

Hardening the API

The borrowing workflow from the previous chapter works when requests arrive one at a time, but it contains a race condition. Two requests can both observe the same book as available before either one marks it as unavailable. Let's see how that can happen.

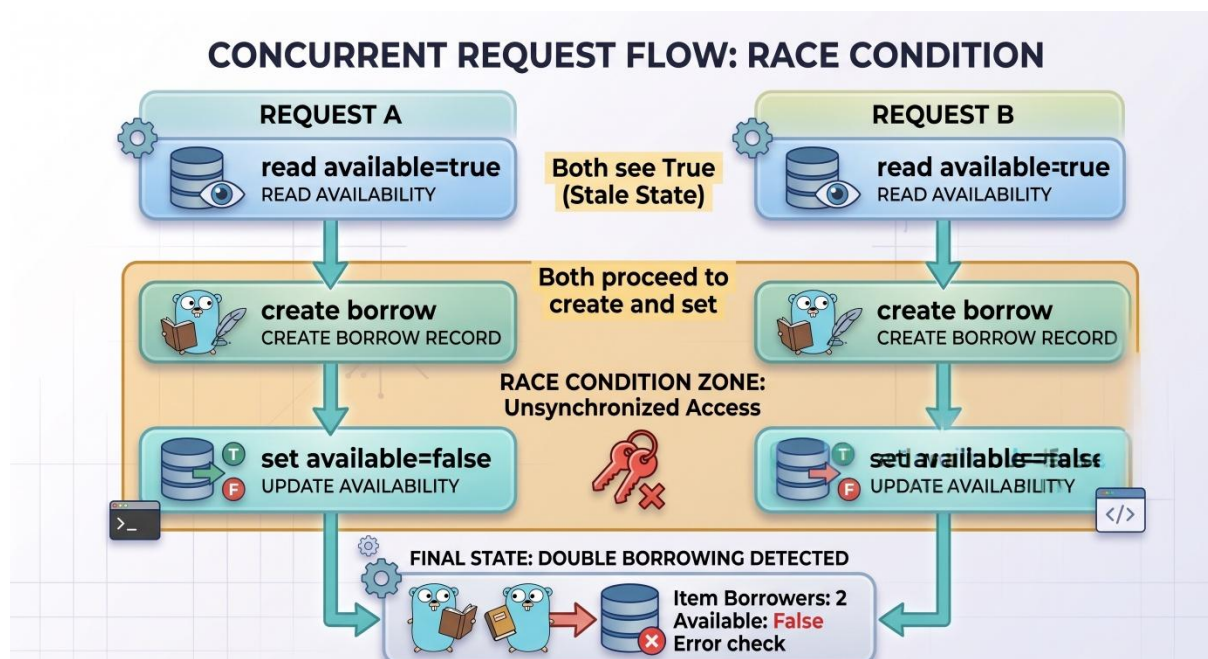
Request A (User A borrows Book #5):

- *GetBook(5)* : This checks whether the book exists and returns a book with the *available* flag set to true.
- Check passes: the book exists and is available.
- Next, *GetUser()* is called and confirms the user exists.
- Now we have a pause where request B runs.
- *BorrowBook(book=5,user=A)* runs. This creates a new borrow record.
- *UpdateBookAvailability(book=5, available=false)* marks the book as unavailable.
- Response: Success.

However, we see the problem when request B runs:

- *GetBook(5)* runs. The book still exists and available is still true the moment this request is made, so the borrowing process can still continue.
- *GetUser(B)*. This user exists.
- Because the book is available we can call *BorrowBook(book=5,user=B)* which creates another borrow record.
- *UpdateBookAvailability(book=5,available=false)* marks the book as unavailable
- This request also succeeds.

Or to visualize this:



This is a classic check-then-act race condition. Each request reads the current state, makes a decision based on that state, and then changes it. Because those steps are separate database operations, both requests can observe *available = true* before either one changes it.

We need the availability check and the changes that follow it to behave as one atomic operation. We will do that with a database transaction and a row lock, ensuring that only one borrowing request can inspect and modify a particular book at a time.

The Fix

Fixing the null values in the response

Before tackling the concurrency problem, let's fix another issue we deliberately left visible in the previous chapter: nullable timestamps. Our previous response conversion ignored the validity flag and used the *Time* field anyway, causing Go's zero *time.Time* value (0001-01-01T00:00:00Z) to appear in the JSON response.

How to solve this? Well, that is actually quite easy. Instead of representing *returned_at* as a *time.Time* value, we will use a **time.Time*. A nil pointer can represent SQL *NULL*, allowing the JSON response to contain *null* rather than Go's zero time value. Let's have a look at an example, *borrow_record_response.go*:

```
package models

import (
    db "bookbackend/internal/database"
    "time"
)

type BorrowRecordResponse struct {
    ID          int32    `json:"id"`
    BookID      int32    `json:"book_id"`
    UserID      int32    `json:"user_id"`
    BorrowedAt  time.Time `json:"borrowed_at"`
    DueDate     time.Time `json:"due_date"`
    ReturnedAt  *time.Time `json:"returned_at"`
}

func ToBorrowRecordResponse(borrowedBook db.BorrowedBook) BorrowRecordResponse {
    response := BorrowRecordResponse{
        ID:          borrowedBook.ID,
        BookID:      borrowedBook.BookID,
        UserID:      borrowedBook.UserID,
        BorrowedAt:  borrowedBook.BorrowedAt.Time,
        DueDate:     borrowedBook.DueDate.Time,
    }

    if borrowedBook.ReturnedAt.Valid {
        response.ReturnedAt = &borrowedBook.ReturnedAt.Time
    } else {
        response.ReturnedAt = nil
    }

    return response
}
```

ToBorrowRecordResponse() checks the *pgx* validity flag. When the database value is present, the response points to that timestamp; otherwise *ReturnedAt* remains *nil* and is encoded as JSON *null*.

We do the same for *borrowed_book_row_response.go*:

```
package models

import (
    db "bookbackend/internal/database"
    "time"
)

type BorrowedBooksRowResponse struct {
    ID          int32    `json:"id"`
    BookID      int32    `json:"book_id"`
    UserID      int32    `json:"user_id"`
    BorrowedAt  time.Time `json:"borrowed_at"`
    DueDate     time.Time `json:"due_date"`
    ReturnedAt  *time.Time `json:"returned_at"`
    Title       string    `json:"title"`
    Author      string    `json:"author"`
}

func ToBorrowRecordRowResponse(row db.GetUserBorrowedBooksRow) BorrowedBooksRowResponse {
    response := BorrowedBooksRowResponse{
        ID:          row.ID,
        BookID:      row.BookID,
        UserID:      row.UserID,
        BorrowedAt:  row.BorrowedAt.Time,
        DueDate:     row.DueDate.Time,
        Title:       row.Title,
        Author:      row.Author,
    }

    if row.ReturnedAt.Valid {
        response.ReturnedAt = &row.ReturnedAt.Time
    } else {
        response.ReturnedAt = nil
    }

    return response
}
```

And for *overdue_books_row_response.go*:

```
package models

import (
    db "bookbackend/internal/database"
    "time"
)

type OverdueBooksRowResponse struct {
    ID          int32    `json:"id"`
    BookID      int32    `json:"book_id"`
    UserID      int32    `json:"user_id"`
    BorrowedAt  time.Time `json:"borrowed_at"`
    DueDate     time.Time `json:"due_date"`
    ReturnedAt  *time.Time `json:"returned_at"`
    Title       string    `json:"title"`
    Author      string    `json:"author"`
    Name        string    `json:"name"`
    Email       string    `json:"email"`
}

func ToOverdueBooksRowResponse(row db.GetOverdueBooksRow) OverdueBooksRowResponse {
    response := OverdueBooksRowResponse{
        ID:          row.ID,
        BookID:      row.BookID,
        UserID:      row.UserID,
        BorrowedAt:  row.BorrowedAt.Time,
        DueDate:     row.DueDate.Time,
        Title:       row.Title,
        Author:      row.Author,
        Name:        row.Name,
        Email:       row.Email,
    }

    if row.ReturnedAt.Valid {
        response.ReturnedAt = &row.ReturnedAt.Time
    } else {
        response.ReturnedAt = nil
    }

    return response
}
```

Making the Borrowing Process Concurrency Safe

Implementing the fix starts by adding a new query to our *books.sql* file:

```
-- name: GetBookForUpdate :one
SELECT * FROM books
WHERE id = $1
FOR UPDATE;
```

This is a normal select statement with an addition. The *FOR UPDATE* tells PostgreSQL to lock the selected row for the duration of the transaction. If another transaction executes the same *SELECT ... FOR UPDATE* against that row, it waits until the first transaction commits or rolls back.

Note: *SELECT ... FOR UPDATE* protects the borrowing path from the race condition. Shortly, we will also add a database constraint so that PostgreSQL independently enforces the rule that a book can have only one active loan.

Before using the new query, regenerate the sqlc code:

```
sqlc generate
```

Because the generated *Querier* interface does not expose *WithTx()*, we need a small interface that combines the generated query methods with the ability to create transaction-scoped queries. We should not modify generated code, so create *internal/api/querierwithtx.go*:

```
// internal/api/querierwithtx.go
package api

import (
    db "bookbackend/internal/database"

    "github.com/jackc/pgx/v5"
)

type QuerierWithTx interface {
    db.Querier
    WithTx(tx pgx.Tx) *db.Queries
}
```

QuerierWithTx embeds the generated *db.Querier* interface and adds *WithTx()*. This lets the handler use the same query methods normally while also creating a **db.Queries* value bound to a transaction.

Now we need to start work on *bookhandlers.go*. As we are going to need transactions, we need to be able to access the database connection pool. So, change the *BookHandler* struct to this:

```
type BookHandler struct {
    queries QuerierWithTx
    pool    *pgxpool.Pool
}
```

Also change the constructor:

```
func NewBookHandler(queries QuerierWithTx, pool *pgxpool.Pool) *BookHandler {  
    return &BookHandler{queries: queries, pool: pool}  
}
```

Note: What are transactions and why do we need them? A transaction groups several database operations into one atomic unit: either all changes are committed or none are. PostgreSQL also acquires locks when necessary while executing statements inside a transaction. In our borrowing workflow, we explicitly use *SELECT ... FOR UPDATE* to lock the selected book row until the transaction either commits or rolls back.

Now we can start to work on our *BorrowBook()* method. The method signature remains the same, and the decoding and validation as well. After that we set up the transaction:

```
tx, err := h.pool.BeginTx(r.Context(), pgx.TxOptions{})  
if err != nil {  
    log.Printf("Unable to start transaction: %v", err)  
    writeError(w, http.StatusInternalServerError, "Internal server error")  
    return  
}  
defer tx.Rollback(r.Context())  
  
qtx := h.queries.WithTx(tx)
```

We start a transaction using the connection pool. A deferred rollback ensures that if the handler returns before the transaction is committed, its changes are discarded. Calling *Rollback()* after a successful *Commit()* is safe in this pattern because the transaction has already been closed. Finally, *WithTx(tx)* creates a generated query object whose operations all execute within that transaction.

Now we want to know if the book is available:

```
book, err := qtx.GetBookForUpdate(r.Context(), req.BookID)  
if err != nil {  
    if errors.Is(err, pgx.ErrNoRows) {  
        writeError(w, http.StatusNotFound, "Book not found")  
        return  
    }  
    log.Printf("Unable to get book: %v", err)  
    writeError(w, http.StatusInternalServerError, "Internal server error")  
    return  
}  
  
if !book.Available {  
    // Book is not available for borrowing  
    writeError(w, http.StatusConflict, "Book is not available for borrowing")  
    return  
}
```

- We use the *GetBookForUpdate()* method. This method executes a *SELECT* statement with a *FOR UPDATE* which not only retrieves the book row, but also places a row-level lock on the selected row in the database. Any other transactions trying to call *GetBookForUpdate()* on the same row will block and wait until the current transaction either commits or rolls back.
- Next we handle the error: we distinguish between a missing book, which results in a *404 Not Found*, and unexpected database errors, which result in a *500 Internal Server Error*.

- We also check the book's availability and if it is not available, return an error.

The next thing to do is fetch the user:

```
_, err = qtx.GetUser(r.Context(), req.UserID)
if err != nil {
    if errors.Is(err, pgx.ErrNoRows) {
        writeError(w, http.StatusNotFound, "User not found")
        return
    }
    log.Printf("Unable to get user: %v", err)
    writeError(w, http.StatusInternalServerError, "Internal server error")
    return
}
```

We now have a book which is available, and an existing user, so we can create a new borrow record:

```
dueDate := time.Now().Add(time.Duration(req.Days) * 24 * time.Hour)

borrowRecord, err := qtx.BorrowBook(r.Context(), db.BorrowBookParams{
    BookID: req.BookID,
    UserID: req.UserID,
    DueDate: pgtype.Timestampt{Time: dueDate, Valid: true},
})

if err != nil {
    log.Printf("Unable to borrow book: %v", err)
    writeError(w, http.StatusInternalServerError, "Internal server error")
    return
}
```

Next update the availability:

```
err = qtx.UpdateBookAvailability(r.Context(), db.UpdateBookAvailabilityParams{
    ID: req.BookID,
    Available: false,
})

if err != nil {
    log.Printf("Unable to update book availability: %v", err)
    writeError(w, http.StatusInternalServerError, "Internal server error")
    return
}
```

Remember, we are still working within the transaction so we can safely update the availability.

Everything is set, so we can finally commit the transaction:

```
if err = tx.Commit(r.Context()); err != nil {  
    log.Printf("Unable to commit transaction: %v", err)  
    writeError(w, http.StatusInternalServerError, "Internal server error")  
    return  
}
```

And finally we return the result of our labours, a borrow record:

```
writeJSON(w, http.StatusOK, models.ToBorrowRecordResponse(borrowRecord))
```

The *BorrowBook()* method now looks like this:

```
func (h *BookHandler) BorrowBook(w http.ResponseWriter, r *http.Request) {
    var req BorrowBookRequest
    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        log.Printf("Unable to decode request body: %v", err)
        writeError(w, http.StatusBadRequest, "Invalid request body")
        return
    }

    if err := req.Validate(); err != nil {
        writeError(w, http.StatusBadRequest, err.Error())
        return
    }

    if req.Days == 0 {
        req.Days = 14
    }
    tx, err := h.pool.BeginTx(r.Context(), pgx.TxOptions{})
    if err != nil {
        log.Printf("Unable to start transaction: %v", err)
        writeError(w, http.StatusInternalServerError, "Internal server error")
        return
    }
    defer tx.Rollback(r.Context())

    qtx := h.queries.WithTx(tx)

    //Check if book exists
    book, err := qtx.GetBookForUpdate(r.Context(), req.BookID)
    if err != nil {
        if errors.Is(err, pgx.ErrNoRows) {
            writeError(w, http.StatusNotFound, "Book not found")
            return
        }
        log.Printf("Unable to get book: %v", err)
        writeError(w, http.StatusInternalServerError, "Internal server error")
        return
    }

    if !book.Available {
        // Book is not available for borrowing
        writeError(w, http.StatusConflict, "Book is not available for borrowing")
        return
    }

    _, err = qtx.GetUser(r.Context(), req.UserID)
    if err != nil {
        if errors.Is(err, pgx.ErrNoRows) {
            writeError(w, http.StatusNotFound, "User not found")
            return
        }
        log.Printf("Unable to get user: %v", err)
        writeError(w, http.StatusInternalServerError, "Internal server error")
        return
    }

    dueDate := time.Now().Add(time.Duration(req.Days) * 24 * time.Hour)

    borrowRecord, err := qtx.BorrowBook(r.Context(), db.BorrowBookParams{
        BookID: req.BookID,
        UserID: req.UserID,
        DueDate: pgtype.Timestampz{Time: dueDate, Valid: true},
    })

    if err != nil {
        var pgErr *pgconn.PgError
    }
```

```

        if errors.As(err, &pgErr) && pgErr.Code == "23505" && pgErr.ConstraintName ==
"idx_one_active_borrow_per_book" {
            writeError(w, http.StatusConflict, "Book is not available for borrowing")
            return
        }
        log.Printf("Unable to borrow book: %v", err)
        writeError(w, http.StatusInternalServerError, "Internal server error")
        return
    }
    err = qtx.UpdateBookAvailability(r.Context(), db.UpdateBookAvailabilityParams{
        ID:      req.BookID,
        Available: false,
    })
    if err != nil {
        log.Printf("Unable to update book availability: %v", err)
        writeError(w, http.StatusInternalServerError, "Internal server error")
        return
    }

    if err = tx.Commit(r.Context()); err != nil {
        log.Printf("Unable to commit transaction: %v", err)
        writeError(w, http.StatusInternalServerError, "Internal server error")
        return
    }
    writeJSON(w, http.StatusOK, models.ToBorrowRecordResponse(borrowRecord))
}

```

Note: An important addition to *BorrowBook()* is conflict handling. A book cannot have more than one active loan, so a violation of the unique constraint is returned to the client as *409 Conflict*.

The *ReturnBook()* method gets similar changes:

```

func (h *BookHandler) ReturnBook(w http.ResponseWriter, r *http.Request) {
    var req ReturnBookRequest

    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        log.Printf("Unable to decode request body: %v", err)
        writeError(w, http.StatusBadRequest, "Invalid request body")
        return
    }

    if err := req.Validate(); err != nil {
        writeError(w, http.StatusBadRequest, err.Error())
        return
    }

    tx, err := h.pool.BeginTx(r.Context(), pgx.TxOptions{})
    if err != nil {
        log.Printf("Unable to start transaction: %v", err)
        writeError(w, http.StatusInternalServerError, "Internal server error")
        return
    }
    defer tx.Rollback(r.Context())

    qtx := h.queries.WithTx(tx)
    // Return the book
    returnRecord, err := qtx.ReturnBook(r.Context(), db.ReturnBookParams{
        BookID: req.BookID,
        UserID: req.UserID,
    })
    if err != nil {
        if errors.Is(err, pgx.ErrNoRows) {
            writeError(w, http.StatusNotFound, "No active borrow record found")
        }
    }
}

```

```

        return
    }
    log.Printf("Unable to retrieve borrow record: %v", err)
    writeError(w, http.StatusInternalServerError, "Internal server error")
    return
}

// Mark book as available
err = qtx.UpdateBookAvailability(r.Context(), db.UpdateBookAvailabilityParams{
    ID:      req.BookID,
    Available: true,
})

if err != nil {
    log.Printf("Unable to update book availability: %v", err)
    writeError(w, http.StatusInternalServerError, "Internal server error")
    return
}
if err = tx.Commit(r.Context()); err != nil {
    log.Printf("Unable to commit transaction: %v", err)
    writeError(w, http.StatusInternalServerError, "Internal server error")
    return
}
writeJSON(w, http.StatusOK, models.ToReturnRecordResponse(returnRecord, "Book returned successfully"))
}

```

The row lock protects the borrowing workflow, but PostgreSQL should also enforce the underlying invariant. Start by creating a migration:

```
goose -dir db/migrations create create_unique_borrowed_index sql
```

And make sure the newly created file reads like this:

```

-- +goose Up
CREATE UNIQUE INDEX idx_one_active_borrow_per_book
  ON borrowed_books(book_id)
  WHERE returned_at IS NULL;

-- +goose Down
DROP INDEX IF EXISTS idx_one_active_borrow_per_book;

```

What does this mean?

1. Only one row per *book_id* can exist where *returned_at* is NULL
2. Once the book is returned (*returned_at* is set) the row no longer matches the index predicate, so another borrow can happen later.

Now you can apply these to the database:

```
goose -dir db/migrations postgres "postgres://<username>:<password>@<hostname>:<portnumber>/<databasename>?sslmode=disable" up
```

Updating main

Because *NewBookHandler()* now also requires the connection pool, update its construction in *main()*:

```
bookhandler := api.NewBookHandler(queries, pool)
```

The only change is that we now pass the connection pool as an extra parameter to the *NewBookHandler()* function.

Preserving history

Our current deletion model has another problem: deleting a book or user also removes its borrowing history. We will solve this with soft deletion. Instead of physically deleting a row, we add a nullable *deleted_at* timestamp and mark the row as deleted.

Soft deletion introduces some additional design questions. For example, because the original users table requires email addresses to be globally unique, a deleted user still prevents another active user from registering with the same email address. We will address that shortly with a partial unique index. Soft-deleted records may also need to be permanently removed after a retention period, depending on application and privacy requirements.

Start by creating a migration:

```
goose -dir db/migrations create add_soft_delete sql
```

Replace the generated migration contents with:

```
-- +goose Up

ALTER TABLE books ADD COLUMN deleted_at TIMESTAMPTZ;
ALTER TABLE users ADD COLUMN deleted_at TIMESTAMPTZ;

-- Add index for performance when filtering out deleted records
CREATE INDEX idx_books_deleted_at ON books(deleted_at);
CREATE INDEX idx_users_deleted_at ON users(deleted_at);

-- +goose Down

DROP INDEX IF EXISTS idx_books_deleted_at;
DROP INDEX IF EXISTS idx_users_deleted_at;

ALTER TABLE books DROP COLUMN deleted_at;
ALTER TABLE users DROP COLUMN deleted_at;
```

Some notes:

- We start by adding two new nullable timestamps to the *books* and *users* tables respectively.
- Add indexes that may help queries filtering on *deleted_at*, depending on the size and the distribution of the data.
- In the *Down* section we reverse those changes by removing the indexes and the new columns.

To apply the migration:

```
goose -dir db/migrations postgres "postgres://<username>:<password>@<hostname>:<portnumber>/<databasename>?sslmode=disable" up
```

We now update the existing queries so that normal API operations ignore soft-deleted rows. Start with *ListBooks*:

```
-- name: ListBooks :many
SELECT * FROM books
WHERE deleted_at IS NULL
ORDER BY id;
```

Note the check for the *deleted_at* column. This will be a common pattern, for example for *GetBook*:

```
-- name: GetBook :one
SELECT * FROM books
WHERE id = $1 AND deleted_at IS NULL;
```

We also need to check for the *deleted_at* timestamp when deleting or updating a book:

```
-- name: UpdateBook :one
UPDATE books
SET title = $2, author = $3, description = $4, year_of_publication = $5, updated_at =
CURRENT_TIMESTAMP
WHERE id = $1 AND deleted_at IS NULL
RETURNING *;

-- name: DeleteBook :one
UPDATE books
SET deleted_at = CURRENT_TIMESTAMP
WHERE id = $1 AND deleted_at IS NULL
RETURNING id;
```

And finally we need to update the *GetBookForUpdate* query:

```
-- name: GetBookForUpdate :one
SELECT * FROM books
WHERE id = $1 AND deleted_at IS NULL
FOR UPDATE;
```

The same sort of thing happens in the *users.sql* file. First the *ListUsers* query:

```
-- name: ListUsers :many
SELECT * FROM users
WHERE deleted_at IS NULL
ORDER BY id;
```

Next the *GetUser* query:

```
-- name: GetUser :one
SELECT * FROM users
WHERE id = $1 AND
deleted_at IS NULL;
```

And finally the *DeleteUser* query which like the book's delete query has become an update:

```
-- name: DeleteUser :one
UPDATE users
SET deleted_at = CURRENT_TIMESTAMP
WHERE id = $1 AND deleted_at IS NULL
RETURNING id;
```

And of course *UpdateUser* needs this as well:

```
-- name: UpdateUser :one
UPDATE users
SET name = $2, email = $3, updated_at = CURRENT_TIMESTAMP
WHERE id = $1 AND deleted_at IS NULL
RETURNING *;
```

Finally we update the *GetUserByEmail* query:

```
-- name: GetUserByEmail :one
SELECT * FROM users
WHERE email = $1 AND
deleted_at IS NULL;
```

To use this new functionality run:

```
sqlc generate
```

Adding a borrowing history endpoint

We now want to have an endpoint where we can see a user's complete borrowing history, regardless of whether the books have been returned. We will start by adding a new query to *borrowed_books.sql* file:

```
-- name: GetUserBorrowHistory :many
SELECT bb.*, b.title, b.author
FROM borrowed_books bb
      JOIN books b ON bb.book_id = b.id
WHERE bb.user_id = $1
ORDER BY bb.borrowed_at DESC;
```

Again run:

```
sqlc generate
```

Before we can continue, add a *history_row_response.go* in the *internal/api/models* directory:

```
package models

import (
    db "bookbackend/internal/database"
    "time"
)

type HistoryRowResponse struct {
    ID          int32    `json:"id"`
    BookID      int32    `json:"book_id"`
    UserID      int32    `json:"user_id"`
    BorrowedAt  time.Time `json:"borrowed_at"`
    DueDate     time.Time `json:"due_date"`
    ReturnedAt  *time.Time `json:"returned_at"`
    Title       string    `json:"title"`
    Author      string    `json:"author"`
}

func ToHistoryRowResponse(row db.GetUserBorrowHistoryRow) HistoryRowResponse {

    response := HistoryRowResponse{
        ID:          row.ID,
        BookID:      row.BookID,
        UserID:      row.UserID,
        BorrowedAt:  row.BorrowedAt.Time,
        DueDate:     row.DueDate.Time,
        Title:       row.Title,
        Author:      row.Author,
    }

    if row.ReturnedAt.Valid {
        response.ReturnedAt = &row.ReturnedAt.Time
    } else {
        response.ReturnedAt = nil
    }
    return response
}
```

Now in the *userhandlers.go* file, add this method:

```
func (handler *UserHandler) BorrowHistory(w http.ResponseWriter, r *http.Request) {
    id, err := parseID(r)
    if err != nil {
        log.Printf("Error parsing ID: %v", err)
        writeError(w, http.StatusBadRequest, "Invalid user ID")
        return
    }

    books, err := handler.queries.GetUserBorrowHistory(r.Context(), id)
    if err != nil {
        log.Printf("Error fetching borrow history: %v", err)
        writeError(w, http.StatusInternalServerError, "Failed to fetch borrow history")
        return
    }
    history := make([]models.HistoryRowResponse, len(books))
    for i, book := range books {
        history[i] = models.ToHistoryRowResponse(book)
    }
    writeJSON(w, http.StatusOK, history)
}
```

The handler parses the user ID, retrieves the borrowing history, converts each database row into an API response model, and returns the resulting list as JSON.

To expose this functionality through the API, update *Routes()* as well:

```
func (handler *UserHandler) Routes() chi.Router {
    r := chi.NewRouter()
    r.Get("/", handler.FetchAllUsers)
    r.Get("/{id}", handler.FetchUserById)
    r.Get("/{id}/history", handler.BorrowHistory)
    r.Post("/", handler.CreateUser)
    r.Put("/{id}", handler.UpdateUser)
    r.Delete("/{id}", handler.DeleteUser)
    return r
}
```

The duplicate email problem

We have one other problem. If a user is soft deleted, no other user can register with the same email as the deleted user. This is also something you might not want. As you can see in the SQL definition of the user table, the *email* should be unique:

```
email VARCHAR(255) UNIQUE NOT NULL
```

In order to make deleted users' email addresses reusable we have to add a migration:

```
goose -dir db/migrations create add_email_index sql
```

And make sure the newly created SQL file looks like this:

```
-- +goose Up
ALTER TABLE users DROP CONSTRAINT users_email_key;

CREATE UNIQUE INDEX idx_unique_active_user_email
  ON users(email)
  WHERE deleted_at IS NULL;

-- +goose Down
DROP INDEX IF EXISTS idx_unique_active_user_email;

ALTER TABLE users
  ADD CONSTRAINT users_email_key UNIQUE (email);
```

Note: Once duplicate email addresses exist among soft-deleted users, rolling this migration back requires resolving those duplicates before the original global unique constraint can be restored.

What does this SQL do?

1. It removes the old global unique constraint: *users_email_key*.
2. And then it replaces it with a partial unique index. This allows multiple users with the same email if they are soft-deleted, and only active users (*deleted_at* is *NULL*) need to be unique.

Now apply it to the database:

```
goose -dir db/migrations postgres "postgres://<username>:<password>@<hostname>:<portnumber>/<databasename>?sslmode=disable" up
```

Because of this we must modify the *CreateUser()* method in *userhandlers.go*:

```
func (handler *UserHandler) CreateUser(w http.ResponseWriter, r *http.Request) {
    var req CreateUserRequest

    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        log.Printf("Error decoding request body: %v", err)
        writeError(w, http.StatusBadRequest, "Invalid request payload")
        return
    }
    if err := req.Validate(); err != nil {
        writeError(w, http.StatusBadRequest, err.Error())
        return
    }
    user, err := handler.queries.CreateUser(r.Context(), db.CreateUserParams{
        Name: req.Name,
        Email: req.Email,
    })

    if err != nil {
        var pgErr *pgconn.PgError
        if errors.As(err, &pgErr) && pgErr.Code == "23505" {
            writeError(w, http.StatusConflict, "Email already exists")
            return
        }
        log.Printf("Error creating user: %v", err)
        writeError(w, http.StatusInternalServerError, "Failed to create user")
        return
    }
    writeJSON(w, http.StatusCreated, models.ToUserResponse(user))
}
```

We will have to do the same for the *UpdateUser()* method:

```
func (handler *UserHandler) UpdateUser(w http.ResponseWriter, r *http.Request) {
    id, err := parseID(r)
    if err != nil {
        log.Printf("Error parsing ID: %v", err)
        writeError(w, http.StatusBadRequest, "Invalid user ID")
        return
    }

    var req UpdateUserRequest
    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        log.Printf("Error decoding request body: %v", err)
        writeError(w, http.StatusBadRequest, "Invalid request payload")
        return
    }

    if err := req.Validate(); err != nil {
        writeError(w, http.StatusBadRequest, err.Error())
        return
    }

    // Update the user
    user, err := handler.queries.UpdateUser(r.Context(), db.UpdateUserParams{
        ID:      id,
        Name:    req.Name,
        Email:   req.Email,
    })

    if err != nil {
        if errors.Is(err, pgx.ErrNoRows) {
            writeError(w, http.StatusNotFound, "User not found")
            return
        }
        var pgErr *pgconn.PgError
        if errors.As(err, &pgErr) && pgErr.Code == "23505" {
            writeError(w, http.StatusConflict, "Email already exists")
            return
        }
        log.Printf("Error updating user: %v", err)
        writeError(w, http.StatusInternalServerError, "Failed to update user")
        return
    }

    writeJSON(w, http.StatusOK, models.ToUserResponse(user))
}
```

The main and only difference is that the methods now return *409 Conflict* if the email already exists.

Also do not forget to add this to the imports section:

"github.com/jackc/pgx/v5/pgconn"

This package contains the *pgconn.PgError*.

Testing the API

When testing this version of the API, use fresh values in *book.json* and *user.json* to avoid conflicts with data created during earlier runs.

Note: `--data-binary` has a double hyphen.

First create a new book:

```
curl -i -X POST http://localhost:3000/books -H "Content-Type: application/json" --data-binary "@book.json"
```

And you should get a response like this:

HTTP/1.1 201 Created

Content-Type: application/json

Date: Thu, 10 Sep 2026 07:34:25 GMT

Content-Length: 243

```
{"id":1,"title":"The Go Programming Language","author":"Alan Donovan","description":"A very comprehensive guide to Go","year_of_publication":2015,"created_at":"2026-09-10T09:34:25.046133+02:00","updated_at":"2026-09-10T09:34:25.046133+02:00"}
```

Next create a new user:

```
curl -i -X POST http://localhost:3000/users -H "Content-Type: application/json" --data-binary "@user.json"
```

And you should see a response like this:

HTTP/1.1 201 Created

Content-Type: application/json

Date: Thu, 10 Sep 2026 07:35:53 GMT

Content-Length: 155

```
{"id":1,"name":"Test user","email":"usertest@example.com","created_at":"2026-09-10T09:35:53.296814+02:00","updated_at":"2026-09-10T09:35:53.296814+02:00"}
```

To demonstrate the constraints we have on the users table, issue the previous command again, and you should see this:

```
HTTP/1.1 409 Conflict
Content-Type: application/json
Date: Thu, 10 Sep 2026 07:36:23 GMT
Content-Length: 33
```

```
{"error": "Email already exists"}
```

As you can see, we now get a 409 error code, indicating a conflict.

For testing purposes we need to create a second book. Create a *book2.json* file with these contents:

```
{
  "title": "Design Patterns: Elements of Reusable Object-Oriented Software",
  "author": "Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides",
  "description": "A very comprehensive guide to design patterns",
  "year_of_publication": 1994
}
```

Use this command to create it:

```
curl -i -X POST http://localhost:3000/books -H "Content-Type: application/json" --data-binary "@book2.json"
```

The output should be something like this:

```
HTTP/1.1 201 Created
Content-Type: application/json
Date: Thu, 10 Sep 2026 07:43:21 GMT
Content-Length: 335

{"id":2,"title":"Design Patterns: Elements of Reusable Object-Oriented Software","author":"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides","description":"A very comprehensive guide to design patterns","year_of_publication":1994,"created_at":"2026-09-10T09:43:21.769076+02:00","updated_at":"2026-09-10T09:43:21.769076+02:00"}
```

Now borrow both books. Update *borrow.json* with the new user ID, and the ID of the first book, call the *localhost:3000/books/borrow* endpoint, then repeat it with the second book ID.

Call the endpoint like this:

```
curl -i -X POST http://localhost:3000/books/borrow -H "Content-Type: application/json" --data-binary "@borrow.json"
```

You should see a response like this:

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu, 10 Sep 2026 07:45:12 GMT
Content-Length: 147

{"id":2,"book_id":2,"user_id":1,"borrowed_at":"2026-09-10T09:45:12.246186+02:00","due_date":"2026-10-10T09:45:12.246847+02:00","returned_at":null}
```

Now check if the books are really borrowed (replace the user ID at the end with the ID of your desired user):

```
curl -i -X GET http://localhost:3000/books/user/1/borrowed
```

And you should see this if the borrowing has gone right:

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
Date: Thu, 10 Sep 2026 07:46:35 GMT
```

```
Content-Length: 499
```

```
[{"id":2,"book_id":2,"user_id":1,"borrowed_at":"2026-09-10T09:45:12.246186+02:00","due_date":"2026-10-10T09:45:12.246847+02:00","returned_at":null,"title":"Design Patterns: Elements of Reusable Object-Oriented Software","author":"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides"},{"id":1,"book_id":1,"user_id":1,"borrowed_at":"2026-09-10T09:44:56.792113+02:00","due_date":"2026-10-10T09:44:56.793656+02:00","returned_at":null,"title":"The Go Programming Language","author":"Alan Donovan"}]
```

Let's return a book:

```
curl -i -X POST http://localhost:3000/books/return -H "Content-Type: application/json" --data-binary "@return.json"
```

And you should see output like:

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
Date: Thu, 10 Sep 2026 07:47:25 GMT
```

```
Content-Length: 72
```

```
{"id":1,"book_id":1,"user_id":1,"message":"Book returned successfully"}
```

Now let's delete the user (replace the user ID with the ID of the user you want to delete):

```
curl -i -X DELETE http://localhost:3000/users/1
```

And you should see the No Content response:

```
HTTP/1.1 204 No Content
```

```
Date: Thu, 10 Sep 2026 07:48:21 GMT
```

And check if the user is still in the list of users:

```
curl -i -X GET http://localhost:3000/users
```

And the user with the ID you just deleted should no longer be there:

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
Date: Thu, 10 Sep 2026 07:49:08 GMT
```

```
Content-Length: 3
```

```
[]
```

Let's inspect the borrowing history, which may provide some insight (replace the ID with the appropriate user ID):

```
curl -i -X GET http://localhost:3000/users/1/history
```

And this should return something like this:

HTTP/1.1 200 OK

Content-Type: application/json

Date: Thu, 10 Sep 2026 07:52:00 GMT

Content-Length: 529

```
[{"id":2,"book_id":2,"user_id":1,"borrowed_at":"2026-09-10T09:45:12.246186+02:00","due_date":"2026-10-10T09:45:12.246847+02:00","returned_at":null,"title":"Design Patterns: Elements of Reusable Object-Oriented Software","author":"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides"}, {"id":1,"book_id":1,"user_id":1,"borrowed_at":"2026-09-10T09:44:56.792113+02:00","due_date":"2026-10-10T09:44:56.793656+02:00","returned_at":"2026-09-10T09:47:25.337267+02:00","title":"The Go Programming Language","author":"Alan Donovan"}]
```

We see two books for this deleted user: one has been returned and the other has not been returned.

Notice that the outstanding loan record still exists even though the user has been soft-deleted.

Depending on the application's legal or privacy requirements, borrowing history may also need a defined retention period. A production system could periodically remove or anonymize records that have exceeded that period.

Summary

In this chapter we took an API that worked in simple, sequential use and asked a much more important question: what happens when operations fail or multiple requests arrive at the same time?

The borrowing process was a good example. In the previous chapter, checking whether a book was available, creating a borrow record and marking the book as unavailable were separate database operations. That left a race window in which two concurrent requests could both see the same book as being available. By wrapping the complete operation in a transaction and using *SELECT ... FOR UPDATE*, we made those changes atomic and ensured that competing requests cannot borrow the same book at the same time. We then added a partial unique index to PostgreSQL, so the database itself also enforces the rule that a book can only have one active borrow record.

We also improved the way the API represents database values. A nullable *returned_at* value should not appear as Go's zero *time.Time* value in JSON, so the response model now uses **time.Time* and returns null when the book has not yet been returned.

Another important change was the introduction of soft deletes. Instead of physically removing books and users from the database, we mark them with a *deleted_at* timestamp and filter deleted records from normal queries. This allows us to preserve the borrowing records that refer to those users and books. On top of that we added a dedicated borrowing-history query and endpoint, so current loans and historical loans are now two separate concepts in the API.

Finally, soft deletion introduced another subtle database problem: a deleted user could still prevent a new active user from registering with the same email address. We solved that at the database level by replacing the global unique constraint with a partial unique index that applies only to active users.

The important lesson from this chapter is that **an API that works is not automatically an API that is robust**. Transactions protect groups of related changes, row locks coordinate concurrent access, database constraints enforce invariants, and nullable response types accurately represent missing values. Soft deletion preserves history while keeping deleted books and users out of normal queries.

At this point the application has become considerably more resilient, but we still need confidence that all of this behaviour continues to work as the code evolves. In the next chapter we will add automated tests and see how the interfaces introduced earlier make the API much easier to test.