing 4. Open, test and close a database deterministically

```

PROCEDURE CheckConnection(db AS DATABASE)
  TRY
    db.Open()
    ? "Connected through " + db.Provider
    IF db.Provider != "sqlite"
      ? JSONENCODE(db.Ping())
    ENDIF
  CATCH error
    ? "Database unavailable: " + STR(error)
  FINALLY
    IF db.IsOpen
      db.Close()
    ENDIF
  ENDTRY
ENDPROC

```

CHECKED - lifecycle syntax and provider operations passed semantic/API tests

SERVICE LIFETIME

Long-running services usually keep one connection per persistent worker session rather than opening on every request. Short utilities may prefer a narrow TRY/FINALLY scope.

Chapter summary

Use explicit lifecycle control when startup ordering, error reporting or shutdown matters. IsOpen reports resource state; Ping validates a remote provider session.

Practice and review

12. Modify the example to keep the connection open only when Ping succeeds.
13. Describe what happens if Close is called during an active transaction.

CHAPTER 5

SQL execution and parameter binding

Separate statement structure from values

LEARNING OBJECTIVES

execute parameterized SQL; understand provider binding strategies; avoid SQL injection through concatenation.

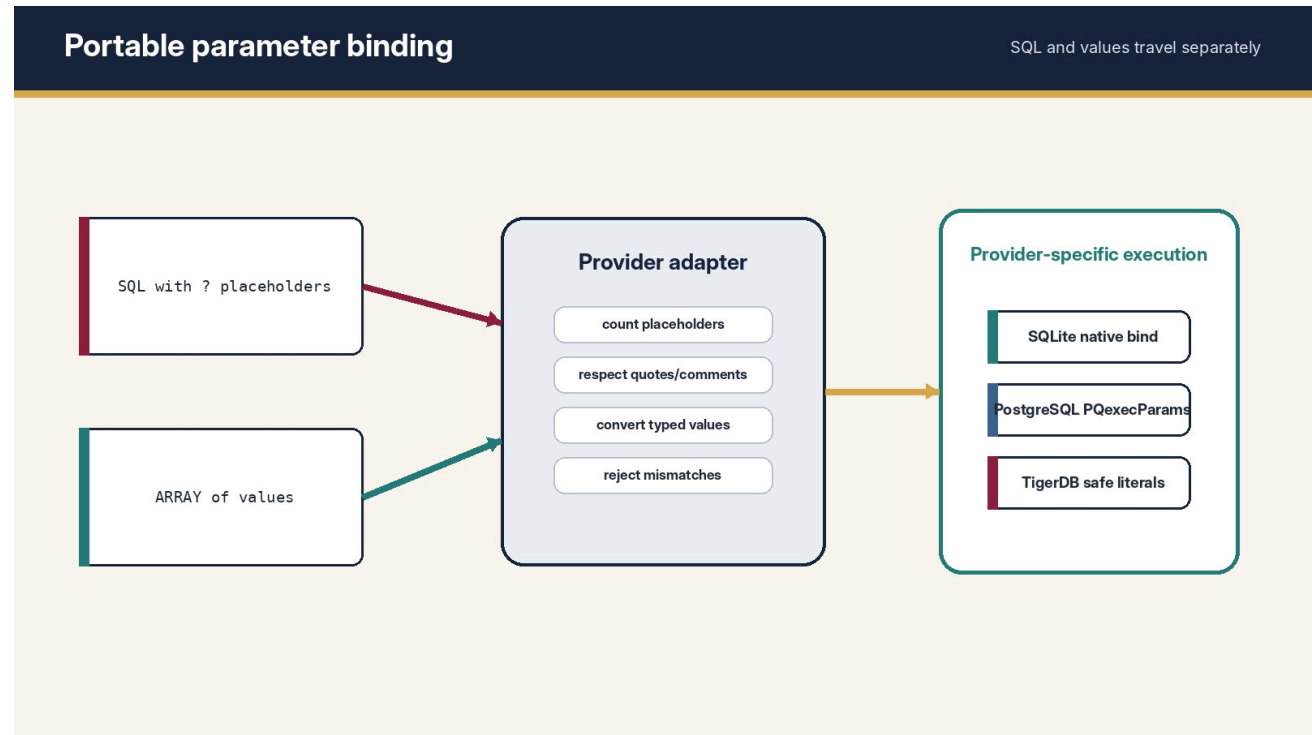


Figure 5. Kokum sends SQL structure and typed parameter values through separate paths

Execute accepts SQL and an optional ARRAY of values. Use a question mark for every value position. The parameter count must match exactly. SQLite binds through the native SQLite API, PostgreSQL rewrites value placeholders and sends values through PQexecParams, and TigerDB produces escaped SQL literals while respecting quoted strings, comments and dollar-quoted bodies.

Parameters are values, not identifiers or SQL fragments. A table name, column name, ordering direction or keyword cannot be replaced with a question mark. Select such structural choices from an application-controlled allow-list and construct only that trusted fragment.

Listing 5. A portable parameterized INSERT

```
FUNCTION AddCustomer(db AS DATABASE, id AS INTEGER, ;
                    name AS STRING, city AS STRING) AS INTEGER
RETURN db.Execute( ;
    "INSERT INTO customers(id, name, city, active) " + ;
    "VALUES (?, ?, ?, ?)", ;
    [id, name, city, .T.])
ENDFUNC
```

EXECUTED - SQLite project and DATABASE API tests; provider adapters also tested

NEVER CONCATENATE INPUT

Do not build SQL with a user-supplied name, city, identifier or search value. Parameter arrays preserve type information and eliminate quoting mistakes.

Table 6. High-level parameter conversion

Kokum value	SQLite	PostgreSQL	TigerDB
NULL	native NULL	NULL parameter	NULL literal

Kokum value	SQLite	PostgreSQL	TigerDB
LOGICAL	integer binding	boolean text	TRUE/FALSE literal
INTEGER / NUMBER	numeric binding	numeric text	numeric literal
DECIMAL	text/numeric affinity	numeric text	decimal literal
STRING / DATE / DATETIME	text binding	text parameter	quoted escaped literal
ARRAY / MAP	not bindable	JSON text	quoted JSON text

Chapter summary

The second argument to Execute and Query methods is an ARRAY of typed values. It is the default defense against value-based SQL injection and provider quoting differences.

Practice and review

14. Rewrite a concatenated customer lookup to use a parameter ARRAY.
15. Explain why ORDER BY ? cannot safely select a column.

CHAPTER 6

Choosing a result-set method

Ask only for the shape the program needs

LEARNING OBJECTIVES

select Query, QueryOne, QueryValue or QueryTable; understand zero-row behavior; include or omit table headings.

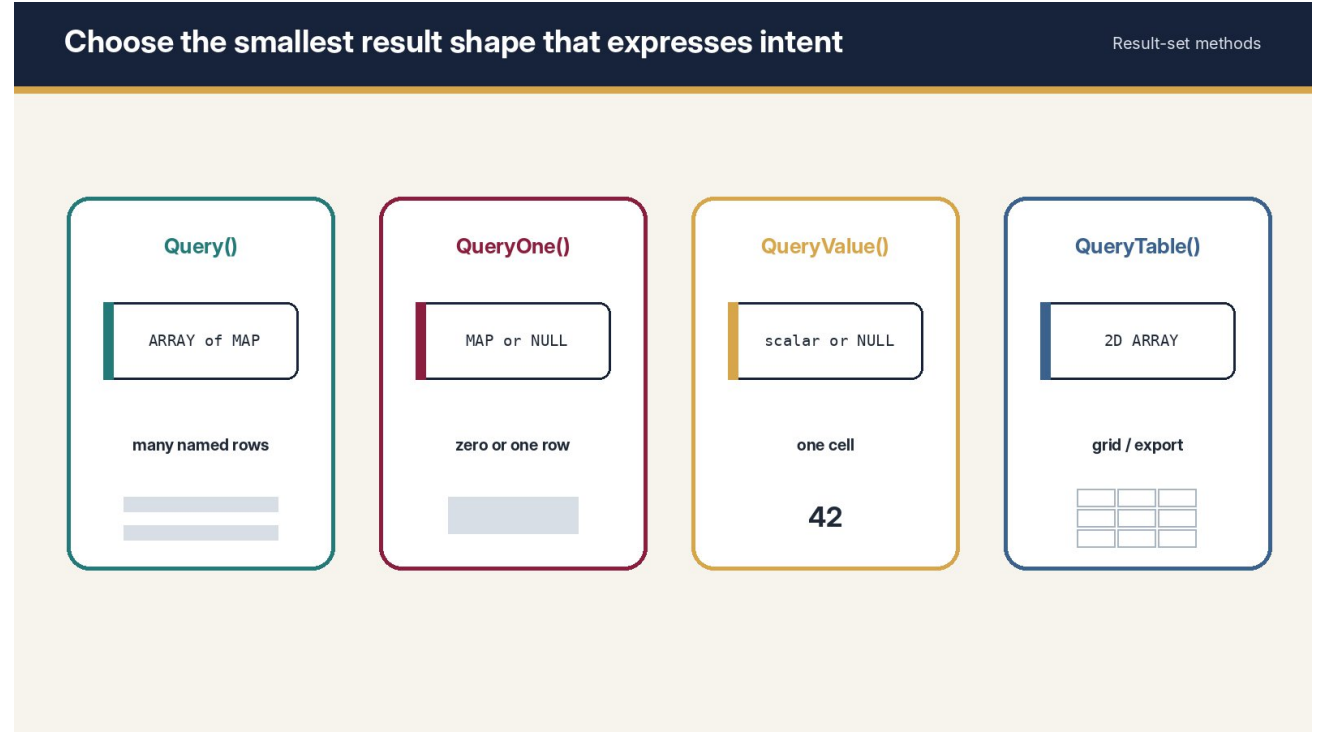


Figure 6. The four result methods express different cardinality and presentation needs

All query methods execute SQL but intentionally return different shapes. Query is the normal programmatic result: an ARRAY whose elements are MAP rows keyed by column label. QueryOne stops at the first row and returns a MAP or NULL. QueryValue returns the first column of the first row or NULL. QueryTable creates a two-dimensional ARRAY, optionally with a first row containing headings.

Listing 6. The four provider-neutral result methods

```
rows = db.Query("SELECT id, name FROM customers ORDER BY id")
row = db.QueryOne("SELECT id, name FROM customers WHERE id = ?", [7])
count = db.QueryValue("SELECT COUNT(*) FROM customers")
table = db.QueryTable( ;
    "SELECT id, name FROM customers ORDER BY id", ;
    NULL, ;
    .T.)
```

CHECKED - exact method signatures validated by compiler and DATABASE API tests

Table 7. Result-set selection guide

Method	Returns	Zero rows	Best use
Query	ARRAY of MAP	empty ARRAY	business logic, APIs, iteration
QueryOne	MAP or NULL	NULL	lookup by key, optional record
QueryValue	scalar or NULL	NULL	COUNT, SUM, settings, existence value
QueryTable	2D ARRAY	empty or headings-only ARRAY	tabular output, export, table variable

INTENT MATTERS

Using Query for COUNT(*) works, but it forces the caller to know

row and column names. QueryValue states the one-cell contract directly.

Chapter summary

Choose the narrowest method whose return contract matches the caller. This reduces indexing code, NULL mistakes and accidental dependence on column ordering.

Practice and review

16. Choose a method for a dashboard count, customer detail page, CSV export and batch of invoices.
17. Call QueryTable once with headings and once without headings.

CHAPTER 7

Working with rows, maps and tables

Turn a result set into useful application data

LEARNING OBJECTIVES

iterate ARRAY-of-MAP results; use SQL aliases as stable map keys; handle QueryTable row positions deliberately.

Query rows are MAP values. Column labels become keys, so aliases are part of the repository contract. Prefer explicit column lists and stable aliases; SELECT * allows schema changes to alter the returned map without changing source code.

Listing 7. Iterate named row maps returned by Query

```
PROCEDURE PrintCustomers(db AS DATABASE)
  LOCAL rows AS ARRAY
  LOCAL row AS MAP

  rows = db.Query( ;
    "SELECT id, name, city AS home_city " + ;
    "FROM customers ORDER BY name")

  FOR EACH row IN rows
    ? STR(row["id"]) + ": " + row["name"] + ;
    " (" + STR(row["home_city"]) + ")"
  NEXT
ENDPROC
```

CHECKED - collection iteration and database result shapes passed semantic/API tests

Column names and aliases

Use aliases when expressions or duplicate column names would otherwise be ambiguous. For joined tables, alias every identifier that the caller reads. A repository can then change its join strategy without changing the map contract.

Listing 8. Stable aliases for joined and aggregated rows

```
rows = db.Query( ;
  "SELECT c.id AS customer_id, c.name AS customer_name, " + ;
  "COUNT(o.id) AS order_count " + ;
  "FROM customers c LEFT JOIN orders o ON o.customer_id = c.id " + ;
  "GROUP BY c.id, c.name ORDER BY c.name")
```

CHECKED - portable query invocation syntax

TABLE RESULTS

QueryTable is positional. When headings are enabled, row 1 contains column names and data begins at row 2. For application logic, Query is usually safer because map keys travel with values.

Chapter summary

Query returns self-describing row maps; QueryTable returns positional arrays. Explicit aliases create a stable boundary between SQL and the caller.

Practice and review

18. Write a query whose map keys are customer_id, customer_name and balance_due.
19. Explain the risk of reading row[1] instead of row["customer_id"].

CHAPTER 8

NULL and provider type conversion

Preserve the difference between missing, false, zero and empty

LEARNING OBJECTIVES

test NULL explicitly; understand provider result conversion; avoid accidental stringification.

A database NULL becomes Kokum NULL. It is not an empty string, zero or logical false. QueryOne and QueryValue also return NULL when no row exists, so the caller must distinguish cardinality from a legitimate value. Use an explicit comparison or TYPEOF when a typed local would otherwise obscure the case.

Listing 9. Handle a nullable scalar result

```
FUNCTION CustomerCity(db AS DATABASE, id AS INTEGER) AS STRING
  LOCAL city

  city = db.QueryValue( ;
    "SELECT city FROM customers WHERE id = ?", ;
    [id])

  IF city = NULL
    RETURN "(not available)"
  ENDIF
  RETURN city
ENDFUNC
```

CHECKED - NULL comparison and QueryValue contract validated

Table 8. Result conversion overview

Database value	Typical Kokum value
SQL NULL	NULL
Boolean	LOGICAL
Integral numeric	INTEGER
Floating numeric	NUMBER
Exact numeric / DECIMAL	DECIMAL where supported
Text	STRING
Date	DATE or provider text according to provider mapping
Timestamp	DATETIME or provider text according to provider mapping
JSON / JSONB	decoded ARRAY, MAP or scalar in PostgreSQL; text/JSON in other paths
SQLite BLOB result	ARRAY of byte integers

DO NOT STRINGIFY TOO EARLY

STR is useful for display and diagnostics, but converting every database value to STRING discards numeric, logical, date and JSON semantics. Keep typed values until the presentation boundary.

Chapter summary

Kokum preserves NULL and maps provider values into language types. Correct result handling begins with explicit cardinality and NULL decisions.

Practice and review

- 20. Change the example so a missing customer and a customer with a NULL city produce different messages.
- 21. List two bugs caused by treating NULL as an empty string.

CHAPTER 9

Affected rows and generated identifiers

Observe mutations without issuing unnecessary queries

LEARNING OBJECTIVES

use Execute return values and Changes; use LastInsertId where meaningful; use PostgreSQL RETURNING for server-generated values.

Execute returns an affected-row count for data-changing statements when the provider reports one. The Changes property records the most recent mutation count. SQLite also exposes LastInsertId for rowid-based inserts. Provider semantics differ, so repository code should not assume every server produces a last identifier the same way.

Listing 10. Read mutation counts and the SQLite last row identifier

```
changed = dbLocal.Execute( ;
    "UPDATE customers SET active = ? WHERE city = ?", ;
    [.F., "Ratnagiri"])
? "Updated: " + STR(changed)
? "Provider changes: " + STR(dbLocal.Changes)

newId = dbLocal.LastInsertId
```

EXECUTED - SQLite live database regression

PostgreSQL generated values

Listing 11. Use RETURNING for PostgreSQL generated columns

```
created = dbPostgres.QueryOne( ;
    "INSERT INTO customers(name, city) VALUES (?, ?) " + ;
    "RETURNING id, name, city", ;
    [name, city])
? "Created id: " + STR(created["id"])
```

CHECKED + PROTOCOL TESTED - libpq provider and result conversion suite

DDL COUNTS

CREATE TABLE and other schema statements may report zero affected rows even when they succeed. Treat successful completion, not a positive count, as the DDL success condition.

Chapter summary

Use affected-row counts to detect optimistic updates and deletes. Use provider-appropriate generated-value retrieval rather than assuming a universal last-insert behavior.

Practice and review

22. Reject an UPDATE when its affected-row count is not exactly one.
23. Implement a PostgreSQL insert that returns both id and created_at.

CHAPTER 10

Transactions and deterministic cleanup

Make multi-statement changes all-or-nothing

LEARNING OBJECTIVES

define transaction boundaries; commit only after every invariant holds; roll back on all failure paths.

A transaction is connection-local

Atomic unit of work

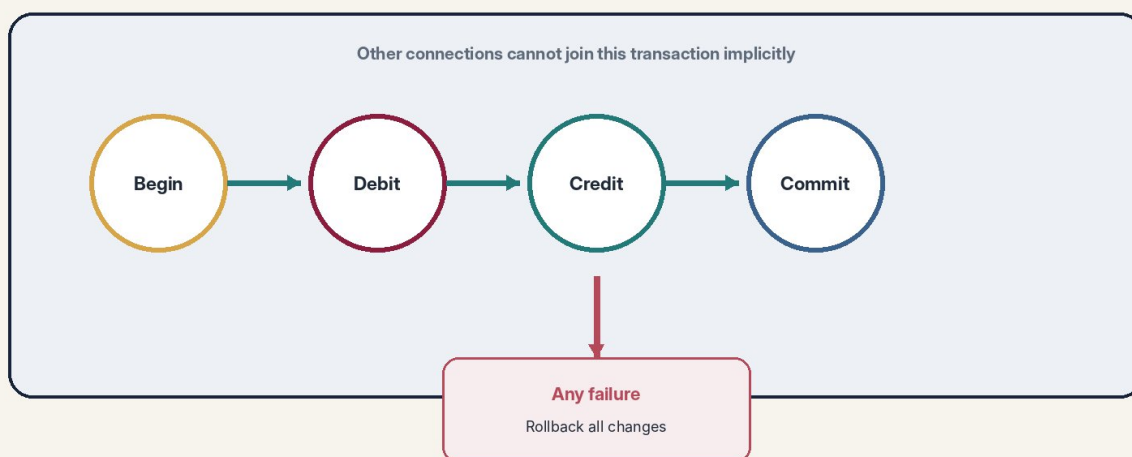


Figure 7. A transaction groups multiple operations on one connection

Begin starts a provider transaction on the current connection. Commit makes its changes durable according to provider configuration; Rollback discards them. The transaction belongs to that connection only. An async worker or another HTTP service worker has a separate connection and cannot silently join it.

Listing 12. A two-statement transfer with explicit rollback

```

FUNCTION Transfer(db AS DATABASE, fromId AS INTEGER, ;
                 toId AS INTEGER, amount AS DECIMAL) AS LOGICAL
  db.Begin()
  TRY
    db.Execute( ;
      "UPDATE accounts SET balance = balance - ? WHERE id = ?", ;
      [amount, fromId])
    db.Execute( ;
      "UPDATE accounts SET balance = balance + ? WHERE id = ?", ;
      [amount, toId])
    db.Commit()
    RETURN .T.
  CATCH error
    db.Rollback()
    ? "Transfer failed: " + STR(error)
    RETURN .F.
  ENDTRY
ENDFUNC
  
```

CHECKED - transaction methods and error path validated by DATABASE API tests

TRANSACTION DISCIPLINE

Begin as late as possible, perform only required work, validate affected counts, and commit promptly. Never wait for user input or a network call while holding a database transaction.

Chapter summary

Transactions are explicit and connection-local. Reliable code commits only after all required statements succeed and rolls back every failure path.

Practice and review

24. Add affected-row validation to both UPDATE statements.
25. Explain why an async task launched after Begin does not participate in the owner connection transaction.

PART 2

SQLite: embedded relational storage

SQLite is the easiest place to learn the common DATABASE API and an excellent fit for local application data, caches, small services and single-node deployments.

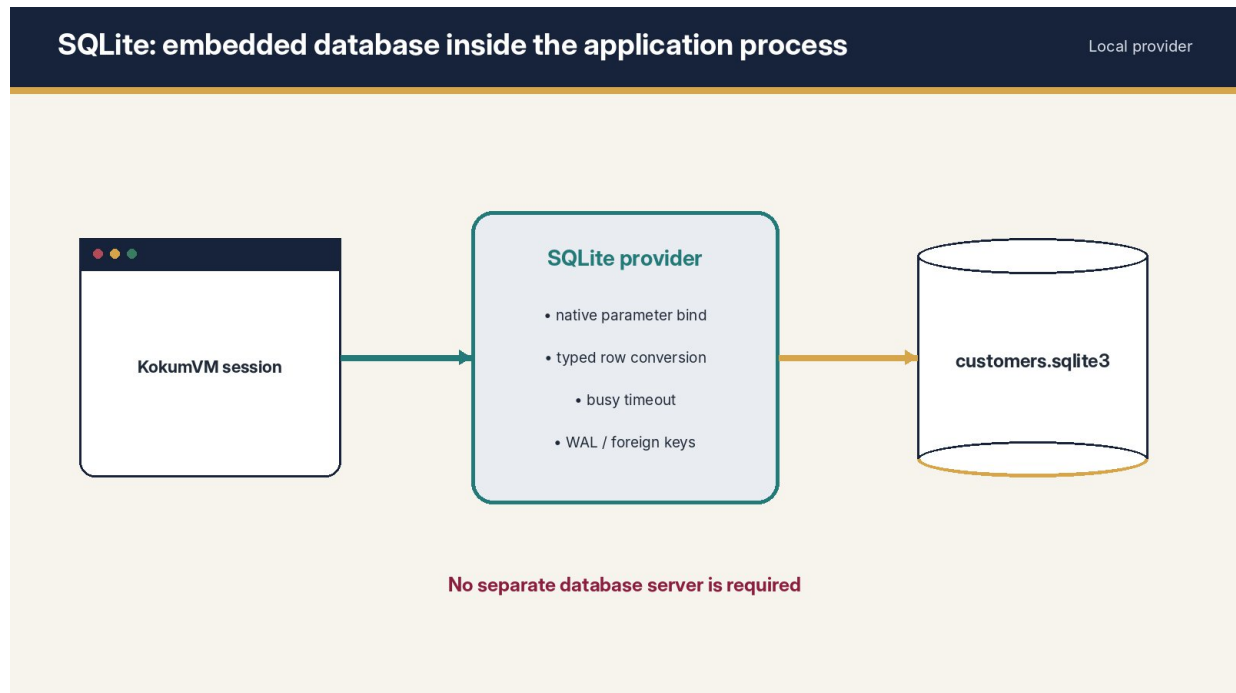


Figure 8. Part 2 overview: SQLite: embedded relational storage

Chapters in this part

- 11. SQLite configuration and open modes
- 12. Schema creation, initialization and migrations
- 13. SQLite CRUD in practice
- 14. Joins, aggregates, sorting and pagination
- 15. SQLite values, dates, decimals and BLOB results
- 16. WAL, contention and deployment

CHAPTER 11

SQLite configuration and open modes

Control file location, creation and connection pragmas

LEARNING OBJECTIVES

configure a local SQLite component; choose an open mode; understand auto-open and parent directory behavior.

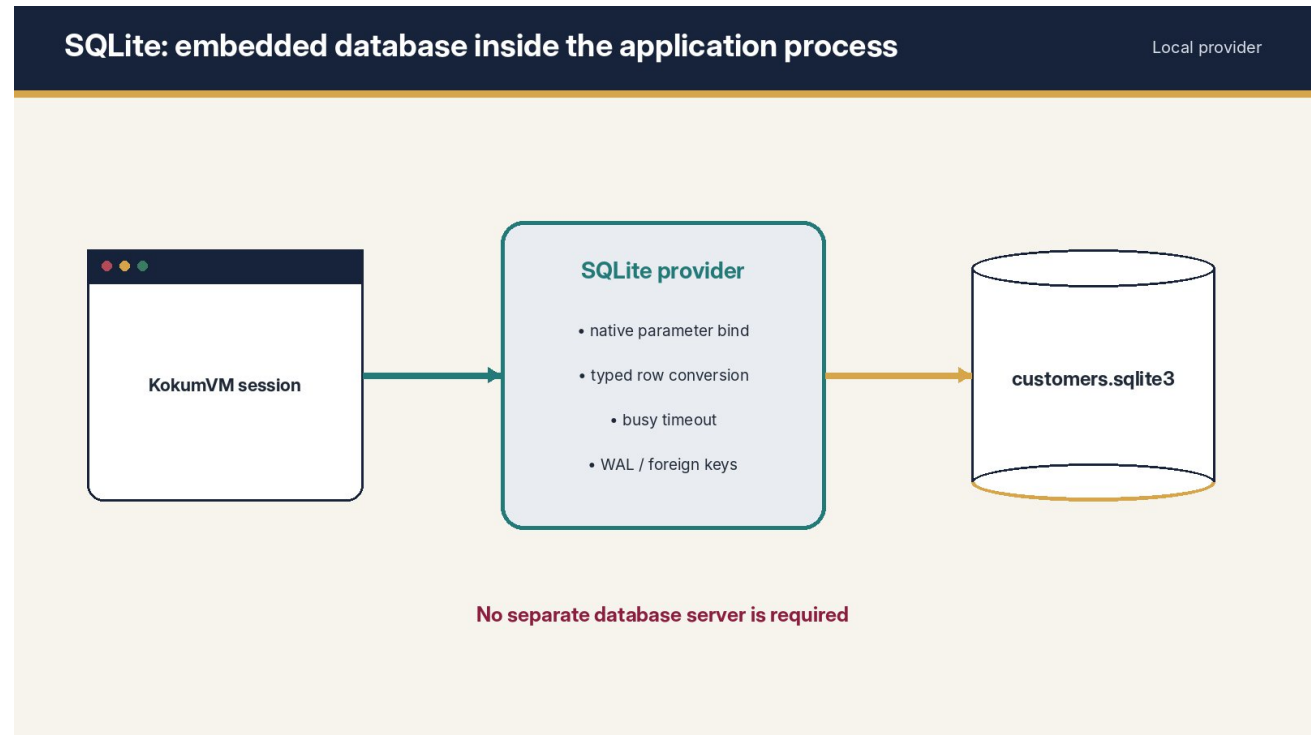


Figure 9. SQLite runs in-process and stores data in a local database file

The SQLite provider is built into KokumVM. A typical component points to a file beneath APP_DIR, opens read-write with creation enabled, creates parent directories, uses a finite busy timeout, enables foreign keys and chooses WAL with NORMAL synchronous behavior.

Listing 13. A production-oriented SQLite db_env.conf section

```
[database:dbLocal]
provider=sqlite
module=company.customers.data
file=${APP_DIR}/data/customers.sqlite3
open_mode=readwrite-create
auto_open=true
create_parent_directory=true
busy_timeout_ms=5000
journal_mode=WAL
foreign_keys=true
synchronous=NORMAL
initialization_script=
```

EXECUTED + CONFIG VALIDATED - generated project, bundle and standalone deployment

Table 9. SQLite open modes

open_mode	Behavior
readwrite-create	Open read-write and create the file when absent.
readwrite	Require an existing writable database.
readonly	Open without allowing mutations.

PATH CHOICE

APP_DIR is predictable for a deployed service. USER_DATA_DIR

is usually better for a per-user desktop data file. Confirm backup and operating-system permissions for either location.

Chapter summary

SQLite configuration controls file placement, creation, locking behavior and important pragmas before application SQL runs.

Practice and review

26. Create a read-only reporting configuration.
27. Explain when `USER_DATA_DIR` is preferable to `APP_DIR`.

CHAPTER 12

Schema creation, initialization and migrations

Create a database deliberately and evolve it safely

LEARNING OBJECTIVES

use initialization_script correctly; write idempotent schema SQL; separate first creation from later migrations.

initialization_script runs only when the configured database file did not exist before opening. It is useful for a new installation, but it is not a general migration engine. Existing installations need versioned, transactional migration logic that records which changes were applied.

Listing 14. Idempotent base schema creation

```
PROCEDURE EnsureSchema(db AS DATABASE)
  db.Execute( ;
    "CREATE TABLE IF NOT EXISTS schema_version ( " + ;
    "version INTEGER NOT NULL)" )
  db.Execute( ;
    "CREATE TABLE IF NOT EXISTS customers ( " + ;
    "id INTEGER PRIMARY KEY AUTOINCREMENT, " + ;
    "name TEXT NOT NULL, city TEXT, " + ;
    "active INTEGER NOT NULL DEFAULT 1)" )
  db.Execute( ;
    "CREATE INDEX IF NOT EXISTS idx_customers_city " + ;
    "ON customers(city)" )
ENDPROC
```

EXECUTED - SQLite customer project and live database file

A minimal migration pattern

Listing 15. Transactional SQLite schema upgrade

```
version = db.QueryValue("PRAGMA user_version")
IF version < 2
  db.Begin()
  TRY
    db.Execute("ALTER TABLE customers ADD COLUMN email TEXT")
    db.Execute("PRAGMA user_version = 2")
    db.Commit()
  CATCH error
    db.Rollback()
    ? STR(error)
  ENDTRY
ENDIF
```

CHECKED - common transaction and query syntax; test on a copy before production

MIGRATION SAFETY

Back up the database, test upgrades from every supported previous version, and make application startup fail clearly when a migration cannot complete.

Chapter summary

Use initialization_script only for first creation. Treat later schema changes as explicit, versioned and tested migrations.

Practice and review

28. Design migration version 3 that adds a unique index on email.
29. Explain why IF NOT EXISTS alone is not a complete migration strategy.

CHAPTER 13

SQLite CRUD in practice

A complete create, read, update and delete cycle

LEARNING OBJECTIVES

write parameterized CRUD; use LastInsertId; check mutation cardinality.

Listing 16. SQLite CRUD with generated identifier and row map

```
PROCEDURE CustomerCrud(db AS DATABASE)
  LOCAL id AS INTEGER
  LOCAL row

  db.Execute( ;
    "INSERT INTO customers(name, city, active) VALUES (?, ?, ?)", ;
    ["Anil Jagtap", "Ratnagiri", .T.])
  id = db.LastInsertId

  row = db.QueryOne( ;
    "SELECT id, name, city, active FROM customers WHERE id = ?", ;
    [id])
  ? JSONENCODE(row)

  db.Execute("UPDATE customers SET city = ? WHERE id = ?", ["Pune", id])
  db.Execute("DELETE FROM customers WHERE id = ?", [id])
ENDPROC
```

EXECUTED - source, bundle, KokumVM and standalone SQLite regression

The example keeps values out of SQL and reads only the columns its caller needs. A production repository should check that UPDATE and DELETE each affect exactly one row. If zero rows are legitimate, return that fact to the caller rather than hiding it.

Table 10. Repository contracts for CRUD

Operation	Recommended return contract
Create	New row map or generated id.
Read by id	MAP or NULL.
List/search	ARRAY of MAP.
Update	Affected count or updated row.
Delete	Affected count.

UNIQUENESS FAILURES

Constraint errors are catchable provider errors. Convert only expected constraints into domain responses; log and propagate unexpected failures with useful context.

Chapter summary

SQLite CRUD uses the same common methods as remote providers. Parameter arrays and explicit result contracts keep repository functions predictable.

Practice and review

- 30. Add optimistic locking with a version column in the UPDATE WHERE clause.
- 31. Return a logical value from DeleteCustomer based on affected rows.

CHAPTER 14

Joins, aggregates, sorting and pagination

Shape relational data close to the database

LEARNING OBJECTIVES

write stable joined result maps; use aggregate aliases; paginate deterministically.

Relational databases are strongest when filtering, joining, grouping and sorting occur in SQL. Transfer only the columns and rows the caller needs. Always add a deterministic ORDER BY before pagination; without it, pages are not a stable contract.

Listing 17. Joined aggregate with stable aliases

```
FUNCTION CustomerOrderSummary(db AS DATABASE) AS ARRAY
  RETURN db.Query( ;
    "SELECT c.id AS customer_id, c.name AS customer_name, " + ;
    "COUNT(o.id) AS order_count, " + ;
    "COALESCE(SUM(o.total), 0) AS order_total " + ;
    "FROM customers c LEFT JOIN orders o " + ;
    "ON o.customer_id = c.id " + ;
    "GROUP BY c.id, c.name " + ;
    "ORDER BY order_total DESC, customer_id")
ENDFUNC
```

CHECKED - Query result contract and SQL invocation

Listing 18. Deterministic LIMIT/OFFSET pagination

```
FUNCTION LoadCustomerPage(db AS DATABASE, ;
  pageSize AS INTEGER, offsetRows AS INTEGER) AS ARRAY
  RETURN db.Query( ;
    "SELECT id, name, city FROM customers " + ;
    "ORDER BY id LIMIT ? OFFSET ?", ;
    [pageSize, offsetRows])
ENDFUNC
```

CHECKED - parameterized query syntax

LARGE RESULT SETS

Query materializes the returned ARRAY. Apply WHERE, LIMIT and suitable indexes rather than loading an entire large table and filtering in Kokum.

Chapter summary

Use SQL to perform relational work and return stable aliases. Pagination requires an explicit deterministic order and appropriate indexes.

Practice and review

32. Add a WHERE clause that reports only active customers.
33. Compare OFFSET pagination with keyset pagination for a large table.

CHAPTER 15

SQLite values, dates, decimals and BLOB results

Understand dynamic storage without losing application meaning

LEARNING OBJECTIVES

bind supported Kokum values; interpret SQLite affinity; handle BLOB query results.

SQLite uses dynamic typing and column affinity. Kokum binds NULL, LOGICAL, INTEGER, NUMBER, DECIMAL, STRING, DATE and DATETIME. Arrays and maps are not accepted as SQLite parameters; encode structured data explicitly with JSONENCODE and store it in TEXT when that design is appropriate.

Listing 19. Bind dates, exact decimals and encoded JSON to SQLite

```
created = DATETIME()  
amount = DECIMAL("1250.75")  
metadata = JSONENCODE({"source": "kokum", "verified": .T.})  
  
db.Execute( ;  
    "INSERT INTO events(created_at, amount, metadata) VALUES (?, ?, ?)", ;  
    [created, amount, metadata])
```

CHECKED + API TESTED - SQLite parameter conversion

Table 11. SQLite low-level result conversion

SQLite result kind	Kokum result
NULL	NULL
INTEGER	INTEGER
FLOAT	NUMBER
TEXT	STRING
BLOB	ARRAY containing byte integers

EXACT MONEY VALUES

Choose a deliberate representation. DECIMAL preserves exact decimal intent in Kokum, but SQLite affinity and SQL expressions still require testing. Many financial schemas store the smallest currency unit as INTEGER.

Chapter summary

SQLite accepts the main scalar Kokum types and returns BLOB data as byte arrays. Structured ARRAY and MAP parameters must be encoded explicitly.

Practice and review

- 34. Store and retrieve a MAP as JSON text.
- 35. Design a test proving that a monetary calculation remains exact for your chosen schema.

CHAPTER 16

WAL, contention and deployment

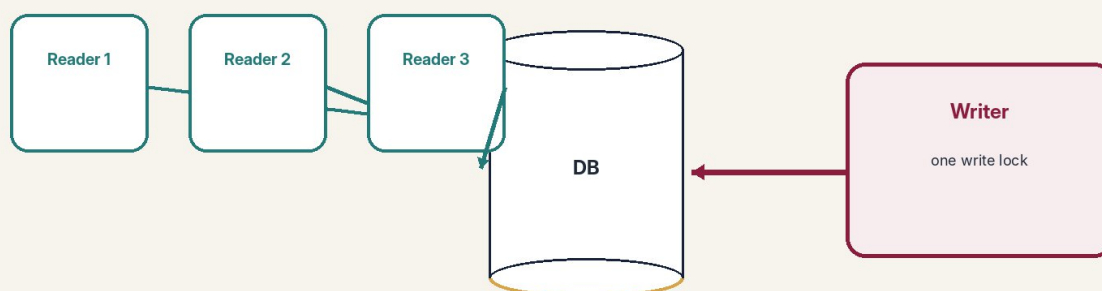
Make a local database reliable under real workload

LEARNING OBJECTIVES

understand WAL reader/writer behavior; configure `busy_timeout`; deploy and back up the complete database state.

SQLite WAL: concurrent readers, serialized writer

Concurrency model



busy_timeout absorbs brief lock contention

Keep write transactions small and commit promptly

Figure 10. WAL permits concurrent readers while writes remain serialized

WAL is a strong default for many Kokum applications because readers do not block a writer in the same way as the rollback journal. SQLite still permits only one writer at a time. `busy_timeout` allows short conflicts to wait rather than fail immediately, but it cannot repair long transactions or an overloaded write path.

Listing 20. A bounded single-statement SQLite write

```
FUNCTION RenameCustomer(db AS DATABASE, id AS INTEGER, ;
                        newName AS STRING) AS LOGICAL
  TRY
    RETURN db.Execute( ;
      "UPDATE customers SET name = ? WHERE id = ?", ;
      [newName, id]) = 1
  CATCH error
    ? "SQLite write failed: " + STR(error)
    RETURN .F.
  ENDTRY
ENDFUNC
```

EXECUTED - SQLite provider and live database regression

BACKUP BOUNDARY

With WAL enabled, a hot database may also have `-wal` and `-shm` files. Use an SQLite-aware backup procedure or `stop/checkpoint` the application according to your operational plan; do not copy only the main file blindly.

Deployment checklist

- Create the data directory with least-privilege ownership.
- Keep the database outside a replaceable application package when upgrades overwrite APP_DIR.
- Monitor file growth, free space, lock errors and integrity checks.
- Test restore, not only backup creation.
- Use read-only mode for reporting deployments that must not mutate data.

Chapter summary

WAL improves concurrency but does not remove SQLite write serialization. Short transactions, a sensible busy timeout and tested backup/restore procedures are essential.

Practice and review

36. Measure lock behavior with two writer processes.
37. Document the exact files and steps required to restore your deployed SQLite database.

PART 3

PostgreSQL: shared transactional services

The PostgreSQL provider is built on libpq and maps the common Kokum DATABASE API onto a mature shared server. This part emphasizes secure configuration, typed results and SQL features commonly used in service applications.

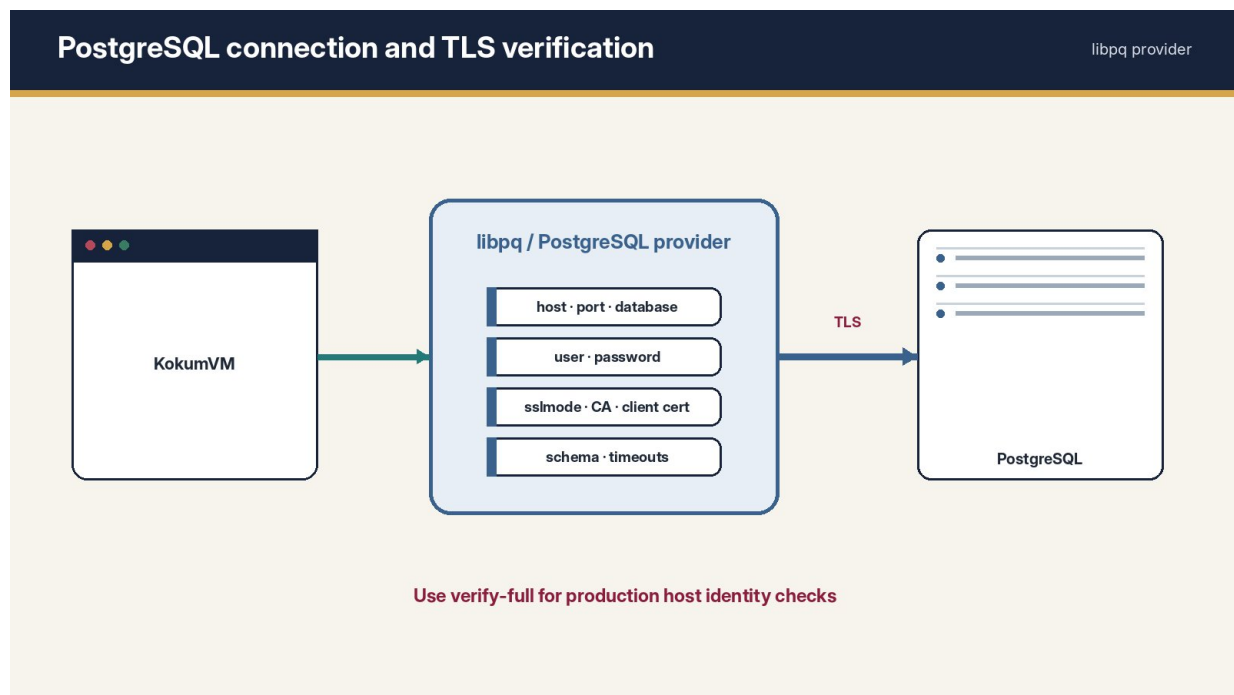


Figure 11. Part 3 overview: PostgreSQL: shared transactional services

Chapters in this part

- 17. PostgreSQL build and connection configuration
- 18. TLS, schema identity and read-only sessions
- 19. Portable parameters, casts and placeholder scanning
- 20. PostgreSQL result types and JSONB
- 21. RETURNING, transactions and diagnostics

CHAPTER 17

PostgreSQL build and connection configuration

Prepare libpq and define a clear application identity

LEARNING OBJECTIVES

recognize the build dependency; configure host, database and credentials; open and test a PostgreSQL session.

The PostgreSQL provider is compiled with libpq. On a Linux development host, the corresponding development package must be available when building Kokum. The canonical provider name is postgresql; postgres and pgsq are accepted aliases by the runtime configuration parser.

Listing 21. PostgreSQL connection configuration

```
[database:dbPostgres]
provider=postgresql
module=company.accounting.data
host=127.0.0.1
port=5432
database=accounting
username=kokum_app
password=${ENV:KOKUM_DBPOSTGRES_PASSWORD}
schema=public
application_name=KokumAccounting
auto_open=false
connect_timeout_ms=10000
query_timeout_ms=30000
read_only=false
```

CONFIG VALIDATED + PROTOCOL TESTED - provider suite and project check

Listing 22. Open PostgreSQL and inspect server identity

```
PROCEDURE OpenPostgres(db AS DATABASE)
TRY
    db.Open()
    ? JSONENCODE(db.Ping())
    ? "Server: " + db.ServerVersion
    ? "Backend PID: " + STR(db.BackendPid)
CATCH error
    ? "PostgreSQL connection failed: " + STR(error)
ENDTRY
ENDPROC
```

CHECKED + PROTOCOL TESTED - libpq adapter tests; run against staging before production

BUILD REQUIREMENT

A source build needs PostgreSQL client headers and libpq. A deployed binary must be packaged according to the platform build and licensing plan used by the project.

Chapter summary

PostgreSQL connections are configured externally and implemented through libpq. ApplicationName, timeouts and provider properties improve observability.

Practice and review

38. Create separate development and production db_env.conf files using the same component ID.
39. Explain why ApplicationName is useful to a database administrator.

CHAPTER 18

TLS, schema identity and read-only sessions

Verify the server you intended to reach

LEARNING OBJECTIVES

choose an sslmode; configure CA and client certificates; use schema and read-only settings intentionally.

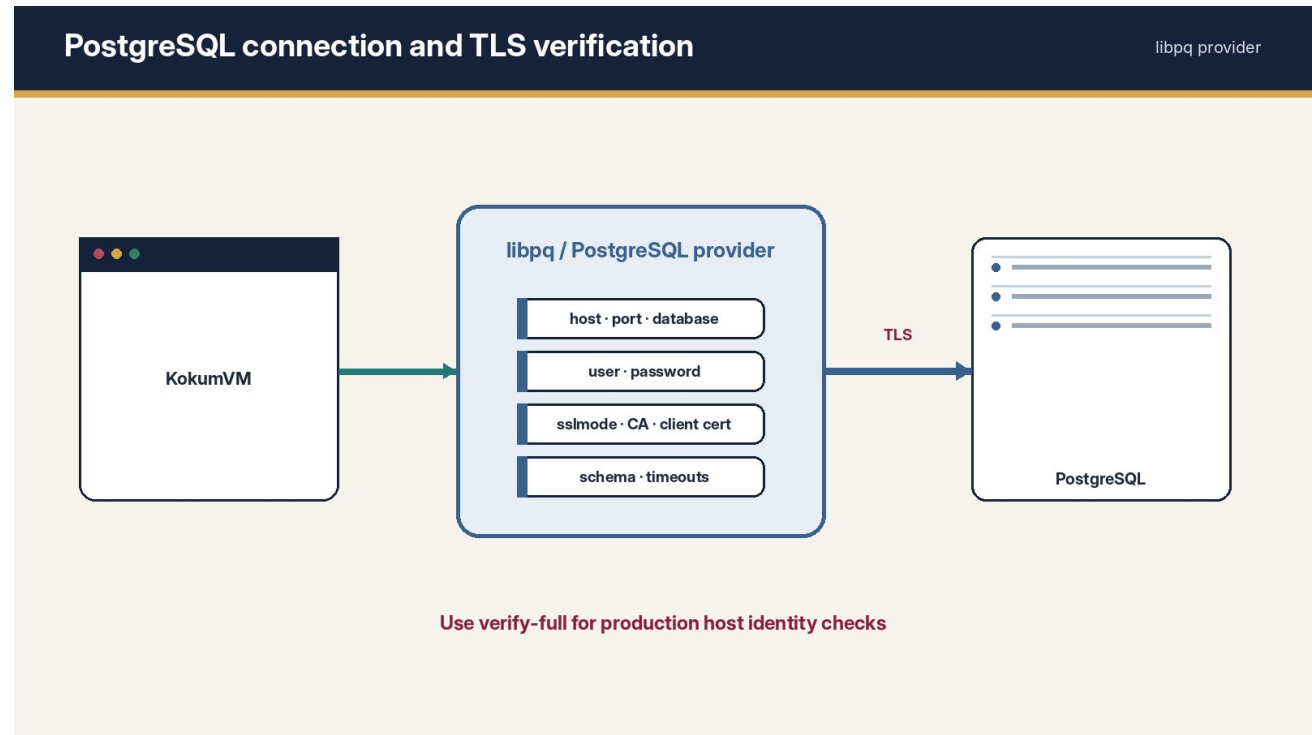


Figure 12. PostgreSQL TLS and identity configuration through libpq

sslmode controls whether encryption and certificate verification are required. Production systems should normally use verify-full with a trusted root certificate so both the certificate chain and host identity are checked. sslcert and sslkey support client certificate authentication where the server requires it.

Listing 23. Verified-identity, read-only PostgreSQL configuration

```
[database:dbPostgres]
provider=postgresql
host=postgres.internal.example
port=5432
database=ledger
username=ledger_reader
password=${ENV:LEDGER_READER_PASSWORD}
schema=reporting
sslmode=verify-full
sslrootcert=${APP_DIR}/tls/postgres-ca.crt
sslcert=${APP_DIR}/tls/client.crt
sslkey=${APP_DIR}/tls/client.key
application_name=KokumLedgerReport
read_only=true
```

CONFIG VALIDATED - PostgreSQL property schema and libpq option construction

Table 12. PostgreSQL sslmode choices

sslmode	Meaning
disable	No TLS.
allow	Try non-TLS, then TLS if required.
prefer	Try TLS first, then non-TLS.
require	Require TLS but do not fully verify identity.
verify-ca	Require TLS and validate the certificate chain.

sslmode	Meaning
verify-full	Validate chain and host identity; preferred for production.

READ-ONLY IS DEFENSE IN DEPTH

read_only=true protects the session from accidental writes, but it does not replace a database role whose privileges are genuinely read-only. Use both.

Chapter summary

TLS settings determine encryption and identity verification. Schema, ApplicationName and read-only mode make a connection more explicit and auditable.

Practice and review

40. Explain the difference between require and verify-full.
41. Design a least-privilege reporting role and matching Kokum configuration.

CHAPTER 19

Portable parameters, casts and placeholder scanning

Use question marks without confusing SQL syntax

LEARNING OBJECTIVES

understand PostgreSQL placeholder rewrite; use explicit casts when inference is ambiguous; recognize ignored question marks.

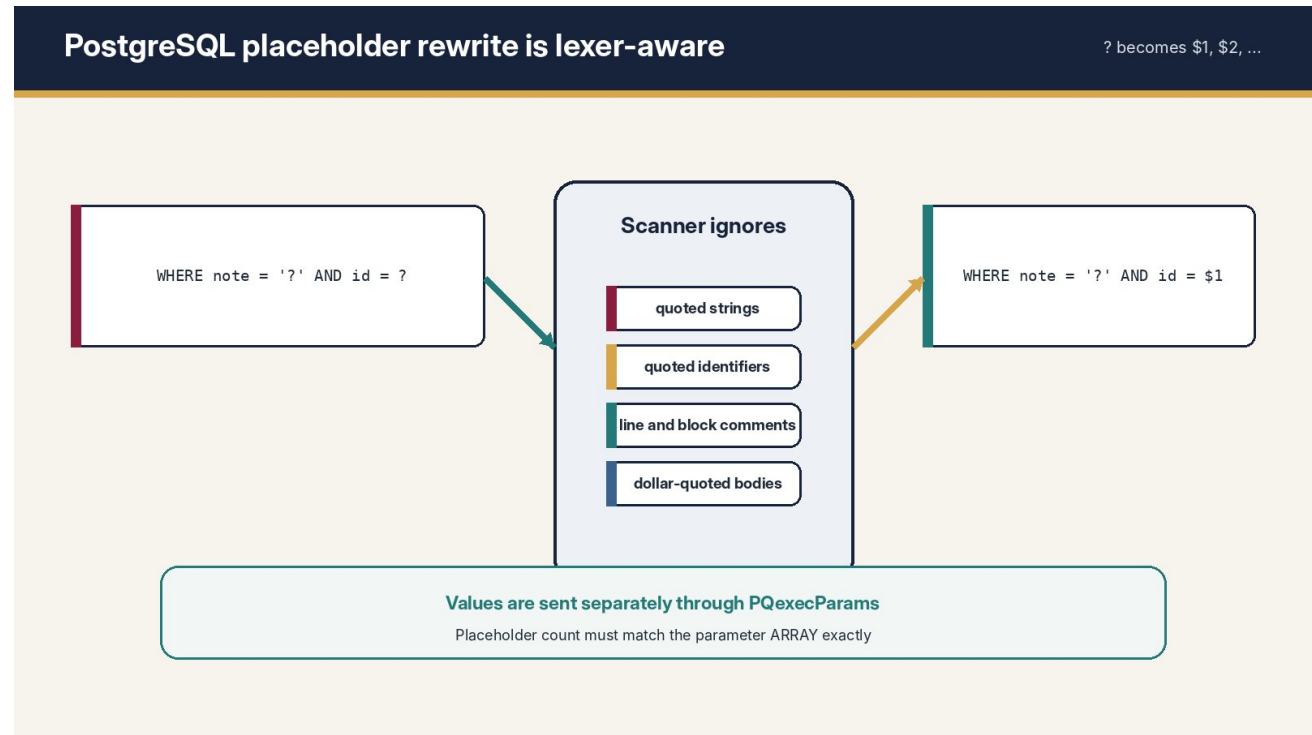


Figure 13. The PostgreSQL adapter rewrites only real value placeholders

Kokum source uses question marks for all common providers. The PostgreSQL adapter rewrites them to PostgreSQL positional parameters while scanning SQL structure. Question marks inside quoted strings, quoted identifiers, line comments, block comments and dollar-quoted bodies are ignored.

Listing 24. Parameterized PostgreSQL query with an explicit cast

```
rows = dbPostgres.Query( ;
    "SELECT id, name FROM customers " + ;
    "WHERE active = ? AND created_at >= ?::timestampz " + ;
    "ORDER BY id", ;
    [.T., "2026-01-01T00:00:00Z"])
```

CHECKED + PROTOCOL TESTED - placeholder scanner and PQexecParams suite

Listing 25. A quoted question mark is not a parameter

```
value = dbPostgres.QueryValue( ;
    "SELECT '?' AS literal_question, ?::integer AS supplied_value", ;
    [42])
```

PROTOCOL TESTED - lexer-aware placeholder rewrite

COUNT MUST MATCH

The number of real placeholders must equal the ARRAY length exactly. A mismatch is rejected before execution, which catches many dynamically assembled SQL errors early.

Chapter summary

Kokum keeps the common question-mark syntax while using PostgreSQL native parameter execution. The scanner respects SQL quoting and comments.

Practice and review

42. Add a JSONB parameter with an explicit `::jsonb` cast.
43. Construct a test containing question marks in a string, comment and real value position.

CHAPTER 20

PostgreSQL result types and JSONB

Receive rich server values as Kokum values

LEARNING OBJECTIVES

understand OID-based conversion; work with decoded JSON and JSONB; preserve exact numeric and temporal meaning.

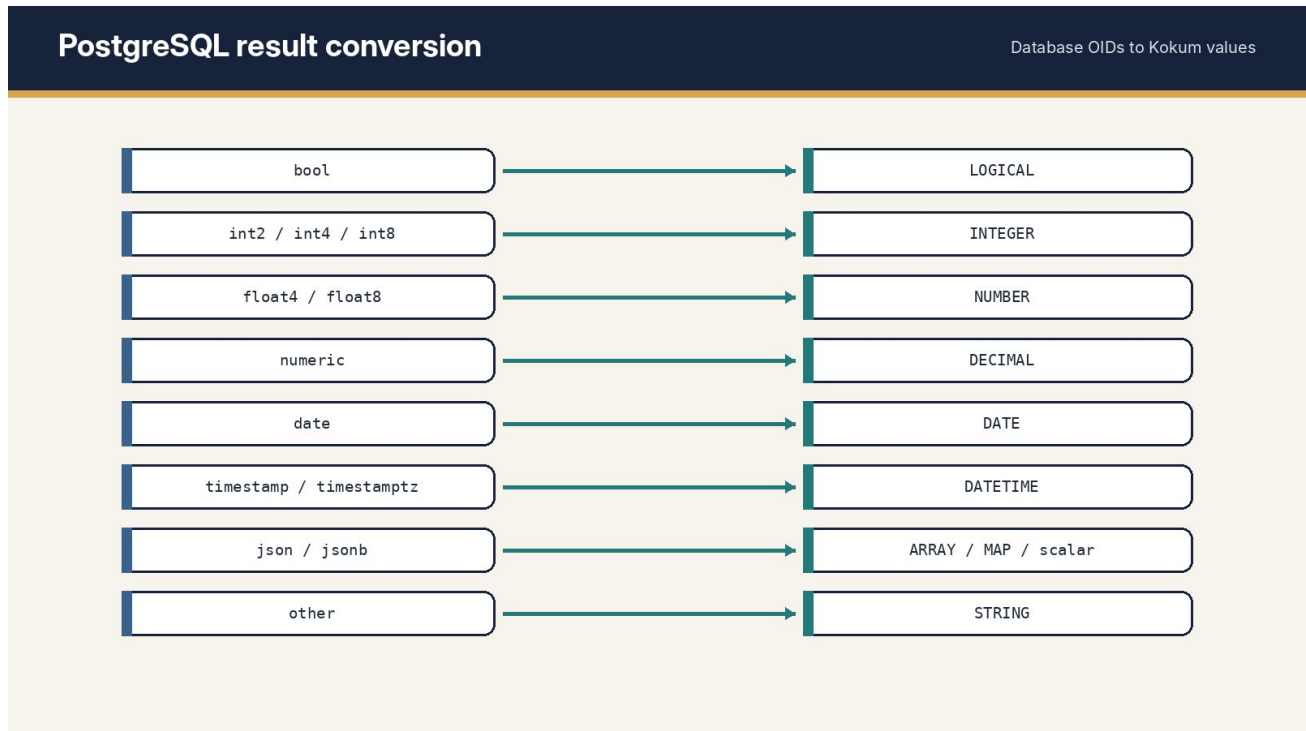


Figure 14. PostgreSQL OIDs determine the Kokum result value type

The provider converts common PostgreSQL OIDs rather than returning every cell as text. Booleans become LOGICAL, integer types become INTEGER, floating types become NUMBER, numeric becomes DECIMAL, dates and timestamps become DATE or DATETIME, and JSON/JSONB is decoded into an ARRAY, MAP or scalar. Unknown types remain STRING so data is not silently lost.

Listing 26. Query a JSONB property and receive decoded metadata

```
FUNCTION FindFeed(db AS DATABASE, exchange AS STRING) AS ARRAY
RETURN db.Query( ;
  "SELECT rec_no, symbol, ltp, metadata " + ;
  "FROM imported_feed " + ;
  "WHERE metadata->'exchange' = ? " + ;
  "ORDER BY rec_no", ;
  [exchange])
ENDFUNC
```

CHECKED + PROTOCOL TESTED - PostgreSQL result conversion suite

Table 13. PostgreSQL result mapping

PostgreSQL type	Kokum value
bool	LOGICAL
int2, int4, int8	INTEGER
float4, float8	NUMBER
numeric	DECIMAL
date	DATE
timestamp, timestampz	DATETIME
json, jsonb	decoded ARRAY, MAP or scalar

PostgreSQL type	Kokum value
unrecognized OID	STRING

TIME ZONES

Use timestamptz for instants and test the exact textual/temporal conversion expected by your application. Database session timezone and client display rules must be explicit.

Chapter summary

PostgreSQL result conversion preserves useful Kokum types, including decoded JSON. Unknown types fall back to strings rather than failing silently.

Practice and review

44. Read a JSONB column and access one nested map value.
45. Test numeric values that exceed ordinary floating-point precision.

CHAPTER 21

RETURNING, transactions and diagnostics

Use server features without hiding provider semantics

LEARNING OBJECTIVES

retrieve generated rows with RETURNING; combine RETURNING with transactions; inspect timing and server properties.

PostgreSQL commonly returns generated identifiers, defaults and trigger-modified columns through RETURNING. Call QueryOne or Query rather than Execute when the statement produces rows. This is more reliable than expecting LastInsertId, which is primarily meaningful for SQLite.

Listing 27. Insert and return the canonical PostgreSQL row

```
FUNCTION CreateCustomer(db AS DATABASE, ;
                        name AS STRING, city AS STRING)
RETURN db.QueryOne( ;
    "INSERT INTO customers(name, city) VALUES (?, ?) " + ;
    "RETURNING id, name, city, created_at", ;
    [name, city])
ENDFUNC
```

CHECKED + PROTOCOL TESTED - query-row conversion

Listing 28. Transactional insert with audit row

```
dbPostgres.Begin()
TRY
    orderRow = dbPostgres.QueryOne( ;
        "INSERT INTO orders(customer_id, total) VALUES (?, ?) " + ;
        "RETURNING id, customer_id, total", ;
        [customerId, total])
    dbPostgres.Execute( ;
        "INSERT INTO audit_log(action, entity_id) VALUES (?, ?)", ;
        ["order.created", orderRow["id"]])
    dbPostgres.Commit()
CATCH error
    dbPostgres.Rollback()
    ? STR(error)
ENDTRY
```

CHECKED - DATABASE transaction syntax and PostgreSQL statement shapes

After requests, LastElapsedMs and RequestCount support lightweight instrumentation. ServerVersion and BackendPid help correlate application behavior with PostgreSQL logs and administrative views.

Chapter summary

Use RETURNING whenever PostgreSQL must provide generated or canonical row values. Transactions and runtime diagnostics make service behavior observable and atomic.

Practice and review

46. Return a trigger-generated updated_at value from an UPDATE statement.
47. Log BackendPid and LastElapsedMs when a query exceeds an application threshold.

PART 4

TigerDB: server and router connectivity

The TigerDB provider uses the public length-prefixed JSON protocol and supports direct server or router endpoints, credentials, TLS, mutual TLS and the same high-level query shapes as other DATABASE providers.

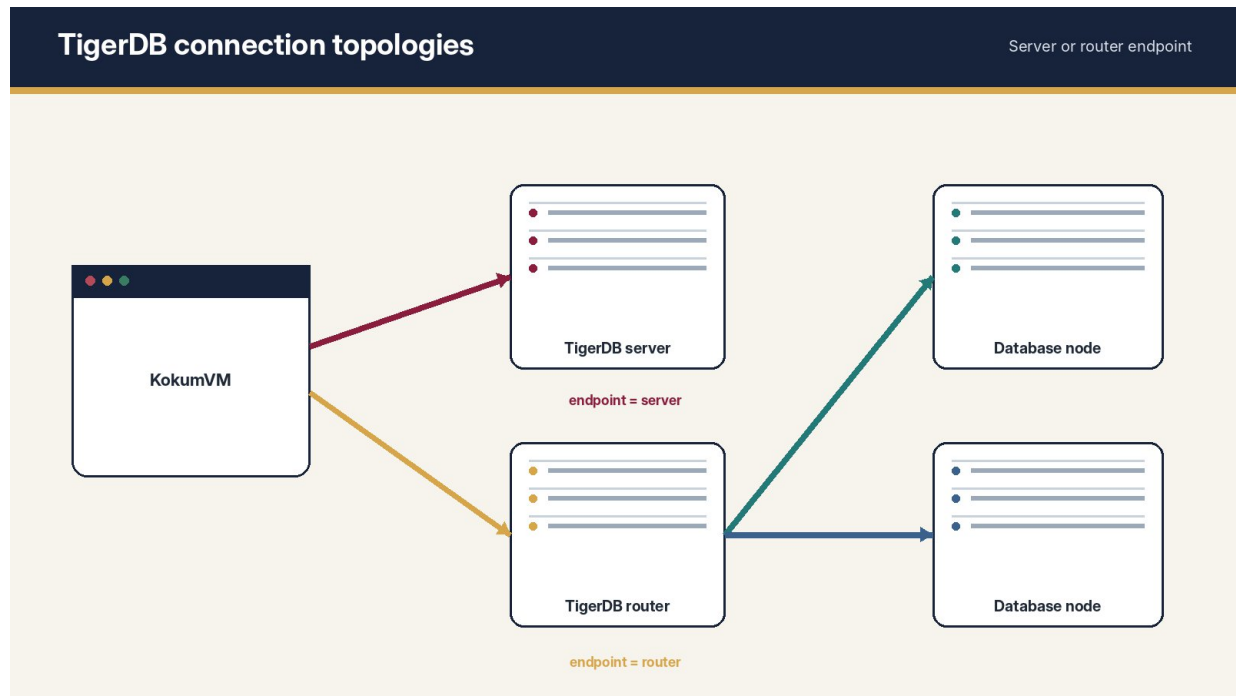


Figure 15. Part 4 overview: TigerDB: server and router connectivity

Chapters in this part

- 22. TigerDB server and router architecture
- 23. Secure TigerDB connection configuration
- 24. TigerDB SQL and result sets
- 25. Ping, UseDatabase, Raw and runtime diagnostics
- 26. TLS, mTLS, timeouts and retry boundaries

CHAPTER 22

TigerDB server and router architecture

Choose the endpoint that matches deployment topology

LEARNING OBJECTIVES

distinguish server and router endpoints; understand protocol framing; keep server details outside SQL code.

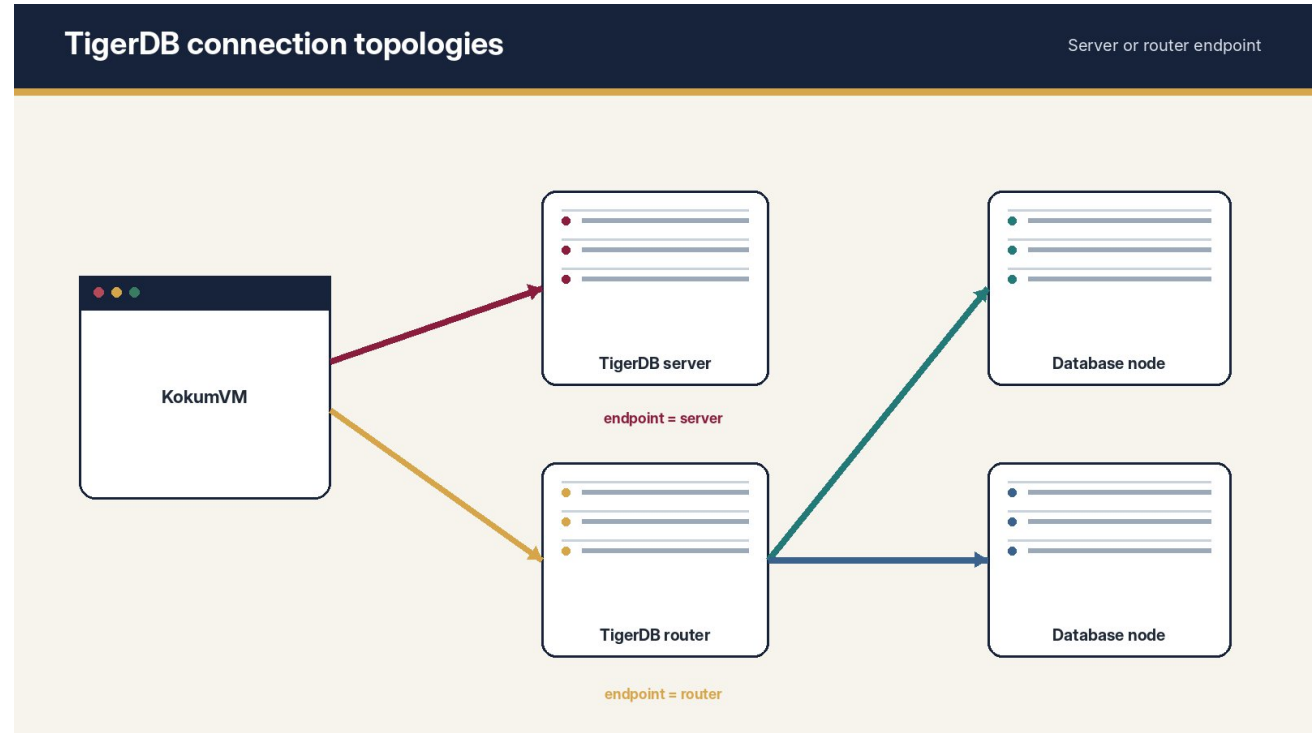


Figure 16. Kokum can connect directly to a TigerDB server or through a TigerDB router

The endpoint property is server or router. A direct server connection targets one TigerDB server. A router endpoint delegates database routing according to the TigerDB deployment. Typical server ports are 9191/9192 and router ports 9292/9293, but the actual server configuration is authoritative.

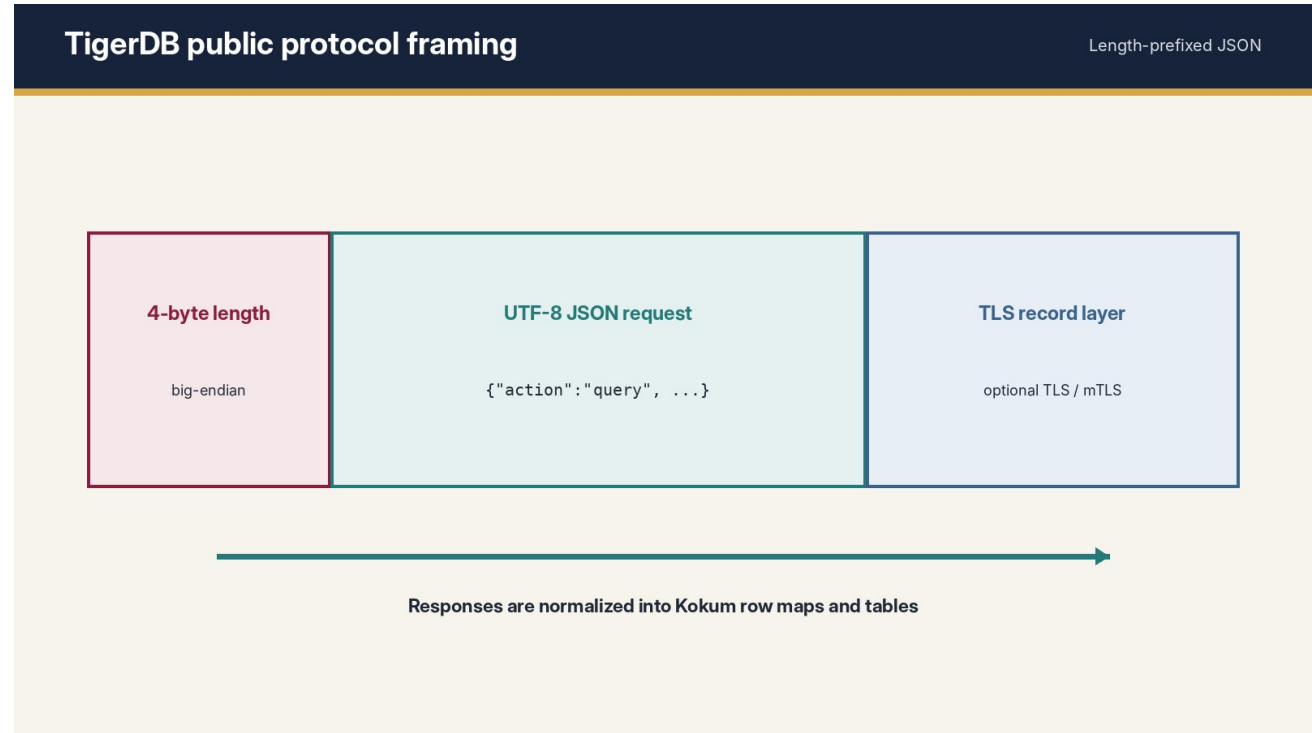


Figure 17. TigerDB requests and responses use a four-byte big-endian length followed by UTF-8 JSON, optionally inside TLS

Listing 29. Inspect TigerDB endpoint configuration at runtime

```
PROCEDURE ShowTigerEndpoint(db AS DATABASE)
? "Endpoint: " + db.Endpoint
? "Host: " + db.Host
? "Port: " + STR(db.Port)
? "Database: " + db.Database
? "TLS: " + STR(db.TLS)
ENDPROC
```

CHECKED + PROTOCOL TESTED - TigerDB provider API suite

CONFIGURATION WINS

Do not copy default port assumptions into code. Read host, port, endpoint and database from the provider configuration so deployments can change without recompilation.

Chapter summary

TigerDB supports direct server and router topologies over its framed JSON protocol. Endpoint selection is configuration, not business logic.

Practice and review

- 48. Describe when a router is preferable to a direct server connection.
- 49. Write an operational log line containing Endpoint, Host, Port and Database.

CHAPTER 23

Secure TigerDB connection configuration

Authenticate the application and verify the peer

LEARNING OBJECTIVES

configure credentials outside source; enable TLS verification; configure optional client certificates.

Listing 30. TLS and mutual-TLS capable TigerDB configuration

```
[database:dbTiger]
provider=tigerdb
module=company.orders.data
endpoint=router
host=router.internal.example
port=9293
database=orders
username=kokum_orders
password=${ENV:KOKUM_TIGERDB_PASSWORD}
tls=true
verify_tls=true
ca_file=${APP_DIR}/tls/tiger-ca.crt
tls_server_name=router.internal.example
client_certificate=${APP_DIR}/tls/client.crt
client_key=${APP_DIR}/tls/client.key
auto_open=false
connect_timeout_ms=10000
query_timeout_ms=30000
max_message_bytes=8388608
```

CONFIG VALIDATED + TLS TESTED - identity and mutual-TLS regression suite

verify_tls=true requires certificate validation. tls_server_name establishes the expected service identity when the connection host is not itself the certificate name. client_certificate and client_key enable mutual TLS when the deployment requires the client to present an identity.

CREDENTIALS AND TLS ARE SEPARATE

A valid username/password authenticates the database account. TLS verifies and protects the transport. Production deployments normally need both.

Chapter summary

TigerDB connection security combines database credentials with TLS peer verification and optional client certificates. All sensitive paths and values belong in external configuration.

Practice and review

50. Create a direct-server configuration with verified TLS but no client certificate.
51. Explain why verify_tls=false is unsuitable for an untrusted network.

CHAPTER 24

TigerDB SQL and result sets

Use the common API over the public protocol

LEARNING OBJECTIVES

execute parameterized TigerDB SQL; receive normalized row maps; use table and scalar result shapes.

TigerDB supports the same Execute, Query, QueryOne, QueryValue and QueryTable calls. Parameter values are converted to escaped SQL literals by a scanner that respects strings, comments and TigerDB dollar-quoted bodies. This protects values while preserving the common question-mark source syntax.

Listing 31. TigerDB query and scalar result through the common API

```
FUNCTION ActiveCustomers(db AS DATABASE, city AS STRING) AS ARRAY
  RETURN db.Query( ;
    "SELECT id, name, city FROM customers " + ;
    "WHERE active = ? AND city = ? ORDER BY name", ;
    [.T., city])
ENDFUNC

count = db.QueryValue( ;
  "SELECT COUNT(*) FROM customers WHERE active = ?", ;
  [.T.] )
```

CHECKED + PROTOCOL TESTED - mock server, parameter conversion and row normalization

The provider normalizes response rows into maps even when the protocol response represents rows as arrays plus a column list. QueryTable constructs headings from response columns or normalized map keys. Execute recognizes common affected-row and insert-id fields in TigerDB responses.

SQL CAPABILITY BELONGS TO THE SERVER

The Kokum provider transports and normalizes requests; the connected TigerDB server version determines SQL grammar and feature support. Test queries against the deployed server build.

Chapter summary

TigerDB uses the same application-facing query shapes while the adapter handles protocol framing, parameter conversion and response normalization.

Practice and review

52. Port a SQLite repository function to TigerDB without changing its return type.
53. Add a test for a zero-row QueryOne response.

CHAPTER 25

Ping, UseDatabase, Raw and runtime diagnostics

Use provider-only operations deliberately

LEARNING OBJECTIVES

test a TigerDB session; switch databases explicitly; send a raw protocol action as JSON text.

Listing 32. TigerDB provider-specific operations and diagnostics

```
PROCEDURE InspectTiger(db AS DATABASE)
  LOCAL health
  LOCAL rawReply

  db.Open()
  health = db.Ping()
  db.UseDatabase("analytics")
  rawReply = db.Raw(JSONENCODE({"action": "ping"}))

  ? JSONENCODE(health)
  ? JSONENCODE(rawReply)
  ? "Requests: " + STR(db.RequestCount)
  ? "Last ms: " + STR(db.LastElapsedMs)
ENDPROC
```

CHECKED + PROTOCOL TESTED - Raw requires a JSON STRING in the current implementation

UseDatabase changes the active database for the connection. Treat that as mutable connection state: call it during controlled initialization rather than unpredictably inside shared repository functions. Raw accepts a JSON string, so build a MAP and call JSONENCODE rather than passing a MAP directly.

RAW IS AN ESCAPE HATCH

Prefer typed provider methods and SQL methods. Raw couples source to protocol details and should be isolated, validated and covered by version-specific tests.

Chapter summary

Ping, UseDatabase and Raw expose TigerDB-specific capabilities. Runtime request counters and timing support diagnostics without changing query results.

Practice and review

54. Wrap Raw in a small versioned adapter function.
55. Explain why changing the active database inside an unrelated query function is risky.

CHAPTER 26

TLS, mTLS, timeouts and retry boundaries

Fail predictably without duplicating committed work

LEARNING OBJECTIVES

set connect and query timeouts; distinguish safe and unsafe retries; use request diagnostics in incident analysis.

connect_timeout_ms bounds session establishment; query_timeout_ms bounds a request; max_message_bytes protects the client from an unreasonable framed response. Timeouts are operational limits, not transaction semantics. After an uncertain network failure, the client may not know whether the server committed a write.

Listing 33. Bounded retry for an idempotent TigerDB read

```
FUNCTION LoadCustomerWithRetry(db AS DATABASE, id AS INTEGER)
  LOCAL attempt AS INTEGER

  FOR attempt = 1 TO 3
    TRY
      RETURN db.QueryOne( ;
        "SELECT id, name, city FROM customers WHERE id = ?", ;
        [id])
    CATCH error
      IF attempt = 3
        ? "Read failed: " + STR(error)
        RETURN NULL
      ENDIF
      SLEEP(attempt * 200)
    ENDTRY
  NEXT
  RETURN NULL
ENDFUNC
```

CHECKED - control flow, query API and timeout error handling

DO NOT RETRY WRITES BLINDLY

A timed-out INSERT may have committed. Use idempotency keys, unique business identifiers or an application-specific reconciliation query before retrying a non-idempotent write.

When diagnosing failures, log the operation category, RequestCount, LastElapsedMs, endpoint identity and a correlation identifier. Do not log passwords, private-key material or unredacted personal data.

Chapter summary

Timeouts bound resource use; retry policy depends on idempotency and uncertainty. TLS tests confirm both server identity and optional client identity behavior.

Practice and review

56. Classify SELECT by primary key, INSERT payment and SET active=false as safe or unsafe to retry automatically.
57. Design an idempotency key for order creation.

PART 5

Remote FlatDB: authenticated portable storage

Remote FlatDB provides a compact remote database service backed by one database file. Its direct K* API is well suited to small deployments and console or service applications that do not need the full DATABASE component model.

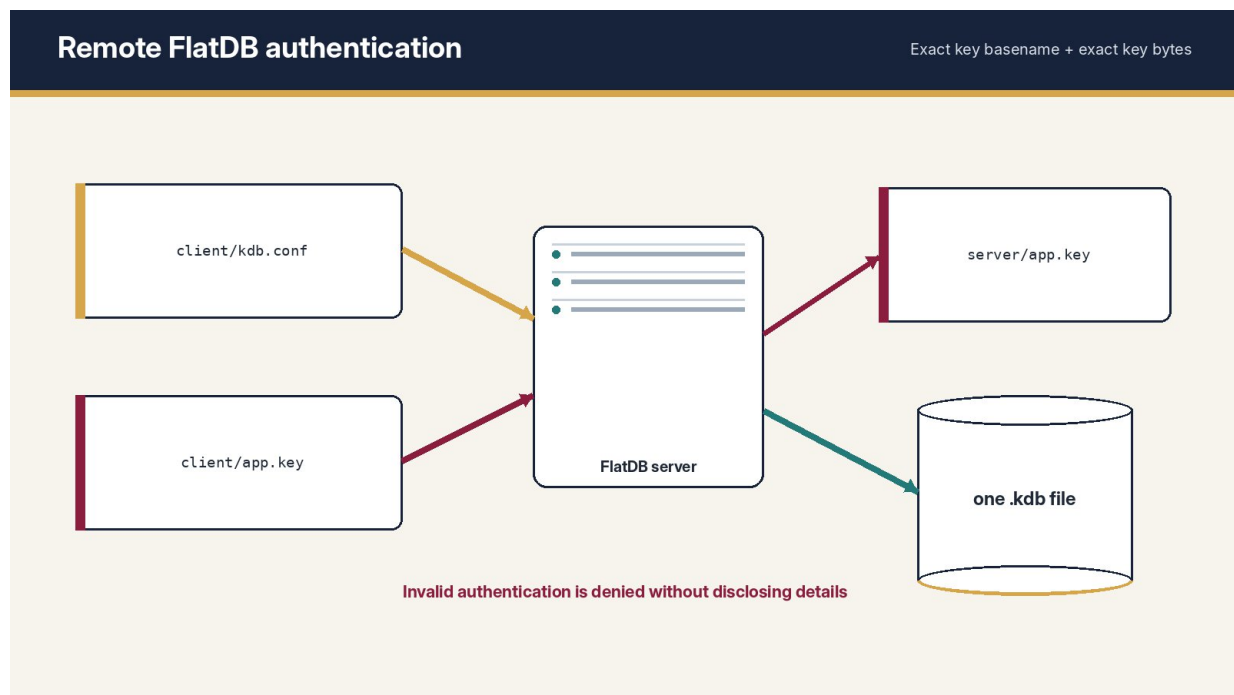


Figure 18. Part 5 overview: Remote FlatDB: authenticated portable storage

Chapters in this part

- 27. Remote FlatDB server, key authentication and kdb.conf
- 28. Remote FlatDB CRUD and result rows
- 29. Path resolution, concurrency and persistence

CHAPTER 27

Remote FlatDB server, key authentication and kdb.conf

Connect only when both sides possess the same key

LEARNING OBJECTIVES

configure kdb.conf; understand exact key matching; test KCONNECTED after KCONNECT.

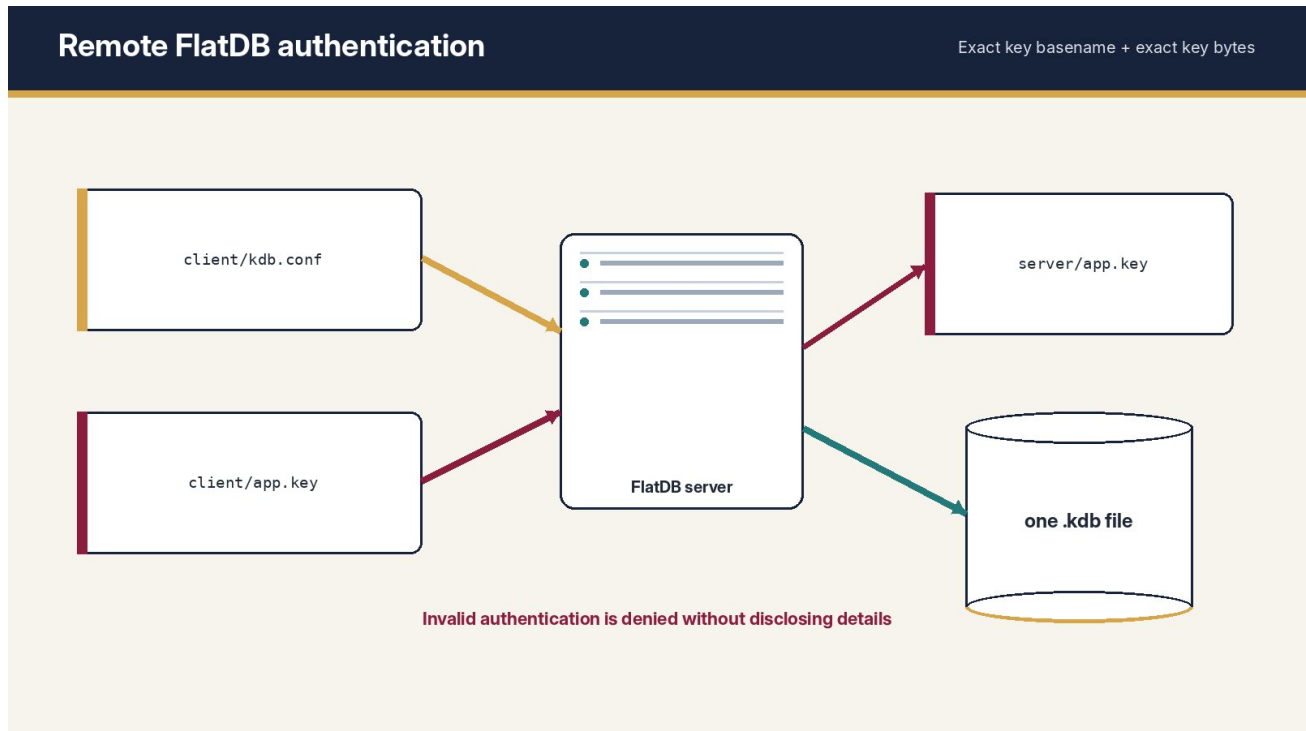


Figure 19. Remote FlatDB requires an exact key filename and exact key contents on client and server

KCONNECT reads a configuration file containing host, port, database filename, key_file and bounded timeout/message settings. Authentication succeeds only when the key basename and bytes match the server configuration. Invalid authentication is denied quietly so the server does not reveal which part was wrong.

Listing 34. Remote FlatDB kdb.conf

```

host=127.0.0.1
port=9437
database=customer_demo.kdb
key_file=customer_demo.key
connect_timeout_ms=5000
request_timeout_ms=30000
max_message_bytes=16777216

```

EXECUTED - authenticated local server regression

Listing 35. Connect and verify Remote FlatDB state

```

KCONNECT USING kdb.conf
IF .NOT. KCONNECTED()
    ? "Remote database is unavailable."
    RETURN 1
ENDIF

```

EXECUTED - interpreter, IR, bytecode and external KokumVM

KEY HANDLING

Generate a strong random binary key, restrict its permissions, distribute it through a secure channel and never commit it to source control.

Chapter summary

Remote FlatDB uses a small external configuration and possession of an exact shared key. Always test KCONNECTED before issuing requests.

Practice and review

58. Create a deployment layout containing the executable, kdb.conf and a protected key file.
59. Explain why an authentication failure intentionally provides little detail.

CHAPTER 28

Remote FlatDB CRUD and result rows

Execute SQL through the direct K* API

LEARNING OBJECTIVES

create tables and indexes; run CRUD statements; process KQUERY row maps.

Listing 36. Complete Remote FlatDB CRUD program

```
FUNCTION Main AS INTEGER
  LOCAL rows AS ARRAY

  KCONNECT USING kdb.conf
  IF .NOT. KCONNECTED()
    RETURN 1
  ENDIF

  KEXECUTE("CREATE TABLE IF NOT EXISTS customers " + ;
    "(id INTEGER PRIMARY KEY, name TEXT, city TEXT)")
  KEXECUTE("CREATE INDEX IF NOT EXISTS idx_customers_city " + ;
    "ON customers(city)")
  KEXECUTE("INSERT INTO customers(id, name, city) " + ;
    "VALUES (1, 'Anil Jagtap', 'Ratnagiri)")

  rows = KQUERY("SELECT id, name, city FROM customers ORDER BY id")
  ? JSONENCODE(rows)

  KEXECUTE("UPDATE customers SET city='Pune' WHERE id=1")
  KEXECUTE("DELETE FROM customers WHERE id=1")
  KDISCONNECT()
  RETURN 0
ENDFUNC
```

EXECUTED - live server, interpreter, IR, bytecode and KokumVM parity

KEXECUTE returns a mutation count where applicable. KQUERY returns an ARRAY of MAP rows. The direct API does not currently accept the common DATABASE parameter ARRAY, so applications must avoid inserting untrusted strings into SQL. Validate and normalize inputs or isolate carefully reviewed quoting helpers until a parameter API is available.

INPUT SAFETY

The example contains fixed literals. Do not substitute raw request values into these SQL strings. Convert numeric identifiers through INT/VAL and use strict allow-lists; treat arbitrary text values as requiring a reviewed safe encoding strategy.

Chapter summary

Remote FlatDB offers direct SQL execution and array-of-map query results. Its compact API is easy to deploy, but input construction requires special care.

Practice and review

60. Add a KQUERY that selects one numeric identifier after strict conversion.
61. Explain why the common DATABASE parameter example cannot simply be copied to KEXECUTE.

CHAPTER 29

Path resolution, concurrency and persistence

Make a small remote service predictable

LEARNING OBJECTIVES

understand relative config lookup; use independent clients safely; verify persistence after restart.

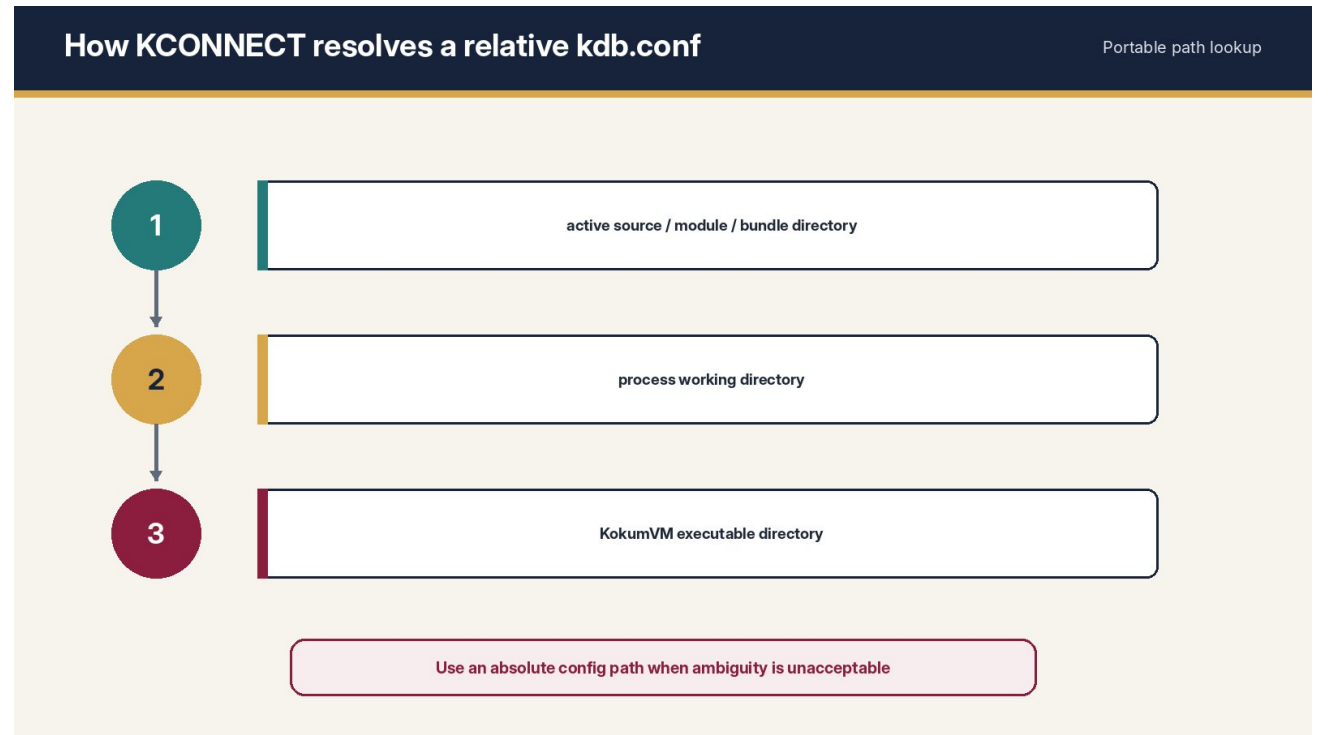


Figure 20. Relative kdb.conf lookup order used by KCONNECT

When kdb.conf is relative, Kokum checks the active source/module/bundle directory, then the process working directory, then the KokumVM executable directory. An absolute path removes ambiguity. The tested server supports multiple independent clients and persists committed data in the configured database file.

Listing 37. Use an absolute configuration path and verify persistent rows

```
PROCEDURE ReconnectAndVerify
  LOCAL rows AS ARRAY

  KCONNECT USING "/opt/kokum/customer-service/kdb.conf"
  IF KCONNECTED()
    rows = KQUERY( ;
      "SELECT id, name FROM customers ORDER BY id"
      ? "Rows after restart: " + STR(LEN(rows))
    )
    KDISCONNECT()
  ENDIF
ENDPROC
```

CHECKED + EXECUTED BEHAVIOR - 24-client concurrency and restart persistence regression

OPERATIONAL FILES

Back up the database file and the server configuration, but protect the authentication key separately. Test server restart and restore while clients use bounded timeouts.

Chapter summary

Config lookup is deterministic, clients are independent, and data persists across server restart. Absolute paths simplify managed service deployments.

Practice and review

62. Predict which kdb.conf is used when copies exist beside the bundle and in the working directory.
63. Write a restart test that inserts, stops the server, restarts it and re-queries the row.

PART 6

Advanced application architecture

The final part moves beyond individual statements. It covers provider-neutral JSON import, module boundaries, asynchronous execution, HTTP services, verification and production operations.

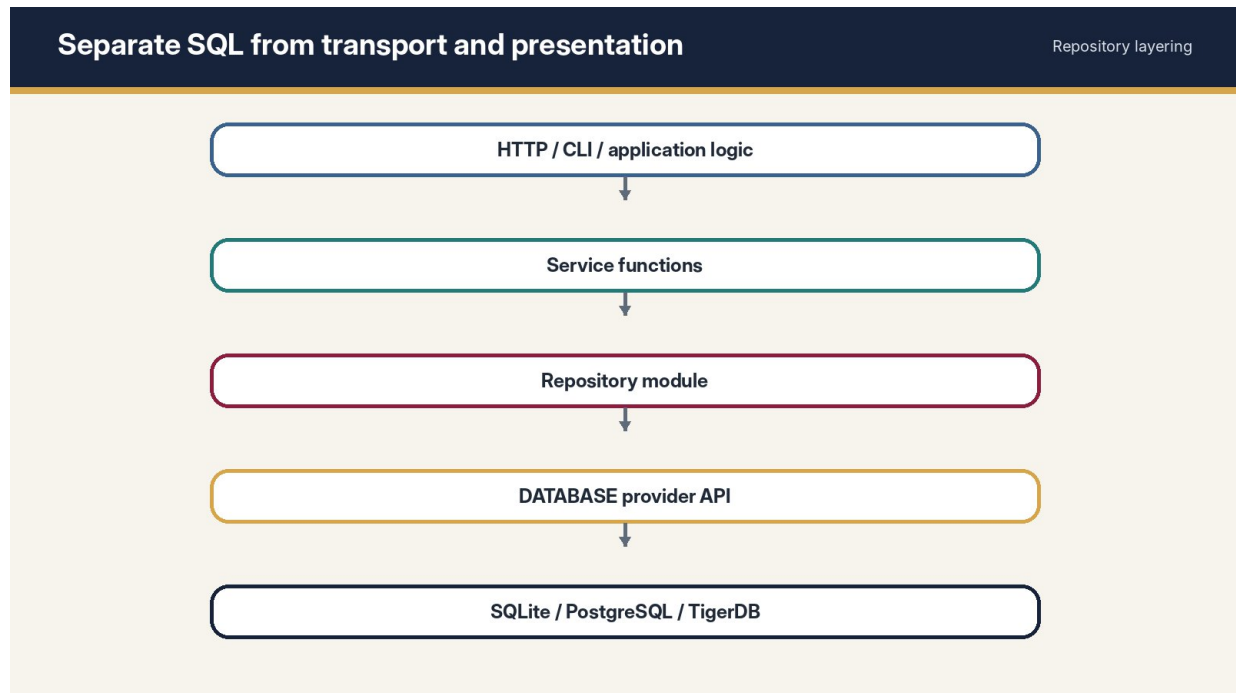


Figure 21. Part 6 overview: Advanced application architecture

Chapters in this part

- 30. Creating tables from JSON
- 31. Repository modules and reusable data access
- 32. Async tasks, threads and connection ownership
- 33. Database-backed HTTP microservices
- 34. Testing, observability, security and deployment

CHAPTER 30

Creating tables from JSON

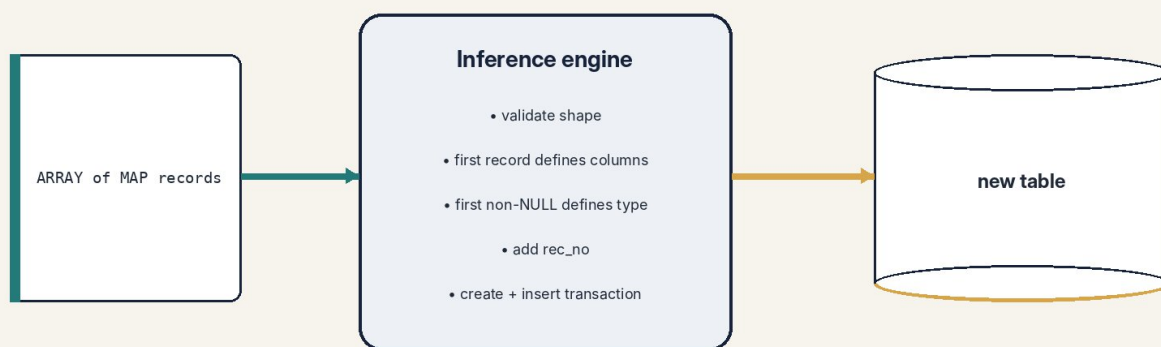
Infer a relational table from unfamiliar records

LEARNING OBJECTIVES

use CREATE TABLE ON DATA FROM JSON; understand inference rules and limits; choose schema-only mode.

CREATE TABLE ... ON DATA ... FROM JSON

Inference and atomic load



SCHEMA ONLY creates the inferred table without inserting records

Figure 22. Kokum validates JSON-shaped data, infers columns and loads rows transactionally

CREATE TABLE ... ON DATA ... FROM accepts a MAP, ARRAY of MAP records or JSON string. Kokum adds rec_no as the first primary key, uses the first record to define field names and order, and uses the first non-NULL value to infer each type. Later missing fields become NULL; later new fields are rejected.

Listing 38. Infer and load a table from an ARRAY of MAP records

```

payload = [ ;
  {"symbol": "NIFTY", "ltp": 24852.75, ;
   "volume": 1200, ;
   "metadata": {"exchange": "NSE", "segment": "INDEX"}}; ;
  {"symbol": "BANKNIFTY", "ltp": 53210.10, ;
   "volume": 840, ;
   "metadata": {"exchange": "NSE", "segment": "INDEX"}}; ;
]

CREATE TABLE imported_feed ;
ON DATA dbPostgres ;
FROM payload
  
```

EXECUTED/API TESTED - provider-neutral JSON table suite

Listing 39. Equivalent method call in schema-only mode

```

dbTiger.CreateTableFromJson( ;
  "imported_feed", ;
  payload, ;
  .T.) && schema only
  
```

CHECKED - method signature and JSON inference tests

Table 14. Selected JSON-table safety limits

Limit	Phase 4.16.7 bound
Source size	16 MiB
Records	100,000
Columns	1,024
Identifier length	1,024 bytes
Nested JSON value	1 MiB

Chapter summary

JSON table creation is transactional and provider-neutral, with deterministic `rec_no` values and bounded inference. It is useful for controlled imports, not as a substitute for reviewed long-term schema design.

Practice and review

64. Use `SCHEMA ONLY` to inspect an inferred table before loading production data.
65. Explain what happens when record 2 introduces a field absent from record 1.

CHAPTER 31

Repository modules and reusable data access

Make SQL a small, testable application boundary

LEARNING OBJECTIVES

define repository contracts; pass DATABASE resources explicitly; isolate provider-specific SQL.

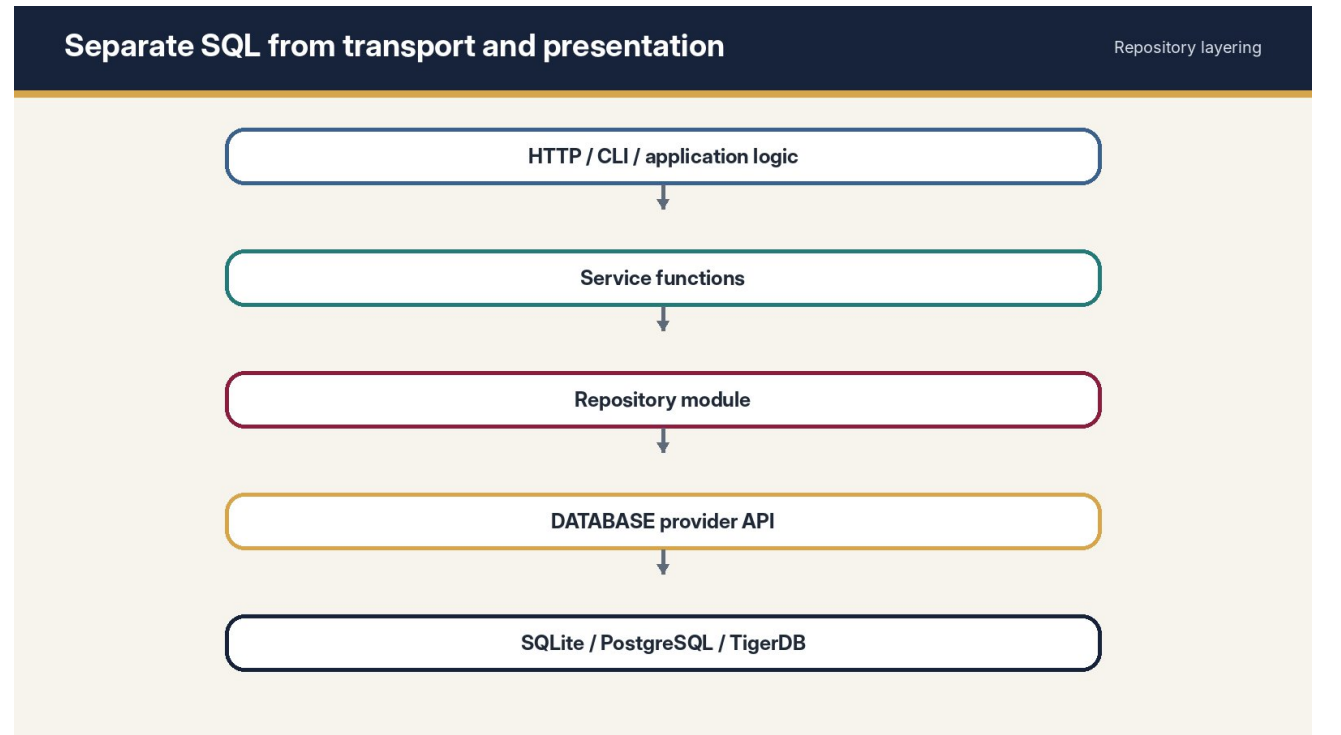


Figure 23. A repository module separates SQL and result mapping from transport and presentation

A repository module should expose business-shaped functions such as FindCustomer, CreateOrder or MonthlyTotals rather than allowing every caller to assemble SQL. Passing a DATABASE resource explicitly makes dependencies visible and allows the same module to be checked with different provider configurations.

Listing 40. A provider-neutral customer repository module

```
MODULE company.customers.repository VERSION "1.0.0"

EXPORT FUNCTION FindById
EXPORT FUNCTION ListActive

FUNCTION FindById(db AS DATABASE, id AS INTEGER)
  RETURN db.QueryOne( ;
    "SELECT id, name, city, active " + ;
    "FROM customers WHERE id = ?", ;
    [id])
ENDFUNC

FUNCTION ListActive(db AS DATABASE) AS ARRAY
  RETURN db.Query( ;
    "SELECT id, name, city FROM customers " + ;
    "WHERE active = ? ORDER BY name", ;
    [.T.])
ENDFUNC
```

CHECKED - module syntax and DATABASE methods passed semantic validation

CONTRACT FIRST

Document whether a repository returns NULL, an empty ARRAY, an affected count or an error for every edge case. Stable contracts matter more than hiding every line of SQL.

Chapter summary

Repository modules centralize parameter binding, SQL aliases, result shapes and provider differences. Explicit DATABASE parameters keep dependencies and tests clear.

Practice and review

66. Create an orders repository with FindById and ListForCustomer functions.
67. Identify one query that should live in a PostgreSQL-specific repository extension.

CHAPTER 32

Async tasks, threads and connection ownership

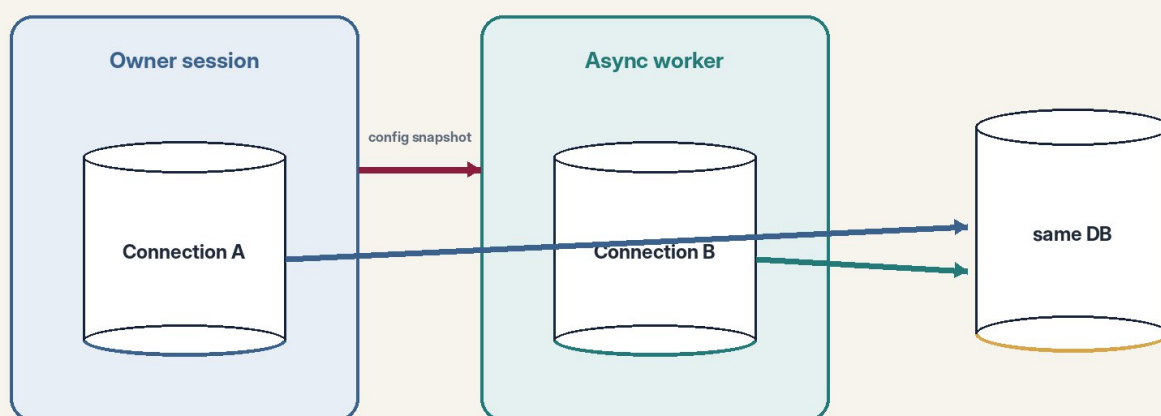
Parallel work gets configuration copies, not shared native handles

LEARNING OBJECTIVES

understand DATABASE transfer to async tasks; avoid cross-connection transaction assumptions; bound parallel query work.

Async database work uses independent connections

No shared native handle



A transaction begun on Connection A does not include Connection B

Figure 24. An async worker reconstructs an independent connection from provider configuration

A DATABASE value crosses an async boundary as provider configuration. The worker creates its own provider connection and native handle. Arrays, maps and other transferable values are copied. This avoids unsynchronized native connection sharing, but it also means owner and worker transactions are independent.

Listing 41. Run a database query in an async worker

```

ASYNC FUNCTION LoadCustomers(db AS DATABASE) AS ARRAY
  RETURN db.Query( ;
    "SELECT id, name, city FROM customers ORDER BY id")
ENDFUNC

PROCEDURE Refresh(db AS DATABASE)
  LOCAL task AS TASK
  LOCAL rows AS ARRAY

  task = LoadCustomers(db)
  rows = AWAIT task
  ? "Loaded: " + STR(LEN(rows))
ENDPROC

```

CHECKED + EXECUTED - async SQLite project uses an independent worker connection

TRANSACTION BOUNDARY

Do not Begin on the owner connection and expect an async worker query or write to participate. Keep one atomic unit of work on one connection and one execution path.

Parallel queries can improve latency only when the provider, server and workload benefit. Bound concurrency; an unbounded task burst can exhaust PostgreSQL sessions, overload TigerDB or create SQLite write contention.

Chapter summary

Kokum deliberately avoids sharing native database handles between threads. Async workers reconstruct independent connections from configuration.

Practice and review

68. Run two independent read-only reports in parallel and await both.
69. Explain why an async write inside an owner transaction breaks atomicity.

CHAPTER 33

Database-backed HTTP microservices

Turn repository results into bounded HTTP responses

LEARNING OBJECTIVES

connect service lifecycle to database lifecycle; validate route values before SQL; map rows and errors to HTTP responses.

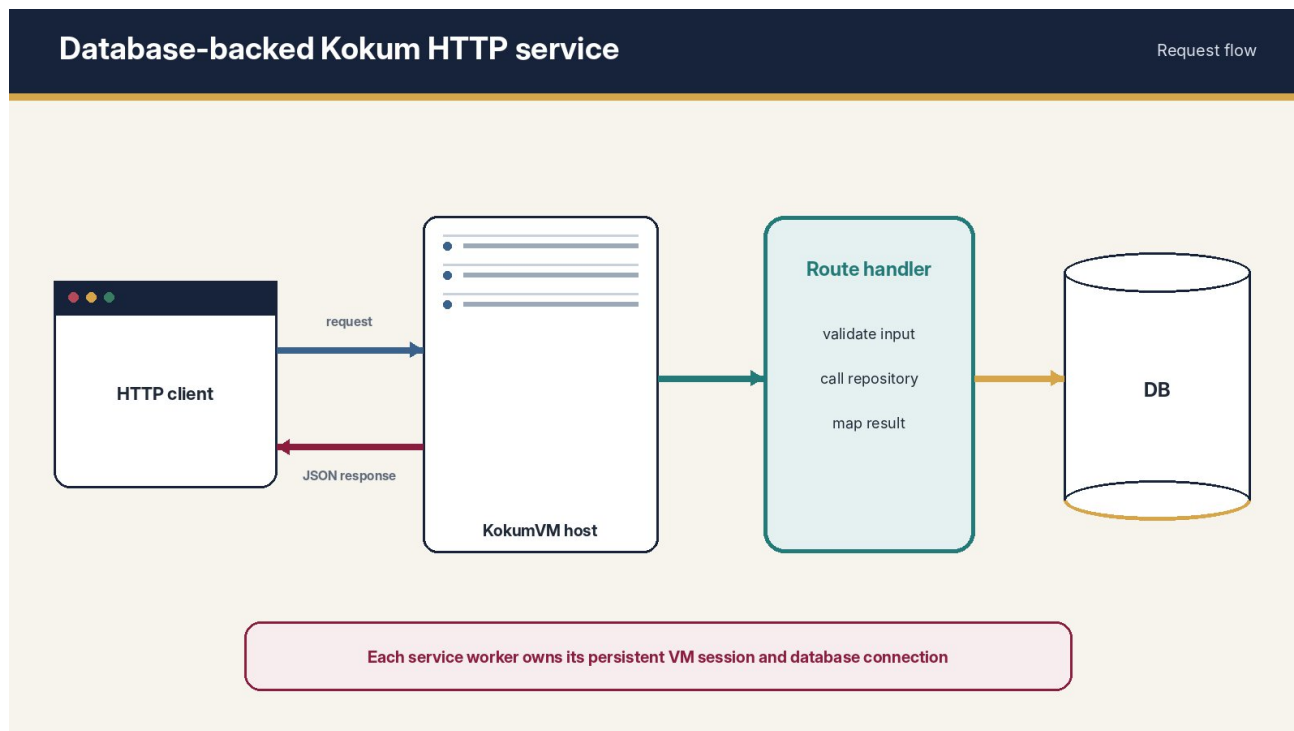


Figure 25. A Kokum HTTP route handler validates input, calls a repository and returns JSON

kokumvm serve hosts a compiled module or application bundle with a bounded request queue and persistent worker sessions. Every worker owns its session and database resources. Startup and shutdown handlers are therefore natural places to establish and release per-worker resources.

Listing 42. Remote FlatDB-backed HTTP route handler with numeric validation

```

FUNCTION CustomerById(request AS MAP) AS MAP
  LOCAL id AS INTEGER
  LOCAL rows AS ARRAY

  id = INT(VAL(request["Params"]["id"]))
  rows = KQUERY( ;
    "SELECT id, name, city FROM customers WHERE id = " + STR(id))

  IF LEN(rows) = 0
    RETURN {"Status": 404, ;
      "Json": {"error": "customer_not_found"}}
  ENDIF
  RETURN {"Status": 200, "Json": rows[1]}
ENDFUNC

```

PROJECT CHECKED + BUNDLE VERIFIED - backend service and FlatDB APIs

Listing 43. Route configuration for the handler

```

[route customer]
method = GET
path = /customers/:id
handler = CustomerById
parameters = 1

```

CONFIG VALIDATED - Kokum backend service configuration

PUBLIC DEPLOYMENT

The Phase 4.15 foundation HTTP host should sit behind a reviewed reverse proxy for public HTTPS, rate limits and production middleware. Never expose database credentials or raw provider errors in an HTTP response.

Chapter summary

Kokum services use bounded workers and persistent sessions. Validate input, call a repository, map not-found and validation cases explicitly, and keep provider errors out of public responses.

Practice and review

70. Add a 400 response for a non-positive customer id.
71. Design a startup health check that fails readiness when the database is unavailable.

CHAPTER 34

Testing, observability, security and deployment

Treat database behavior as an operational contract

LEARNING OBJECTIVES

build a layered verification plan; log useful diagnostics safely; deploy code, configuration and secrets separately.

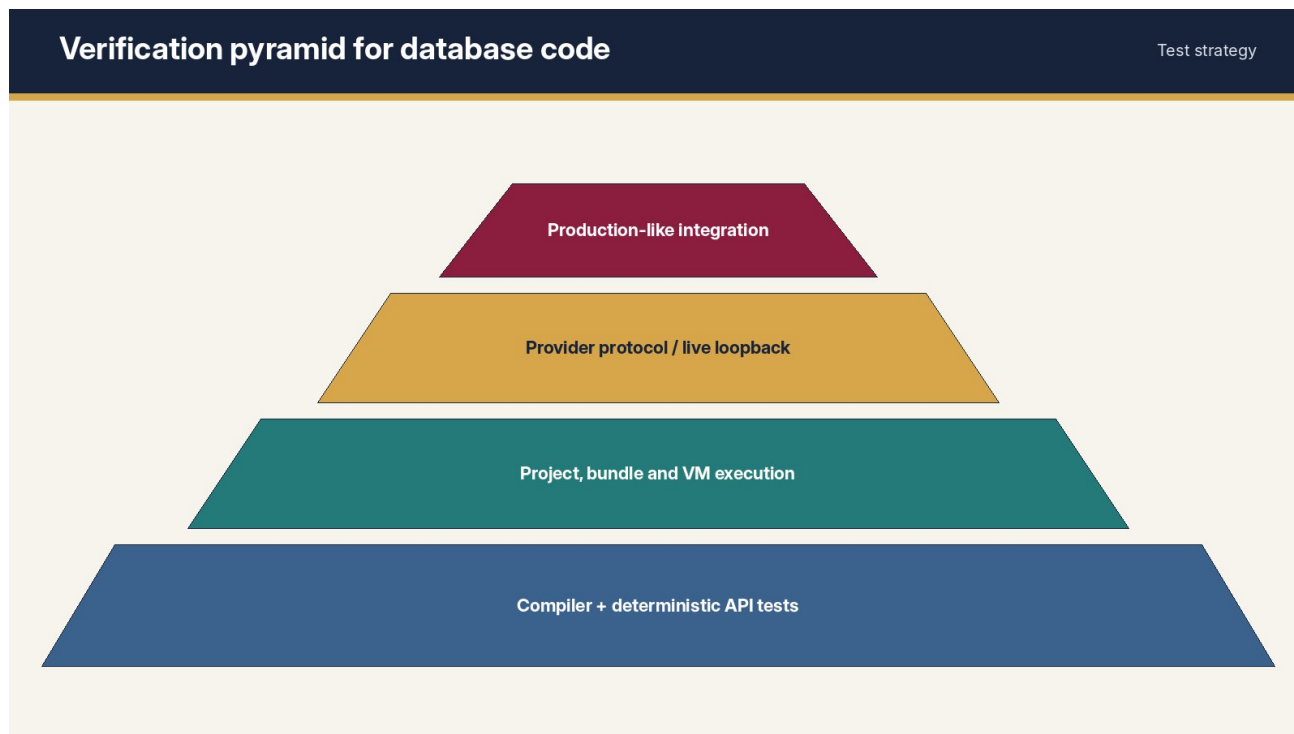


Figure 26. A database test strategy grows from deterministic API checks to production-like integration

Database code should be tested at several levels. Compiler checks catch type and method errors. Provider API tests exercise conversion and error paths deterministically. Project and bundle tests verify generated configuration and runtime integration. Live tests expose permissions, SQL dialect, collation, network and server-version behavior.

Listing 44. Database verification commands used for this book

```
# Repeatable Phase 4.16.7 database verification
make database-test
make json-table-test
make postgresql-test
make tigerdb-test
make flatdb-test

./kokumc project check examples/kokum_022_sqlite/kokum.toml
./kokumc project check examples/kokum_029_postgresql/kokum.toml
./kokumc project check examples/kokum_023_tigerdb/kokum.toml
```

EXECUTED - all listed targets and project checks passed

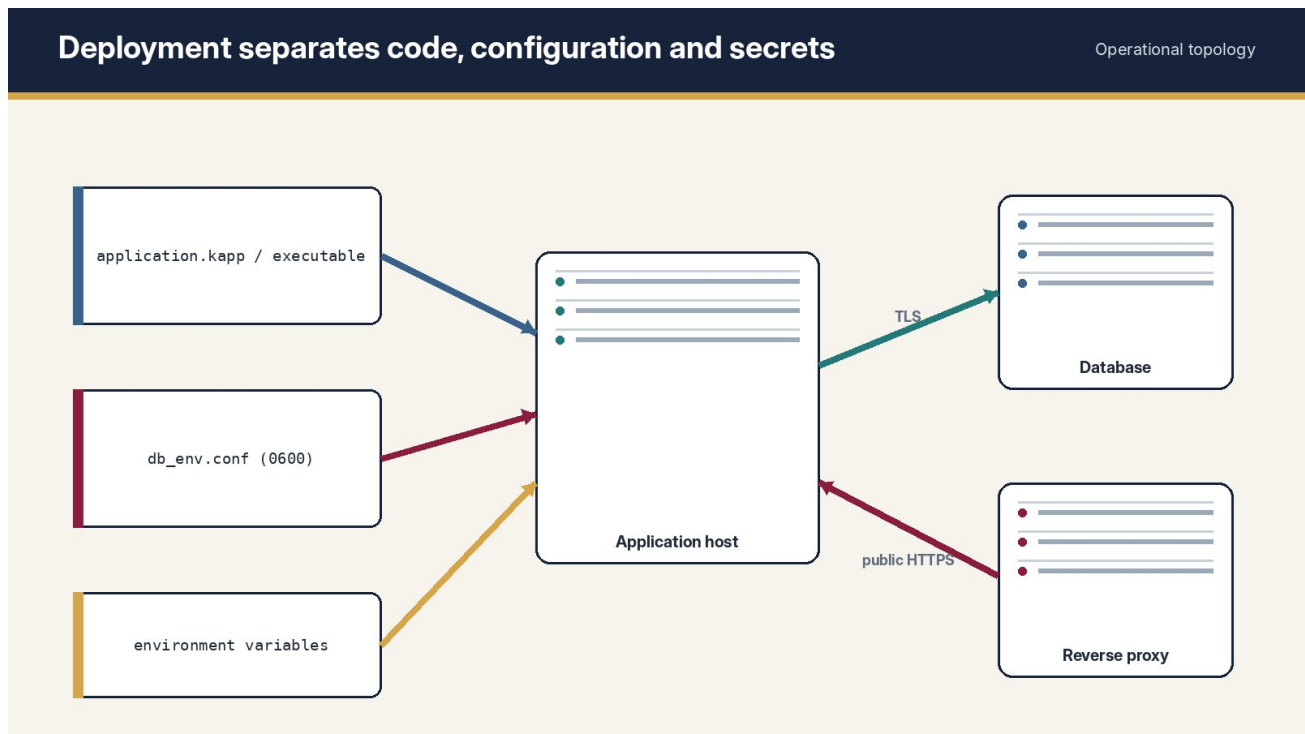


Figure 27. Deploy application code, external configuration and injected secrets as separate concerns

Operational signals

- Connection failures by provider and endpoint.
- Request duration and slow-query threshold events.
- Affected-row mismatches and transaction rollbacks.
- Pool/session pressure or SQLite busy errors.
- Migration version, backup status and restore test date.
- Remote RequestCount and LastElapsedMs where available.

LOG SAFELY

Log statement identity, duration, affected count, correlation id and sanitized error class. Do not log passwords, keys, full connection strings, access tokens or unrestricted personal data.

Chapter summary

Reliable database programming includes repeatable tests, useful telemetry, least privilege, external configuration, backups, migrations and rehearsed recovery.

Practice and review

72. Create a CI job that runs deterministic provider tests and a staging job that runs live integration tests.
73. Write a deployment checklist covering secrets, TLS, migrations, backup and rollback.

APPENDIX A

DATABASE API quick reference

The following methods and properties describe the Phase 4.16.7 DATABASE resource. A provider may reject a provider-only method with a catchable error.

Table 15. DATABASE methods

Member	Return / behavior	Providers
Open()	LOGICAL true; opens the configured connection	SQLite, PostgreSQL, TigerDB
Close()	NULL; rolls back an unfinished transaction	All DATABASE providers
Begin(), Commit(), Rollback()	NULL; explicit connection-local transaction	All DATABASE providers
Execute(sql [, params])	Affected row count where reported	All DATABASE providers
Query(sql [, params])	ARRAY of MAP rows	All DATABASE providers
QueryOne(sql [, params])	MAP or NULL	All DATABASE providers
QueryValue(sql [, params])	First cell or NULL	All DATABASE providers
QueryTable(sql [, params] [, headings])	Two-dimensional ARRAY	All DATABASE providers
CreateTableFromJson(name, source [, schemaOnly])	Creates and optionally loads inferred table	SQLite, PostgreSQL, TigerDB
Ping()	Provider response MAP/value	PostgreSQL, TigerDB
UseDatabase(name)	Changes active database	TigerDB
Raw(jsonString)	Sends raw public-protocol request	TigerDB

Table 16. DATABASE properties

Property	Meaning	Availability
Id	Stable component identifier	All
Provider	Canonical provider name	All
IsOpen	Current connection state	All
LastInsertId	Last provider insert identifier when supported	All property; chiefly SQLite
Changes	Most recent mutation count	All
File	SQLite database file	SQLite
Host, Port, Database	Remote endpoint identity	PostgreSQL, TigerDB
RequestCount, LastElapsedMs	Remote request diagnostics	PostgreSQL, TigerDB
Endpoint, TLS	TigerDB topology/security state	TigerDB
Schema, SSLMode, ApplicationName, ReadOnly	PostgreSQL session configuration	PostgreSQL
ServerVersion, BackendPid	PostgreSQL server identity	PostgreSQL

APPENDIX B

Configuration key reference

SQLite

Table 17. SQLite db_env.conf keys

Key	Purpose
file	Database file path.
open_mode	readwrite-create, readwrite or readonly.
auto_open	Open during application session initialization.
create_parent_directory	Create missing parent directories.
busy_timeout_ms	Wait bound for lock contention.
journal_mode	DELETE, TRUNCATE, PERSIST, MEMORY, WAL or OFF.
foreign_keys	Enable foreign-key enforcement.
synchronous	OFF, NORMAL, FULL or EXTRA.
initialization_script	SQL used only when creating an absent file.

PostgreSQL

Table 18. PostgreSQL db_env.conf keys

Key	Purpose
host / port / database	Server endpoint and database.
username / password	Database credentials; password should normally be ENV-backed.
schema	Initial search path/schema.
sslmode	disable, allow, prefer, require, verify-ca or verify-full.
sslrootcert / sslcert / sslkey	CA and optional client identity files.
application_name	Visible application session name.
auto_open	Open during session initialization.
connect_timeout_ms / query_timeout_ms	Connection and statement/request bounds.
read_only	Request a read-only session.

TigerDB

Table 19. TigerDB db_env.conf keys

Key	Purpose
endpoint	server or router.
host / port / database	Remote target and initial database.
username / password	TigerDB credentials.
tls / verify_tls	Enable TLS and peer verification.
ca_file / tls_server_name	Trust anchor and expected service identity.
client_certificate / client_key	Optional mutual-TLS identity.
auto_open	Open during session initialization.
connect_timeout_ms / query_timeout_ms	Connection and request bounds.
max_message_bytes	Maximum accepted framed response.

Remote FlatDB

Table 20. *kdb.conf* keys

Key	Purpose
host / port	FlatDB server endpoint; default port is 9437.
database	Remote database filename.
key_file	Client key whose basename and bytes must match the server.
connect_timeout_ms	Connection bound.
request_timeout_ms	Request bound.
max_message_bytes	Maximum framed message size.

APPENDIX C

Result and type mapping reference

Result cardinality

Table 21. Choosing a result method

Need	Method	Caller receives
Many rows	Query	ARRAY of MAP
Zero or one row	QueryOne	MAP or NULL
One cell	QueryValue	scalar or NULL
Grid/export shape	QueryTable	2D ARRAY, optional heading row

JSON table mapping

Table 22. Provider targets for inferred JSON fields

Inferred value	SQLite	PostgreSQL	TigerDB
LOGICAL	INTEGER	BOOLEAN	BOOLEAN
INTEGER	INTEGER	BIGINT / integer mapping	INTEGER
NUMBER	REAL	DOUBLE PRECISION / numeric mapping	DOUBLE
DECIMAL	NUMERIC	NUMERIC	DECIMAL
STRING	TEXT	TEXT	STRING
DATE / DATETIME	TEXT	DATE / timestamp mapping	DATE / DATETIME
ARRAY / MAP	TEXT JSON	JSONB	JSON

SCHEMA REVIEW

Inference is deterministic but not omniscient. Review nullability, lengths, indexes, constraints, precision, collation and long-term compatibility before treating an inferred table as a production schema.

APPENDIX D

Remote FlatDB direct API

Table 23. Remote FlatDB built-ins

Built-in	Contract
KCONNECT(path)	Read kdb.conf, authenticate and establish the client session.
KCONNECT USING path	Statement sugar for KCONNECT(path).
KCONNECTED()	LOGICAL connection state.
KEXECUTE(sql)	Execute SQL and return affected count where applicable.
KQUERY(sql)	Return ARRAY of MAP rows.
KDISCONNECT()	Close the current FlatDB session.

Listing 45. Minimal Remote FlatDB query

```
KCONNECT USING kdb.conf
IF KCONNECTED()
  ? JSONENCODE(KQUERY("SELECT id, name FROM customers ORDER BY id"))
  KDISCONNECT()
ENDIF
```

EXECUTED - live server and all Kokum execution engines

APPENDIX E

Troubleshooting guide

Table 24. Common database problems

Symptom	Likely cause	Action
Open fails immediately	Missing file, refused network, credentials, TLS or configuration lookup	Confirm resolved db_env.conf/kdb.conf, endpoint, permissions and certificate paths.
Parameter count mismatch	Real ? placeholders and ARRAY length differ	Check quoted strings/comments and dynamically appended SQL.
QueryOne returns NULL	No row matched	Treat as expected cardinality or inspect WHERE values.
Unexpected map key	Column label/alias changed	Use explicit aliases and repository contract tests.
SQLite busy/locked	Long writer or concurrent write burst	Shorten transactions, configure busy_timeout, serialize writes where appropriate.
PostgreSQL TLS failure	CA, hostname, sslmode or file permissions	Use verify-full with the correct CA and service name; inspect libpq/server logs.
TigerDB timeout	Network/server load, query cost or response limit	Inspect endpoint, LastElapsedMs, server logs and max_message_bytes.
FlatDB KCONNECTED is false	Config/key mismatch or unavailable server	Verify exact key basename and bytes, path lookup and port.
Live build works, deployment fails	External config/cert/data file absent beside deployment	Stage external files and permissions explicitly.

Catch and report without leaking secrets

Listing 46. Bounded diagnostic handling

```

TRY
    rows = db.Query("SELECT id, name FROM customers ORDER BY id")
CATCH error
    ? "Customer query failed: " + STR(error)
    && Application logs should add correlation id and provider identity,
    && but never passwords, key bytes or private-key contents.
ENDTRY

```

CHECKED - TRY/CATCH and Query syntax

APPENDIX F

Verification record for this edition

The examples and claims in this edition were checked against the final Phase 4.16.7 source tree used to prepare the book. The following matrix distinguishes live execution from deterministic provider testing.

Table 25. Edition verification matrix

Area	Verification performed	Result
Common DATABASE C API	Lifecycle, methods, result shapes, ownership and errors	PASS
SQLite	IDE synchronization, project check, build, bundle, KokumVM execution, database file and standalone deployment	PASS
PostgreSQL	libpq adapter, placeholder scanner, result conversion, errors and component schema; no external live server was used	PASS
TigerDB	Protocol adapter, parameter conversion, row normalization, TLS identity and mutual TLS against deterministic test servers	PASS
Remote FlatDB	Authenticated live local server, CRUD, interpreter/IR/bytecode/VM parity, 24-client concurrency and restart persistence	PASS
JSON table creation	Validation, inference, transaction and provider mappings	PASS
Repository examples	kokumc semantic check with zero warnings	PASS
Database-backed service example	Module check, project check, build, bundle creation and bundle verification	PASS

POSTGRESQL BOUNDARY

The PostgreSQL provider suite passed, but this environment did not use an independently administered live PostgreSQL server. Run the book examples against your staging server, schema, extensions, roles and TLS setup.

Repeatable commands

Listing 47. Core verification commands

```
make database-test
make json-table-test
make postgresql-test
make tigerdb-test
make flatdb-test

./kokumc project check examples/kokum_022_sqlite/kokum.toml
./kokumc project check examples/kokum_029_postgresql/kokum.toml
./kokumc project check examples/kokum_023_tigerdb/kokum.toml
./kokumc project check examples/kokum_029_json_table/kokum.toml
```

EXECUTED - all commands passed while preparing this book

CLOSING NOTE

Build the data boundary deliberately

Good database code is not measured only by whether a query returns rows. It is measured by whether values are bound safely, result contracts are stable, transactions preserve invariants, provider differences are explicit, failures are observable, secrets remain private, and the application can be upgraded and restored with confidence.

Kokum provides a compact foundation: a provider-neutral DATABASE API, typed result conversion, explicit transactions, secure external configuration, direct Remote FlatDB support, JSON table import, asynchronous connection reconstruction and a bounded HTTP service host. The responsibility of the application is to combine those tools into a clear data architecture.

FINAL PRINCIPLE

Keep SQL close to a repository, keep credentials outside code, keep one transaction on one connection, and test the exact deployment you intend to operate.

END OF VOLUME