Diffchecker

Untitled diff

15 removals 223 lines

```
use actix_web::{App, HttpResponse, HttpServer, Responder, get, post, web};

use futures::future::join_all;
use reqwest::Client;
use rusqlite::{Connection, params};
use serde::Deserialize;
use serde::Serialize;
use std::fs::{self, File};
use std::io::copy;
use std::path::Path;
use std::process::Command;
use std::sync::Arc;
use std::sync::Mutex;
use tokio::sync::Semaphore;

#[derive(Debug, Serialize)]
struct Item {
    hash: String,
    title: String,
    dt: String,
    cat: String,
    size: Option<i64>,
}
#[derive(Serialize, Deserialize, Debug)]
struct ImageData {
    title: String,
    img_url_array: Vec<String>,
    page_url: String,
}
#[derive(Deserialize, Debug)]
struct SearchRequest {
    titles: Vec<String>,
}
struct AppState {
    conn: Mutex<Connection>,
```

24 additions 239 lines

```
use actix_web::{App, HttpResponse, HttpServer, Responder, get, post, web};
use deadpool_postgres::{Config, Pool, Runtime};
use futures::future::join_all;
use reqwest::Client;
use rusqlite::{Connection, params};
use serde::{Deserialize, Serialize};
use std::fs;

use std::io::copy;
use std::path::Path;
use std::process::Command;
use std::sync::Arc;
use std::sync::Mutex;
use tokio::sync::Semaphore;
use tokio_postgres::NoTls;
#[derive(Debug, Serialize)]
struct Item {
    hash: String,
    title: String,
    dt: String,
    cat: String,
    size: Option<i64>,
}
#[derive(Serialize, Deserialize, Debug)]
struct ImageData {
    title: String,
    img_url_array: Vec<String>,
    page_url: String,
}
#[derive(Deserialize, Debug)]
struct SearchRequest {
    titles: Vec<String>,
}
struct AppState {
    conn: Mutex<Connection>,
}

fn process_search_term(term: &str) -> String {
    let term = term.split_whitespace().collect::<Vec<_>>().join(" ");
    let term = term.replace(" ", ".%.");
    format!("{}.", term)
}
async fn download_image(url: &str, path: &Path) -> Result<(), String> {
    let client = Client::builder()
        .no_proxy()
        .build()
        .map_err(|e| e.to_string())?;
    let response = client.get(url).send().await.map_err(|e| e.to_string())?;
    if !response.status().is_success() {
        return Err(format!("Failed to download image: {}", response.status()));
    }
    let content = response.bytes().await.map_err(|e| e.to_string())?;
    if content.is_empty() {
        return Err("Downloaded file is empty".to_string());
    }
    let mut file = fs::File::create(path).map_err(|e| e.to_string())?;
    copy(&mut content.as_ref(), &mut file).map_err(|e| e.to_string())?;
    Ok(())
}
#[get("/rarbg")]
async fn get_items(
    data: web::Data<AppState>,
    query: web::Query<std::collections::HashMap<String, String>>,
) -> impl Responder {
    let conn = data.conn.lock().unwrap();
```

```
}
struct PgAppState {
    pool: Pool,
}
fn process_search_term(term: &str) -> String {
    let term = term.split_whitespace().collect::<Vec<_>>().join(" ");
    let term = term.replace(" ", ".%.");
    format!("{}.", term)
}
async fn download_image(url: &str, path: &Path) -> Result<(), String> {
    let client = Client::builder()
        .no_proxy()
        .build()
        .map_err(|e| e.to_string())?;
    let response = client.get(url).send().await.map_err(|e| e.to_string())?;
    if !response.status().is_success() {
        return Err(format!("Failed to download image: {}", response.status()));
    }
    let content = response.bytes().await.map_err(|e| e.to_string())?;
    if content.is_empty() {
        return Err("Downloaded file is empty".to_string());
    }
    let mut file = fs::File::create(path).map_err(|e| e.to_string())?;
    copy(&mut content.as_ref(), &mut file).map_err(|e| e.to_string())?;
    Ok(())
}
#[get("/rarbg")]
async fn get_items(
    data: web::Data<AppState>,
    query: web::Query<std::collections::HashMap<String, String>>,
) -> impl Responder {
    let conn = data.conn.lock().unwrap();
```

```rust
    let title_filter = query.get("title").map(|s| s.as_str());
    let query_str = match title_filter {
        Some(title) => {
            let processed_title = process_search_term(title);
            format!(
                "SELECT hash, title, dt, cat, size FROM items WHERE LOWER(title) LIKE LOWER('%{}%') ORDER BY title ASC LIMIT 10000",
                processed_title
            )
        }
        None => "SELECT hash, title, dt, cat, size FROM items ORDER BY title ASC LIMIT 10000"
            .to_string(),
    };
    let mut stmt = conn.prepare(&query_str).unwrap();
    let item_iter = stmt
        .query_map(params![], |row| {
            Ok(Item {
                hash: row.get(0)?,
                title: row.get(1)?,
                dt: row.get(2)?,
                cat: row.get(3)?,
                size: row.get(4)?,
            })
        })
        .unwrap();
    let mut items = Vec::new();
    for item in item_iter {
        items.push(item.unwrap());
    }
    HttpResponse::Ok().json(items)
}
#[post("/rarbg/batch")]
async fn get_items_batch(
    data: web::Data<AppState>,
    search_request: web::Json<SearchRequest>,
) -> impl Responder {
    let conn = data.conn.lock().unwrap();
    let titles = &search_request.titles;
    let mut query_str = String::from("SELECT hash, title, dt,
```

```rust
    let title_filter = query.get("title").map(|s| s.as_str());
    let query_str = match title_filter {
        Some(title) => {
            let processed_title = process_search_term(title);
            format!(
                "SELECT hash, title, dt, cat, size FROM items WHERE LOWER(title) LIKE LOWER('%{}%') ORDER BY title ASC LIMIT 10000",
                processed_title
            )
        }
        None => "SELECT hash, title, dt, cat, size FROM items ORDER BY title ASC LIMIT 10000"
            .to_string(),
    };
    let mut stmt = conn.prepare(&query_str).unwrap();
    let item_iter = stmt
        .query_map(params![], |row| {
            Ok(Item {
                hash: row.get(0)?,
                title: row.get(1)?,
                dt: row.get(2)?,
                cat: row.get(3)?,
                size: row.get(4)?,
            })
        })
        .unwrap();
    let mut items = Vec::new();
    for item in item_iter {
        items.push(item.unwrap());
    }
    HttpResponse::Ok().json(items)
}
#[post("/rarbg/batch")]
async fn get_items_batch(
    data: web::Data<AppState>,
    search_request: web::Json<SearchRequest>,
) -> impl Responder {
    let conn = data.conn.lock().unwrap();
    let titles = &search_request.titles;
    let mut query_str = String::from("SELECT hash, title, dt,
cat, size FROM items WHERE ");
    for (index, title) in titles.iter().enumerate() {
        let processed_title = process_search_term(title);
        if index > 0 {
            query_str.push_str(" OR ");
        }
        query_str.push_str(&format!("LOWER(title) LIKE LOWER('%{}%')", processed_title));
    }
    query_str.push_str(" ORDER BY title ASC LIMIT 10000");
    let mut stmt = conn.prepare(&query_str).unwrap();
    let item_iter = stmt
        .query_map(params![], |row| {
            Ok(Item {
                hash: row.get(0)?,
                title: row.get(1)?,
                dt: row.get(2)?,
                cat: row.get(3)?,
                size: row.get(4)?,
            })
        })
        .unwrap();
    let mut items = Vec::new();
    for item in item_iter {
        items.push(item.unwrap());
    }
    HttpResponse::Ok().json(items)
}
#[post("/zup")]
async fn handle_post(data: web::Json<ImageData>) -> impl Responder {
    let title = &data.title;
    let page_url = &data.page_url;
    let base_dir = Path::new("C:\\Users\\aa\\Desktop\\zup");
    let dir_path = base_dir.join(title);
    if !dir_path.exists() {
        fs::create_dir_all(&dir_path).expect("Failed to create directory");
    }
    let total_count = data.img_url_array.len();
```

```
cat, size FROM items WHERE ");
    for (index, title) in titles.iter().enumerate() {
        let processed_title = process_search_term(title);
        if index > 0 {
            query_str.push_str(" OR ");
        }
        query_str.push_str(&format!("LOWER(title) LIKE LOWER('%{}%')", processed_title));
    }
    query_str.push_str(" ORDER BY title ASC LIMIT 10000");
    let mut stmt = conn.prepare(&query_str).unwrap();
    let item_iter = stmt
        .query_map(params![], |row| {
            Ok(Item {
                hash: row.get(0)?,
                title: row.get(1)?,
                dt: row.get(2)?,
                cat: row.get(3)?,
                size: row.get(4)?,
            })
        })
        .unwrap();
    let mut items = Vec::new();
    for item in item_iter {
        items.push(item.unwrap());
    }
    HttpResponse::Ok().json(items)
}
#[post("/zup")]
async fn handle_post(data: web::Json<ImageData>) -> impl Responder {
    let title = &data.title;
    let page_url = &data.page_url;
    let base_dir = Path::new("C:\\Users\\aa\\Desktop\\zup");
    let dir_path = base_dir.join(title);
    if !dir_path.exists() {
        fs::create_dir_all(&dir_path).expect("Failed to create directory");
    }
    let total_count = data.img_url_array.len();
    let success_count = Arc::new(std::sync::atomic::AtomicUsize::new(0));
    let mut failed_urls = Vec::new();
    let semaphore = Arc::new(Semaphore::new(8));
    let mut tasks = Vec::new();
    for (index, url) in data.img_url_array.iter().enumerate() {
        let file_name = format!("{:04}.jpg", index + 1);
        let file_path = dir_path.join(&file_name);
        let url = url.clone();
        let semaphore = semaphore.clone();
        let success_count = success_count.clone();
        tasks.push(tokio::spawn(async move {
            let _permit = semaphore.acquire().await.unwrap();
            if file_path.exists() {
                return Ok(());
            }
            match download_image(&url, &file_path).await {
                Ok(_) => {
                    let current_count =
                        success_count.fetch_add(1, std::sync::atomic::Ordering::SeqCst);
                    let progress = ((current_count + 1) as f32 / total_count as f32) * 100.0;
                    println!("Download progress: {:.2}%", progress);
                    Ok(())
                }
                Err(e) => {
                    eprintln!("Failed to download {}: {}", url, e);
                    Err(url)
                }
            }
        }));
    }
```

```
    let success_count = Arc::new(std::sync::atomic::AtomicUsize::new(0));
    let mut failed_urls = Vec::new();
    let semaphore = Arc::new(Semaphore::new(8));
    let mut tasks = Vec::new();
    for (index, url) in data.img_url_array.iter().enumerate() {
        let file_name = format!("{:04}.jpg", index + 1);
        let file_path = dir_path.join(&file_name);
        let url = url.clone();
        let semaphore = semaphore.clone();
        let success_count = success_count.clone();
        tasks.push(tokio::spawn(async move {
            let _permit = semaphore.acquire().await.unwrap();
            if file_path.exists() {
                return Ok(());
            }
            match download_image(&url, &file_path).await {
                Ok(_) => {
                    let current_count =
                        success_count.fetch_add(1, std::sync::atomic::Ordering::SeqCst);
                    let progress = ((current_count + 1) as f32 / total_count as f32) * 100.0;
                    println!("Download progress: {:.2}%", progress);
                    Ok(())
                }
                Err(e) => {
                    eprintln!("Failed to download {}: {}", url, e);
                    Err(url)
                }
            }
        }));
    }
    let results = join_all(tasks).await;
    for result in results {
        if let Ok(Err(url)) = result {
            failed_urls.push(url);
        }
    }
    println!("{}\n已完成！", title);
    if !failed_urls.is_empty() {
        let html_content = format!(
            r#"<html>
                <body>
                    <h1><a href="{}">{}</a></h1>
                    <ul>
                        {}
                    </ul>
                </body>
            </html>"#,
            page_url,
            title,
            failed_urls
                .iter()
                .map(|url| format!("<li><a href=\"{}\">{}</a></li>", url, url))
                .collect::<Vec<_>>()
                .join("")
        );
        fs::write(dir_path.join("failed_downloads.html"), html_content)
            .expect("Failed to write HTML file");
    } else {
        let failed_file_path = dir_path.join("failed_downloads.html");
        if failed_file_path.exists() {
            fs::remove_file(failed_file_path).expect("Failed to delete failed_downloads.html");
        }
    }
    let _ = Command::new("C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe")
```

```
    let results = join_all(tasks).await;
    for result in results {
        if let Ok(Err(url)) = result {
            failed_urls.push(url);
        }
    }
    println!("{}\n已完成！", title);
    if !failed_urls.is_empty() {
        let html_content = format!(
            r#"<html>
            <body>
                <h1><a href="{}">{}</a></h1>
                <ul>
                    {}
                </ul>
            </body>
        </html>"#,
            page_url,
            title,
            failed_urls
                .iter()
                .map(|url| format!("<li><a href=\"{}\">{}</a></li>", url, url))
                .collect::<Vec<_>>()
                .join("")
        );
        fs::write(dir_path.join("failed_downloads.html"), html_content)
            .expect("Failed to write HTML file");
    } else {
        let failed_file_path = dir_path.join("failed_downloads.html");
        if failed_file_path.exists() {
            fs::remove_file(failed_file_path).expect("Failed to delete failed_downloads.html");
        }
    }
    let _ = Command::new("C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe")
```

```
.arg(dir_path.to_str().unwrap())
        .output();
    HttpResponse::Ok().body(format!("{}\n已完成！", title))
}
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let db_path = "C:\\Users\\aa\\Downloads\\rarbg_db\\rarbg_db.sqlite";
    let conn = Connection::open(db_path).unwrap();
    let app_state = web::Data::new(AppState {
        conn: Mutex::new(conn),
    });
    HttpServer::new(move || {
        App::new()
.app_data(app_state.clone())
            .service(get_items)
            .service(get_items_batch)
```

```
.arg(dir_path.to_str().unwrap())
        .output();
    HttpResponse::Ok().body(format!("{}\n已完成！", title))
}
async fn init_pool() -> Pool {
    let mut cfg = Config::new();
    cfg.host = Some("localhost".to_string());
    cfg.user = Some("postgres".to_string());
    cfg.password = Some("4545".to_string());
    cfg.dbname = Some("your_database_name".to_string());
cfg.create_pool(Some(Runtime::Tokio1), NoTls).unwrap()
}
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let sqlite_conn = Connection::open("C:\\Users\\aa\\Downloads\\rarbg_db\\rarbg_db.sqlite").unwrap();
    let pg_pool = init_pool().await;
    let app_state = web::Data::new(AppState {
        conn: Mutex::new(sqlite_conn),
    });
    let pg_app_state = web::Data::new(PgAppState { pool: pg_pool });
    HttpServer::new(move || {
        App::new()
.app_data(app_state.clone())
.app_data(pg_app_state.clone())
            .service(get_items)
            .service(get_items_batch)
```

```
                .service(handle_post)

        })
        .bind("127.0.0.1:46644")?
        .run()
        .await
}

```

```
                .service(handle_post)
.service(get_items_batch_pq)
        })
        .bind("127.0.0.1:46644")?
        .run()
        .await
}

```