Gestión del Estado y Persistencia en C/C++

Antonio Espín Herranz

Contenidos

- Conexión a bases de datos relacionales y no relacionales:
  - Integración con **MySQL**, **PostgreSQL** y bases de datos NoSQL como **MongoDB**.
  - Uso de librerías **ORM** como **soci** o **libpqxx**.

- Estrategias de persistencia y acceso rápido a datos:
  - Caching con **Redis** o **Memcached** en aplicaciones C++.
  - Implementación de acceso en tiempo real a grandes volúmenes de datos.

Bases de datos Relacionales SQLite3, MySQL y PostGreSQL

Librerías

- Para trabajar con estas bases de datos tenemos librerías en el gestor de paquetes: vcpkg

- Para instalar:
  - Sqlite3 **vcpkg install slite3**
  - MySQL **vcpkg install libmysql**
  - PostgreSQL **vcpkg install libpq**

- Para integrar en Visual Studio:
  - **vcpkg integrate install**

Problemas con libmysql

- Ir a la carpeta de vcpkg
  - git pull
  - vcpkg update
  - vcpkg upgrade
    - Puede dar un warning y habrá que ejecutar con:
    - vcpkg upgrade --no-dry-run

- Después de actualizar:
  - vcpkg remove libmysql
  - vcpkg remove --outdated
  - vcpkg install libmysql
  - vcpkg integrate install

Librería sqlite3

Prepared Statement

- **Ventajas de usar prepared statements:**
  - **Seguridad**: Evita inyecciones SQL.
  - **Rendimiento**: Puedes reutilizar la sentencia preparada para múltiples inserciones.
  - **Flexibilidad**: Puedes vincular diferentes tipos de datos (texto, enteros, blobs, etc.).

Ejemplo de prepared statement

- **Abrir conexión:**

```
sqlite3* db;
sqlite3_stmt* stmt;
int rc;

// Abrir la base de datos
rc = sqlite3_open("mi_base.db", &db);
if (rc) {
        std::cerr << "No se puede abrir la base de datos: " << sqlite3_errmsg(db) << std::endl;
         return rc;
}
```

Ejemplo de prepared statement

- SQL y preparar la sentencia:

```
// Sentencia SQL con parámetros
const char* sql = "INSERT INTO usuarios(nombre, edad, activo, fecha_registro) VALUES (?, ?, ?, ?);";

rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
if (rc != SQLITE_OK) {
    std::cerr << "Error al preparar la sentencia: " << sqlite3_errmsg(db) << std::endl;
    sqlite3_close(db);
    return rc;
}
```

Ejemplo de prepared statement

- Varios tipos de datos a insertar:

```
// Datos a insertar
  const char* nombre = “Ana";
  int edad = 35;
  bool activo = true; // SQLite no tiene tipo booleano, se usa 0/1
  const char* fecha = "2025-09-03";

  // Vincular parámetros
  sqlite3_bind_text(stmt, 1, nombre, -1, SQLITE_STATIC);      // nombre (TEXT)
  sqlite3_bind_int(stmt, 2, edad);                            // edad (INTEGER)
  sqlite3_bind_int(stmt, 3, activo ? 1 : 0);            // activo (BOOLEAN como INTEGER)
  sqlite3_bind_text(stmt, 4, fecha, -1, SQLITE_STATIC);     // fecha_registro (TEXT)
```

Ejemplo prepared statement

- Ejecutar sentencia y liberar recursos:

```
rc = sqlite3_step(stmt);
if (rc != SQLITE_DONE) {
    std::cerr << "Error al insertar la fila: " << sqlite3_errmsg(db) << std::endl;
} else {
    std::cout << "Fila insertada correctamente." << std::endl;
}

// Liberar recursos
sqlite3_finalize(stmt);
sqlite3_close(db);
```

Ejemplo de prepared statement

- BOOLEAN en SQLite se representa como INTEGER (0 o 1).

- Fechas se suelen guardar como texto en formato ISO (YYYY-MM-DD), aunque también puedes usar INTEGER como timestamp Unix.
- Puedes usar SQLITE_TRANSIENT en lugar de SQLITE_STATIC si los datos no viven más allá del sqlite3_bind_text.
  - ***Ojo con datos en punteros ...***

- Para el tipo REAL de sqlite3: sqlite3_bind_double

Filas afectadas

- Cuando ejecutamos sentencias de insert / delete / update podemos obtener el número de filas afectadas:

- int filasAfectadas = **sqlite3_changes**(conexión)

- Si estamos dentro de una transacción y hemos realizado varias operaciones:
- int filas = **sqlite3_total_changes**(conexion);

Transacciones

- Se envían las instrucciones de SQL:
- Begin transaction
- Commit
- Rollback

- Con la función: sqlite3_exec

Ejemplo 1 de 2

```
// Iniciar transacción
  rc = sqlite3_exec(db, "BEGIN TRANSACTION;", nullptr, nullptr, nullptr);
  if (rc != SQLITE_OK) {
     std::cerr << "Error al iniciar transacción: " << sqlite3_errmsg(db) << std::endl;
     return false;
  }

  // Primera operación
  const char* sql1 = "UPDATE empleados SET activo = 0 WHERE cargo = 'Intern';";
  rc = sqlite3_exec(db, sql1, nullptr, nullptr, nullptr);
  if (rc != SQLITE_OK) {
     std::cerr << "Error en la primera operación: " << sqlite3_errmsg(db) << std::endl;
     sqlite3_exec(db, "ROLLBACK;", nullptr, nullptr, nullptr);
     return false;
  }
```

Ejemplo 2 de 2

```
// Segunda operación
  const char* sql2 = "DELETE FROM empleados WHERE edad > 65;";
  rc = sqlite3_exec(db, sql2, nullptr, nullptr, nullptr);
  if (rc != SQLITE_OK) {
     std::cerr << "Error en la segunda operación: " << sqlite3_errmsg(db) << std::endl;
     sqlite3_exec(db, "ROLLBACK;", nullptr, nullptr, nullptr);
     return false;
  }

  // Confirmar transacción
  rc = sqlite3_exec(db, "COMMIT;", nullptr, nullptr, nullptr);
  if (rc != SQLITE_OK) {
     std::cerr << "Error al confirmar transacción: " << sqlite3_errmsg(db) << std::endl;
     return false;
  }

  std::cout << "Transacción completada con éxito." << std::endl;
```

Librería libmysql vs sqlite3

- Con mysql: diferencias con sqlite3

`sqlite3_prepare_v2`	`mysql_stmt_prepare`
`sqlite3_bind_*`	`mysql_stmt_bind_param` con `MYSQL_BIND`
`sqlite3_step`	`mysql_stmt_execute`
`sqlite3_finalize`	`mysql_stmt_close`
`sqlite3_changes`	`mysql_stmt_affected_rows`
`sqlite3_exec` con callback	`mysql_query` + `mysql_store_result` + `mysql_fetch_row`

MySQL

libmysql

Objetos principales de la librería

Objeto / Tipo	Descripción breve
`MYSQL`	Representa la conexión a la base de datos. Se usa con `mysql_init()` y `mysql_real_connect()`.
`MYSQL_RES`	Contiene el conjunto de resultados de una consulta. Se obtiene con `mysql_store_result()` o `mysql_use_result()`.
`MYSQL_ROW`	Representa una fila del resultado. Se obtiene con `mysql_fetch_row()`.
`MYSQL_FIELD`	Describe los campos (columnas) del resultado. Se obtiene con `mysql_fetch_fields()`.
`MYSQL_STMT`	Representa una consulta preparada. Se usa con `mysql_stmt_prepare()` y `mysql_stmt_execute()`.
`MYSQL_BIND`	Se usa para hacer bind de parámetros o resultados en consultas preparadas.
`MYSQL_TIME`	Estructura para manejar fechas y horas en bind.
`my_bool` / `bool`	Se usa para indicar valores nulos o estados en bind.
`unsigned long`	Se usa para indicar la longitud de campos en bind.

Objetos de la librería

- MYSQL es el punto de partida: conecta con la base de datos.
  - **La conexión**

- MYSQL_STMT te permite preparar y ejecutar consultas con parámetros.
  - **La sentencia**

- MYSQL_BIND se usa para pasar valores a esas consultas o recibir resultados.
  - **Gestión de parámetros en las consultas**

- MYSQL_RES, MYSQL_ROW y MYSQL_FIELD se usan para manejar resultados en consultas no preparadas.
  - **Campos, resultados, etc.**

Conexión y ejecución de consultas

Función	Propósito
`mysql_init()`	Inicializa una estructura `MYSQL` para establecer la conexión.
`mysql_real_connect()`	Conecta al servidor MySQL con los parámetros de host, usuario, contraseña, base de datos y puerto.
`mysql_stmt_init()`	Inicializa una estructura `MYSQL_STMT` para trabajar con sentencias preparadas.
`mysql_stmt_prepare()`	Prepara una sentencia SQL con parámetros para su ejecución posterior.
`mysql_stmt_bind_param()`	Asocia variables C++ a los parámetros de la sentencia preparada.
`mysql_stmt_execute()`	Ejecuta la sentencia preparada con los parámetros vinculados.
`mysql_stmt_bind_result()`	Asocia buffers para recibir los resultados de la consulta.
`mysql_stmt_fetch()`	Recupera fila por fila los resultados de la consulta.
`mysql_stmt_close()`	Libera los recursos asociados a la sentencia preparada.
`mysql_close()`	Cierra la conexión con el servidor MySQL.

Tipos de parámetros

Macro	Tipo de dato en C++	Descripción
`MYSQL_TYPE_TINY`	`char`, `int8_t`	Entero pequeño (1 byte)
`MYSQL_TYPE_SHORT`	`short`, `int16_t`	Entero corto (2 bytes)
`MYSQL_TYPE_LONG`	`int`, `int32_t`	Entero largo (4 bytes)
`MYSQL_TYPE_LONGLONG`	`long long`, `int64_t`	Entero muy largo (8 bytes)
`MYSQL_TYPE_FLOAT`	`float`	Número decimal de precisión simple
`MYSQL_TYPE_DOUBLE`	`double`	Número decimal de doble precisión
`MYSQL_TYPE_STRING`	`char[]`, `std::string`	Cadena de texto (longitud fija o variable)
`MYSQL_TYPE_VAR_STRING`	`char[]`, `std::string`	Cadena de texto (longitud variable)
`MYSQL_TYPE_DATE`	`MYSQL_TIME`	Fecha (sin hora)
`MYSQL_TYPE_DATETIME`	`MYSQL_TIME`	Fecha y hora
`MYSQL_TYPE_TIME`	`MYSQL_TIME`	Hora (sin fecha)
`MYSQL_TYPE_TIMESTAMP`	`MYSQL_TIME`	Marca de tiempo
`MYSQL_TYPE_NULL`	—	Valor nulo

Ejecución de consultas sin parámetros

Función	Propósito
`mysql_init()`	Inicializa la conexión.
`mysql_real_connect()`	Conecta al servidor MySQL.
`mysql_query()`	Ejecuta una consulta SQL directa (sin parámetros).
`mysql_store_result()`	Recupera el conjunto de resultados.
`mysql_fetch_row()`	Itera sobre cada fila del resultado.
`mysql_free_result()`	Libera la memoria del resultado.
`mysql_close()`	Cierra la conexión.

Transacciones

Función	Descripción
`mysql_autocommit()`	Activa o desactiva el modo autocommit. Debe desactivarse para iniciar una transacción manual.
`mysql_commit()`	Confirma (commit) la transacción: aplica todos los cambios realizados.
`mysql_rollback()`	Cancela (rollback) la transacción: revierte todos los cambios realizados.

Registros afectados y último id generado

Función	Descripción
`mysql_stmt_affected_rows()`	Devuelve el número de filas afectadas por una sentencia preparada (`INSERT` , `UPDATE` , `DELETE`).
`mysql_insert_id()`	Devuelve el último valor autogenerado por una columna `AUTO_INCREMENT` tras un `INSERT` .

```
// Ejecutar la sentencia preparada
if (mysql_stmt_execute(stmt)) {
    std::cerr << "Error al ejecutar la sentencia: " << mysql_stmt_error(stmt) << std::endl;
    return 1;
}

// Obtener número de filas afectadas
my_ulonglong filas = mysql_stmt_affected_rows(stmt);
std::cout << "Filas afectadas: " << filas << std::endl;

// Obtener último ID generado (solo si la columna 'id' es AUTO_INCREMENT)
my_ulonglong ultimo_id = mysql_insert_id(conn);
std::cout << "Último ID generado: " << ultimo_id << std::endl;
```

Librerías ORM

soci / libpqxx

Proyectos con C++ 17

Instalación

- **vcpkg install soci** → Con esto no se instala un backend de mysql, faltaría:
- **vcpkg install soci[core,mysql]**

- **vcpkg install libpqxx**

- Para integrar en visual studio:
  - **vcpkg integrate install**

- **soci** es un **ORM** (object relational mapping). Nos abstrae de utilizar el SQL y podemos trabajar con un modelo orientado a objetos.

- **libpqxx** es un **cliente** puro para **postgresql**, pero no es un ORM.

soci

- Es una abstracción de acceso a base de datos que permite escribir código C++ sin preocuparse por el SQL específico del motor.
- Tiene una sintaxis tipo ORM, aunque no es un ORM completo, no te quita el SQL, pero permite realizar las consultas de una forma mas estructurada.
- Soporta múltiples backends: PostgreSQL, MySQL, SQLite, Oracle.
- Puedes usar macros para generar clases que se mapean a tablas

libmqxx

- Es el wrapper oficial en C++ para PostgreSQL.
- No es un ORM, sino una interfaz directa al motor PostgreSQL.
- Ofrece acceso a transacciones, consultas, y manejo de resultados con una API moderna en C++.
- Ideal si quieres control total sobre el SQL y rendimiento.

Comparativa soci vs libpqxx

Característica	SOCI	libpqxx
Tipo	Abstracción DB / semi-ORM	Cliente PostgreSQL puro
Soporte de motores	PostgreSQL, MySQL, SQLite...	Solo PostgreSQL
Nivel de abstracción	Alto (menos SQL explícito)	Bajo (SQL explícito)
Facilidad de uso	Más amigable para ORM	Más técnico y detallado
Instalación vcpkg	✅	✅
Rendimiento	Bueno, pero depende del backend	Excelente para PostgreSQL

SOCI

- SOCI se define como una **biblioteca de acceso a bases de datos para C++**, que:

- Usa **bindings tipo C++** para ejecutar SQL.

- Permite **mapear resultados a variables C++** con soci::into() y pasar parámetros con soci::use().

- Soporta múltiples backends (MySQL, PostgreSQL, SQLite...).

- Pero **no**:
  - Genera SQL automáticamente.
  - Mapea clases C++ a tablas de forma automática.
  - Hace migraciones, validaciones o relaciones entre entidades.

Objetos principales de SOCI

- **soci::session**
  - El objeto principal que representa una conexión activa con la base de datos.
    - soci::session sql(soci::mysql, "db=empresa3 user=antonio password=antonio host=127.0.0.1 port=3307");
  - Se pasa por referencia a las clases DAO (como EmpleadoRepository) para ejecutar consultas.

Objetos principales de SOCI

- **soci::into**
  - Se usa para recibir resultados de una consulta SQL en variables C++.
    - sql << "SELECT nombre FROM empleados WHERE id = :id", **soci::use**(id), **soci::into**(nombre);

- **soci::use**
  - Para inyectar un parámetro en una consulta parametrizada.

- **soci::indicator**
  - Permite detectar si un valor devuelto por la base de datos es NULL.

    - soci::indicator ind;
    - sql << "SELECT id FROM empleados WHERE id = :id", soci::use(id), soci::into(emp.id, ind);
    - if (ind == soci::i_null) { /* manejar ausencia de datos */ }

Objetos principales de SOCI

- **soci::rowset<soci::row>**
  - Permite recorrer múltiples resultados de una consulta SELECT.
    - soci::rowset<soci::row> rs = sql.prepare << "SELECT id, nombre, cargo FROM empleados";
    - for (const auto& r : rs) { /* recorrer resultados */ }
  - Listados y consultas que devuelven varias filas.

Objetos principales de SOCI

- **#include <optional>**
- **std::optional<T>**

- Parte de **C++17**, representa un valor que puede existir o no.
  - std::optional<Empleado> resultado = repo.recuperarEmpleado(5);
  - if (resultado) { std::cout << resultado->nombre; }

  - **std::nullopt**
    - Constante que representa un std::optional vacío.
    - return std::nullopt;

Transacciones

```
sql.begin(); // Inicia la transacción

try {
   sql << "INSERT INTO empleados(nombre, cargo) VALUES(:nombre, :cargo)",
      soci::use("Luis"), soci::use("Contable");

   sql << "UPDATE empleados SET cargo = 'Gerente' WHERE id = 5";

   sql.commit(); // Confirma los cambios
} catch (...) {
   sql.rollback(); // Revierte todo si hay error
   throw;
}
```

Transacciones

- **soci::transaction**

Método	Control manual	Rollback automático	Recomendado para
`sql.begin()`	✅ Sí	❌ No	Procesos más complejos
`soci::transaction`	❌ No	✅ Sí	Operaciones simples y seguras

Recuperar el id autogenerado

- Después de un insert:
  - int nuevold;
  - sql << "INSERT INTO empleados(nombre, cargo) VALUES(:nombre, :cargo)",
  - soci::use("Marta"), soci::use("Diseñadora");

  - Para mysql:
    - sql << "SELECT **LAST_INSERT_ID**()", soci::into(nuevold);

Número de registros afectados

- SOCI permite obtener el número de filas modificadas por una consulta de acción (UPDATE, DELETE, etc.) usando **get_affected_rows**()

- sql << "UPDATE empleados SET cargo = 'Analista' WHERE cargo = 'Junior'";
- std::size_t filas = **sql.get_affected_rows**();
- std::cout << "Filas actualizadas: " << filas << std::endl;

PostGreSQL

libpq

Librería libpq

- **PGconn***
- Objeto principal de conexión a PostgreSQL.
  - Se obtiene con PQconnectdb(conninfo) y se libera con PQfinish(conn).
  - const char* conninfo = "host=127.0.0.1 port=5433 dbname=empresa3 user=antonio password=antonio";
  - PGconn* conn = PQconnectdb(conninfo);

Consultas

- Mejor parametrizadas:
  - Sin parametrizar, concatenando parámetros:
    - std::string query = "DELETE FROM tbempleados WHERE id = " + std::to_string(id);
    - PGresult* res = **PQexec**(conn_, query.c_str());
  - Parametrizada:
    - const char* query = "UPDATE tbempleados SET nombre = $1, cargo = $2 WHERE id = $3";
    - const char* paramValues[3] = { emp.nombre.c_str(), emp.cargo.c_str(), std::to_string(emp.id).c_str() };
    - PGresult* res = PQexecParams(conn_, query, 3, nullptr, paramValues, nullptr, nullptr, 0);

Otros objetos

Objeto / Función	Descripción
`PGresult*`	Resultado de una consulta
`PQresultStatus(res)`	Verifica si la consulta fue exitosa
`PQgetvalue(res, row, col)`	Obtiene el valor de una celda
`PQntuples(res)`	Número de filas devueltas
`PQclear(res)`	Libera la memoria del resultado
`PQerrorMessage(conn)`	Mensaje de error si algo falla

Transacciones 1 de 2

```
void ejecutarTransaccion(PGconn* conn) {
    PGresult* res;

    // Iniciar transacción
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
        PQclear(res);
        throw std::runtime_error("Error al iniciar la transacción");
    }
    PQclear(res);
```

Transacciones 2 de 2

```
try {
    // Operación 1
    res = PQexec(conn, "DELETE FROM tbempleados WHERE id = 10");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) throw
std::runtime_error("Error al eliminar");
    PQclear(res);

    // Operación 2
    const char* query = "INSERT INTO tbempleados(id, nombre, cargo)
VALUES($1, $2, $3)";
    const char* params[3] = { "10", "Laura", "Directivo" };
    res = PQexecParams(conn, query, 3, nullptr, params, nullptr, nullptr,
0);
    if (PQresultStatus(res) != PGRES_COMMAND_OK) throw
std::runtime_error("Error al insertar");
    PQclear(res);

// Confirmar transacción
    res = PQexec(conn, "COMMIT");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
            throw std::runtime_error("Error al confirmar");
    PQclear(res);

  } catch (const std::exception& ex) {
    std::cerr << "ERROR: " << ex.what() << std::endl;

    // Revertir transacción
    res = PQexec(conn, "ROLLBACK");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
       std::cerr << "Error al revertir la transacción: " <<
PQerrorMessage(conn) << std::endl;
    }
    PQclear(res);
  }
}
```

Filas afectadas

- Para insert, update y delete
- int filasAfectadas = std::stoi(**PQcmdTuples**(res));

Ultimo id generado

```
const char* query = "INSERT INTO tbempleados(nombre, cargo) VALUES($1, $2) RETURNING id";
const char* params[2] = { emp.nombre.c_str(), emp.cargo.c_str() };

PGresult* res = PQexecParams(conn_, query, 2, nullptr, params, nullptr, nullptr, 0);

if (PQresultStatus(res) != PGRES_TUPLES_OK) {
    PQclear(res);
    throw std::runtime_error("Error al insertar empleado");
}

int nuevoId = std::stoi(PQgetvalue(res, 0, 0));
PQclear(res);
```

BD NoSQL
MongoDB

Mongodb

- La librería **mongo-cxx-driver**, está disponible en **vcpkg**.
- https://www.mongodb.com/docs/languages/cpp/cpp-driver/current/

- **vcpkg install mongo-cxx-driver**
- **vcpkg integrate install**

- **OJO esta librería no ubica los .h en los mismos directorios que otras librerías.**

- Se instalan en:
  - **\vcpkg\installed\x64-windows\include\mongocxx\v-noabi**

En Visual Studio

- Agregar estas rutas de forma manual:
  - Propiedades del proyecto → **C/C++** → General → Additional Include Directories
    - $(VcpkgRoot)installed\x64-windows\include\mongocxx\v_noabi
    - $(VcpkgRoot)installed\x64-windows\include\bsoncxx\v_noabi
  - Propiedades del proyecto → **Linker** → General → Additional Library Directories
    - $(VcpkgRoot)installed\x64-windows\lib
  - Propiedades del proyecto → **Linker** → Input → Additional Dependencies
    - **mongocxx-v_noabi-rhi-md.lib**
    - **bsoncxx-v_noabi-rhi-md.lib**
- **Mejor copiar las DLLs de la carpeta bin a X64/Debug de la solución:**
  - Copia estas DLLs desde C:\vcpkg\installed\x64-windows\bin a x64\Debug:
    - **Si falta alguna DLL es posible que no se haya instalado correctamente el paquete de mongo en vcpkg → VER SOLUCION**
      - **mongocxx-v_noabi-rhi-md.dll**
      - **bsoncxx-v_noabi-rhi-md.dll**
      - **utf8proc.dll**
      - **libbson-1.0.dll**
      - **libmongoc-1.0.dll**
      - **snappy.dll (si está presente)**

Solución problemas con las DLLs

- Instala el cliente **C nativo de MongoDB**
  - **vcpkg install mongo-c-driver --triplet x64-windows**
    - **libbson-1.0.dll**
    - **libmongoc-1.0.dll**
  - **vcpkg install snappy --triplet x64-windows**
    - **snappy.dll (si está presente)**

- **Reinstala el cliente C++** asegurándote de que lo enlaza:
  - **vcpkg install mongo-cxx-driver --triplet x64-windows**
    - Esto enlaza mongocxx y bsoncxx con el cliente C correctamente.

Solución problemas con las DLLs

- Si no aparecen las DLLs en el directorio bin, eliminar y volver a instalar:
  - **vcpkg remove mongo-cxx-driver --recurse**

- Instalar el cliente nativo de C
  - **vcpkg install mongo-c-driver --triplet x64-windows**

- Instalar el cliente de C++
  - **vcpkg install mongo-cxx-driver --triplet x64-windows**

Ejemplo: conexión con mongo

```
#include <mongocxx/client.hpp>
#include <mongocxx/exception/exception.hpp>
#include <mongocxx/instance.hpp>
#include <mongocxx/uri.hpp>

#include <iostream>

void testConexion() {

   try {
      mongocxx::instance inst{};  // static para asegurar duración
      mongocxx::client client{ mongocxx::uri{"mongodb://host.docker.internal:27017"} };

      std::cout << "Conexion establecida con MongoDB." << std::endl;
   }
   catch (const std::exception& e) {
      std::cerr << "Error: " << e.what() << std::endl;
   }
}
```

Ejemplo: listar documentos

```
void testListar() {
   try {
      mongocxx::instance inst{};  // Inicializa el entorno MongoDB
      mongocxx::client client{ mongocxx::uri{"mongodb://host.docker.internal:27017"} };

      auto db = client["bdwebs"];         // Accede a la base de datos
      auto collection = db["webs"];       // Accede a la colección

      // Itera sobre los documentos
      for (auto&& doc : collection.find({})) {
         std::cout << bsoncxx::to_json(doc) << std::endl;
      }

   }
   catch (const std::exception& e) {
      std::cerr << "Error: " << e.what() << std::endl;
      return;
   }
}
```

Estrategias de persistencia y acceso rápido a datos

Caching con Redis o Memcached en aplicaciones C++

- **Caching con Redis o Memcached en C++** puede mejorar drásticamente el rendimiento las aplicaciones, especialmente en sistemas que hacen muchas consultas repetitivas o requieren baja latencia.

- **Caching** en C++
  - Reduce la carga en la base de datos
  - Minimiza el tiempo de respuesta
  - Mejora la escalabilidad
  - Evita cálculos repetidos

Redis vs Memcached en C++

Característica	Redis 🧠	Memcached ⚡
Tipo de datos	Estructuras complejas (listas, sets, hashes)	Clave-valor simple
Persistencia	Opcional (snapshot o append-only)	No persistente
Escalabilidad	Alta, con clustering	Alta, con distribución simple
Seguridad	Autenticación, ACLs	Básica
Casos de uso	Sesiones, contadores, colas	Caching de objetos simples
Cliente C++	hiredis + wrapper C++	libmemcached

Redis

- Es una **base de datos clave – valor**.
- Redis tiene dos formas de guardar datos en disco:
  - **RDB (Redis Database Snapshot)**: guarda snapshots periódicos.
  - **AOF (Append-Only File)**: registra cada operación que modifica datos.

  - En el arranque si usamos: **redis-server *--appendonly yes***, Redis:
    - Comienza a registrar **cada comando que cambia el estado** (como SET, DEL, etc.)
    - Guarda estos comandos en un archivo llamado **appendonly.aof**

    - Puede **reconstruir el estado completo** del servidor ejecutando ese archivo en el arranque
    - Esto es útil si quieres que Redis **recupere automáticamente los datos** después de un reinicio, incluso si lo usas como caché con tolerancia a fallos.

Redis: patrones típicos

- **Caché al margen (Cache-aside)**
  - El patrón más común:
  - El microservicio consulta Redis antes de ir a la base de datos.
  - Si el dato está en Redis → se devuelve directamente.
  - Si no está → se consulta la base de datos, se guarda en Redis y se devuelve.

```
auto val = redis.get("user:123");
if (!val) {
   val = fetchFromDatabase("user:123");
   redis.set("user:123", *val);
}
```

Redis: patrones típicos

- **Cache distribuida**
  - Redis puede actuar como **caché compartida entre múltiples microservicios**, incluso en distintos contenedores o nodos. Esto permite:
  - Compartir resultados de consultas costosas
  - Evitar duplicación de lógica de negocio
  - Reducir carga en servicios centrales
  - Redis Cluster o Redis Sentinel se usan para alta disponibilidad y escalabilidad horizontal

Redis: patrones típicos

- **Caché de sesión o tokens:**
  - Redis se usa para almacenar:
  - Tokens JWT o sesiones de usuario
  - Estados temporales de autenticación
  - Flags de autorización
  - Esto permite que servicios de autenticación y autorización trabajen de forma desacoplada.

Redis: patrones típicos

- Caché de eventos o colas:
  - Redis también puede actuar como:
  - **Buffer de eventos** entre microservicios
  - **Cola temporal** usando listas (LPUSH, RPOP)
  - **Pub/Sub** para notificaciones entre servicios

Librería hiredis

- Se instala con vcpkg:
  - vcpkg install hiredis
  - vcpkg integrate install

  - Para utilizarla en los proyectos de C++ (aunque esta desarrollada en C):
  - **#include <hiredis/hiredis.h>**

Librería hiredis: objetos

- redisContext Representa la conexión con el servidor Redis
- redisReply Representa la respuesta de Redis a un comando
- redisReader Parse de respuestas

Librería hiredis: objetos

- **redisContext**
- Es el núcleo de la conexión.
- Contiene información como:
  - IP, puerto
  - Estado de error (err, errstr)
  - Socket y buffer de lectura/escritura

- redisContext* context = redisConnect("127.0.0.1", 6379);

Librería hiredis: objetos

- **redisReply**
- Devuelto por redisCommand()

- Tiene campos según el tipo de respuesta:
  - type: tipo de respuesta (REDIS_REPLY_STRING, ARRAY, INTEGER, etc.)
  - str: si es una cadena
  - integer: si es un número
  - elements: número de elementos si es un array
  - element[]: array de redisReply* si es una respuesta compuesta

- redisReply* reply = (redisReply*)redisCommand(context, "GET clave");
- std::cout << reply->str << std::endl;

Librería hiredis: objetos

- **redisReader (menos común en C++)**
- Se usa para **parsear respuestas** en modo asincrónico o embebido.
- Útil si estás integrando **hiredis** en una arquitectura personalizada.

Jerarquía funcional (no orientada a objetos)

- redisContext
  - redisCommand(...) → redisReply
    - type
    - str / integer / elements
    - element[] (si es array)

- Tipos de respuesta:

```
#define REDIS_REPLY_STRING    1
#define REDIS_REPLY_ARRAY     2
#define REDIS_REPLY_INTEGER   3
#define REDIS_REPLY_NIL       4
#define REDIS_REPLY_STATUS    5
#define REDIS_REPLY_ERROR     6
```

Otros tipos de redis

- Hash
  - HSET empleado:123 campo valor
  - Hset empleado:123 id 123 nombre pepe cargo telefonista
- String
  - Serializar como JSON o CSV
- List
  - Lista de Ids o nombres
  - Lpush pedidos 12345
  - Lpush pedidos 44533
  - Lpush pedidos 55123
  - Lrange pedidos 0 -1
- Set
  - Para evitar duplicados

Comandos para listas

Comando	Descripción
`LPUSH`	Inserta al inicio de la lista
`RPUSH`	Inserta al final de la lista
`LPOP`	Extrae el primer elemento
`RPOP`	Extrae el último elemento
`LRANGE`	Obtiene un rango de elementos
`LLEN`	Devuelve la longitud de la lista
`LTRIM`	Recorta la lista a un rango específico
`BLPOP`	Extrae el primer elemento, bloqueando si vacío

Comandos para conjuntos

Comando	Descripción
`SADD key val`	Añade uno o más elementos al conjunto
`SREM key val`	Elimina uno o más elementos del conjunto
`SMEMBERS key`	Devuelve todos los elementos del conjunto
`SCARD key`	Devuelve el número de elementos del conjunto
`SISMEMBER key val`	Verifica si un elemento pertenece al conjunto
`SPOP key`	Elimina y devuelve un elemento aleatorio del conjunto
`SRANDMEMBER key [count]`	Devuelve uno o varios elementos aleatorios sin eliminarlos
`SUNION key1 key2`	Devuelve la unión de dos conjuntos
`SINTER key1 key2`	Devuelve la intersección de dos conjuntos
`SDIFF key1 key2`	Devuelve la diferencia entre dos conjuntos
`SUNIONSTORE dest key1 key2`	Guarda la unión en un nuevo conjunto
`SINTERSTORE dest key1 key2`	Guarda la intersección en un nuevo conjunto
`SDIFFSTORE dest key1 key2`	Guarda la diferencia en un nuevo conjunto

Ejemplos conjuntos

- SADD empleados "juan" "ana" "pedro"
- SADD gerentes "ana" "lucas"

- SINTER empleados gerentes # → "ana"
- SUNION empleados gerentes # → "juan", "ana", "pedro", "lucas"
- SDIFF empleados gerentes # → "juan", "pedro"

MemCached

- Definición: Sistema de caché distribuido en memoria, diseñado para almacenar pares clave-valor simples.

- Objetivo: Acelerar aplicaciones web reduciendo la carga en bases de datos.

- Características:
  - Muy rápido y ligero.
  - Multihilo: aprovecha múltiples núcleos.
  - Ideal para almacenar objetos temporales como resultados de consultas SQL, sesiones o fragmentos HTML.
  - No admite persistencia ni estructuras de datos complejas.

Comparativa Redis vs MemCached

Característica	Memcached	Redis
Modelo de datos	Clave-valor simple	Clave-valor con estructuras complejas
Persistencia	No	Sí (RDB, AOF)
Multihilo	Sí	No (monohilo)
Tipos de datos	Solo cadenas	Listas, conjuntos, hashes, etc.
Transacciones	No	Sí
Replicación	No	Sí
Expiración de claves	Sí	Sí
Rendimiento	Muy rápido para datos simples	Rápido y flexible
Casos de uso ideales	Caché de objetos simples	Sesiones, colas, contadores, pub/sub

MemCached

- Sólo almacena cadenas, una opción es guardar los objetos en una cadena de JSON para luego poder recuperar el json y poder restaurar el objeto.

Implementación de acceso en tiempo real a grandes volúmenes de datos

- **Flujo típico de datos**
  - **Crow** recibe una petición HTTP (por ejemplo, /usuario?id=123).
  - El handler consulta **Redis** con hiredis para ver si el dato está en caché.
  - Si no está, consulta la base de datos con **SOCI**.
  - El resultado se guarda en Redis para futuras consultas.
  - Se devuelve la respuesta al cliente vía Crow.
  - Se registra una métrica con **prometheus-cpp** (tiempo de respuesta, hits en caché, etc.).
  - **Grafana** visualiza esas métricas en tiempo real.
  - Si hay otros servicios que necesitan datos, se comunican vía **gRPC**.

Componentes

Componente	Librería	Rol en el sistema
Base de datos SQL	`soci`	Acceso a PostgreSQL, MySQL, SQLite, etc. con interfaz moderna en C++.
API REST	`crow`	Servidor HTTP ligero para exponer endpoints en tiempo real.
Caché en memoria	`hiredis`	Cliente Redis para acelerar acceso a datos frecuentes o temporales.
Monitorización	`prometheus-cpp`	Exportación de métricas para observabilidad.
Visualización	`Grafana`	Dashboards en tiempo real basados en métricas de Prometheus.
Comunicación RPC	`gRPC`	Comunicación eficiente entre servicios distribuidos en tiempo real.

Ejemplo

```
// Crow endpoint
crow::SimpleApp app;

CROW_ROUTE(app, "/usuario/<int>")([](int id){
   // 1. Consultar Redis con hiredis
   // 2. Si no está, usar SOCI para consultar la base de datos
   // 3. Guardar en Redis
   // 4. Registrar métrica con prometheus-cpp
   // 5. Devolver JSON
   return crow::response(200, "{\"id\":123,\"nombre\":\"Laura\"}");
});

app.port(8080).multithreaded().run();
```