



Untitled diff

- 15 removals

223 lines

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

2 use futures::future::join_all;
3 use request::Client;
4 use rusqlite::{Connection, params};
5 use serde::Deserialize;
6 use serde::Serialize;
7 use std::fs::{self, File};

8 use std::io::copy;
9 use std::path::Path;
10 use std::process::Command;
11 use std::sync::Arc;
12 use std::sync::Mutex;
13 use tokio::sync::Semaphore;

14 #[derive(Debug, Serialize)]
15 struct Item {
16     hash: String,
17     title: String,
18     dt: String,
19     cat: String,
20     size: Option<i64>,
21 }
22 #[derive(Serialize, Deserialize,
  Debug)]
23 struct ImageData {
24     title: String,
25     img_url_array: Vec<String>,
26     page_url: String,
27 }
28 #[derive(Deserialize, Debug)]
29 struct SearchRequest {
30     titles: Vec<String>,
31 }
32 struct AppState {
33     conn: Mutex<Connection>,
```

+ 24 additions

239 lines

```
1 use actix_web::{App, HttpResponse,
  HttpServer, Responder, get, post,
  web};
2 use deadpool_postgres::{Config, Pool,
  Runtime};
3 use futures::future::join_all;
4 use request::Client;
5 use rusqlite::{Connection, params};
6 use serde::{Deserialize, Serialize};
7 use std::fs;

8 use std::io::copy;
9 use std::path::Path;
10 use std::process::Command;
11 use std::sync::Arc;
12 use std::sync::Mutex;
13 use tokio::sync::Semaphore;
14 use tokio_postgres::NoTls;

15 #[derive(Debug, Serialize)]
16 struct Item {
17     hash: String,
18     title: String,
19     dt: String,
20     cat: String,
21     size: Option<i64>,
22 }
23 #[derive(Serialize, Deserialize,
  Debug)]
24 struct ImageData {
25     title: String,
26     img_url_array: Vec<String>,
27     page_url: String,
28 }
29 #[derive(Deserialize, Debug)]
30 struct SearchRequest {
31     titles: Vec<String>,
32 }
33 struct AppState {
34     conn: Mutex<Connection>,
```

```

34 }

35 fn process_search_term(term: &str) ->
String {
36     let term =
term.split_whitespace().collect::
<Vec<_>>().join(" ");
37     let term = term.replace(" ",
".%.");
38     format!("{}", term)
39 }
40 async fn download_image(url: &str,
path: &Path) -> Result<(), String> {
41     let client = Client::builder()
42         .no_proxy()
43         .build()
44         .map_err(|e| e.to_string())?;
45     let response =
client.get(url).send().await.map_err(
|e| e.to_string())?;
46     if
!response.status().is_success() {
47         return Err(format!("Failed to
download image: {} ",
response.status()));
48     }
49     let content =
response.bytes().await.map_err(|e|
e.to_string())?;
50     if content.is_empty() {
51         return Err("Downloaded file
is empty".to_string());
52     }
53     let mut file =
fs::File::create(path).map_err(|e|
e.to_string())?;
54     copy(&mut content.as_ref(), &mut
file).map_err(|e| e.to_string())?;
55     Ok(())
56 }
57 #[get("/rarbg")]
58 async fn get_items(
59     data: web::Data<AppState>,
60     query:
web::Query<std::collections::HashMap<
String, String>>,
61 ) -> impl Responder {
62     let conn =
data.conn.lock().unwrap();

```

```

35 }

36 struct PgAppState {
37     pool: Pool,
38 }

39 fn process_search_term(term: &str) ->
String {
40     let term =
term.split_whitespace().collect::
<Vec<_>>().join(" ");
41     let term = term.replace(" ",
".%.");
42     format!("{}", term)
43 }
44 async fn download_image(url: &str,
path: &Path) -> Result<(), String> {
45     let client = Client::builder()
46         .no_proxy()
47         .build()
48         .map_err(|e| e.to_string())?;
49     let response =
client.get(url).send().await.map_err(
|e| e.to_string())?;
50     if
!response.status().is_success() {
51         return Err(format!("Failed to
download image: {} ",
response.status()));
52     }
53     let content =
response.bytes().await.map_err(|e|
e.to_string())?;
54     if content.is_empty() {
55         return Err("Downloaded file
is empty".to_string());
56     }
57     let mut file =
fs::File::create(path).map_err(|e|
e.to_string())?;
58     copy(&mut content.as_ref(), &mut
file).map_err(|e| e.to_string())?;
59     Ok(())
60 }
61 #[get("/rarbg")]
62 async fn get_items(
63     data: web::Data<AppState>,
64     query:
web::Query<std::collections::HashMap<
String, String>>,
65 ) -> impl Responder {
66     let conn =
data.conn.lock().unwrap();

```

```

63     let title_filter =
        query.get("title").map(|s|
            s.as_str());
64     let query_str = match
        title_filter {
65         Some(title) => {
66             let processed_title =
                process_search_term(title);
67             format!(
68                 "SELECT hash, title,
                dt, cat, size FROM items WHERE
                LOWER(title) LIKE LOWER('%{}%') ORDER
                BY title ASC LIMIT 10000",
69                 processed_title
70             )
71         }
72         None => "SELECT hash, title,
                dt, cat, size FROM items ORDER BY
                title ASC LIMIT 10000"
73             .to_string(),
74     };
75     let mut stmt =
        conn.prepare(&query_str).unwrap();
76     let item_iter = stmt
77         .query_map(params![], |row| {
78             Ok(Item {
79                 hash: row.get(0)?,
80                 title: row.get(1)?,
81                 dt: row.get(2)?,
82                 cat: row.get(3)?,
83                 size: row.get(4)?,
84             })
85         })
86         .unwrap();
87     let mut items = Vec::new();
88     for item in item_iter {
89         items.push(item.unwrap());
90     }
91     HttpResponse::Ok().json(items)
92 }
93 #[post("/rarbg/batch")]
94 async fn get_items_batch(
95     data: web::Data<AppState>,
96     search_request:
        web::Json<SearchRequest>,
97 ) -> impl Responder {
98     let conn =
        data.conn.lock().unwrap();
99     let titles =
        &search_request.titles;
100     let mut query_str =
        String::from("SELECT hash, title, dt,

```

```

67     let title_filter =
        query.get("title").map(|s|
            s.as_str());
68     let query_str = match
        title_filter {
69         Some(title) => {
70             let processed_title =
                process_search_term(title);
71             format!(
72                 "SELECT hash, title,
                dt, cat, size FROM items WHERE
                LOWER(title) LIKE LOWER('%{}%') ORDER
                BY title ASC LIMIT 10000",
73                 processed_title
74             )
75         }
76         None => "SELECT hash, title,
                dt, cat, size FROM items ORDER BY
                title ASC LIMIT 10000"
77             .to_string(),
78     };
79     let mut stmt =
        conn.prepare(&query_str).unwrap();
80     let item_iter = stmt
81         .query_map(params![], |row| {
82             Ok(Item {
83                 hash: row.get(0)?,
84                 title: row.get(1)?,
85                 dt: row.get(2)?,
86                 cat: row.get(3)?,
87                 size: row.get(4)?,
88             })
89         })
90         .unwrap();
91     let mut items = Vec::new();
92     for item in item_iter {
93         items.push(item.unwrap());
94     }
95     HttpResponse::Ok().json(items)
96 }
97 #[post("/rarbg/batch")]
98 async fn get_items_batch(
99     data: web::Data<AppState>,
100     search_request:
        web::Json<SearchRequest>,
101 ) -> impl Responder {
102     let conn =
        data.conn.lock().unwrap();
103     let titles =
        &search_request.titles;
104     let mut query_str =
        String::from("SELECT hash, title, dt,

```

```

        cat, size FROM items WHERE ");
101     for (index, title) in
        titles.iter().enumerate() {
102         let processed_title =
        process_search_term(title);
103         if index > 0 {
104             query_str.push_str(" OR
        ");
105         }
106         query_str.push_str(&format!
        ("LOWER(title) LIKE LOWER('%{}%')",
        processed_title));
107     }
108     query_str.push_str(" ORDER BY
        title ASC LIMIT 10000");
109     let mut stmt =
        conn.prepare(&query_str).unwrap();
110     let item_iter = stmt
        .query_map(params![], |row| {
111         Ok(Item {
112             hash: row.get(0)?,
113             title: row.get(1)?,
114             dt: row.get(2)?,
115             cat: row.get(3)?,
116             size: row.get(4)?,
117         })
118     })
119     .unwrap();
120     let mut items = Vec::new();
121     for item in item_iter {
122         items.push(item.unwrap());
123     }
124     HttpResponse::Ok().json(items)
125 }
126 #[post("/zup")]
127 async fn handle_post(data:
        web::Json<ImageData>) -> impl
        Responder {
128     let title = &data.title;
129     let page_url = &data.page_url;
130     let base_dir =
        Path::new("C:\\Users\\aa\\Desktop\\zu
        p");
131     let dir_path =
        base_dir.join(title);
132     if !dir_path.exists() {
133         fs::create_dir_all(&dir_path).expect(
        "Failed to create directory");
134     }
135     let total_count =
        data.img_url_array.len();

```

```

        cat, size FROM items WHERE ");
105     for (index, title) in
        titles.iter().enumerate() {
106         let processed_title =
        process_search_term(title);
107         if index > 0 {
108             query_str.push_str(" OR
        ");
109         }
110         query_str.push_str(&format!
        ("LOWER(title) LIKE LOWER('%{}%')",
        processed_title));
111     }
112     query_str.push_str(" ORDER BY
        title ASC LIMIT 10000");
113     let mut stmt =
        conn.prepare(&query_str).unwrap();
114     let item_iter = stmt
        .query_map(params![], |row| {
115         Ok(Item {
116             hash: row.get(0)?,
117             title: row.get(1)?,
118             dt: row.get(2)?,
119             cat: row.get(3)?,
120             size: row.get(4)?,
121         })
122     })
123     .unwrap();
124     let mut items = Vec::new();
125     for item in item_iter {
126         items.push(item.unwrap());
127     }
128     HttpResponse::Ok().json(items)
129 }
130 #[post("/zup")]
131 async fn handle_post(data:
        web::Json<ImageData>) -> impl
        Responder {
132     let title = &data.title;
133     let page_url = &data.page_url;
134     let base_dir =
        Path::new("C:\\Users\\aa\\Desktop\\zu
        p");
135     let dir_path =
        base_dir.join(title);
136     if !dir_path.exists() {
137         fs::create_dir_all(&dir_path).expect(
        "Failed to create directory");
138     }
139     let total_count =
        data.img_url_array.len();

```

```

137     let success_count =
138         Arc::new(std::sync::atomic::AtomicUsize::new(0));
139     let mut failed_urls = Vec::new();
140     let semaphore =
141         Arc::new(Semaphore::new(8));
142     let mut tasks = Vec::new();
143     for (index, url) in
144         data.img_url_array.iter().enumerate()
145     {
146         let file_name = format!("{:04}.jpg", index + 1);
147         let file_path =
148             dir_path.join(&file_name);
149         let url = url.clone();
150         let semaphore =
151             semaphore.clone();
152         let success_count =
153             success_count.clone();
154         tasks.push(tokio::spawn(async
155             move {
156                 let _permit =
157                     semaphore.acquire().await.unwrap();
158                 if file_path.exists() {
159                     return Ok(());
160                 }
161                 match
162                     download_image(&url,
163                     &file_path).await {
164                     Ok(_) => {
165                         let current_count
166                             =
167                             success_count.fetch_add(1,
168                             std::sync::atomic::Ordering::SeqCst);
169                         let progress =
170                             ((current_count + 1) as f32 /
171                             total_count as f32) * 100.0;
172                         println!
173                             ("Download progress: {:.2}%",
174                             progress);
175                         Ok(())
176                     }
177                     Err(e) => {
178                         eprintln!("Failed
179                             to download {}: {}", url, e);
180                         Err(url)
181                     }
182                 }
183             }
184         ));
185     }

```

```

141     let success_count =
142         Arc::new(std::sync::atomic::AtomicUsize::new(0));
143     let mut failed_urls = Vec::new();
144     let semaphore =
145         Arc::new(Semaphore::new(8));
146     let mut tasks = Vec::new();
147     for (index, url) in
148         data.img_url_array.iter().enumerate()
149     {
150         let file_name = format!("{:04}.jpg", index + 1);
151         let file_path =
152             dir_path.join(&file_name);
153         let url = url.clone();
154         let semaphore =
155             semaphore.clone();
156         let success_count =
157             success_count.clone();
158         tasks.push(tokio::spawn(async
159             move {
160                 let _permit =
161                     semaphore.acquire().await.unwrap();
162                 if file_path.exists() {
163                     return Ok(());
164                 }
165                 match
166                     download_image(&url,
167                     &file_path).await {
168                     Ok(_) => {
169                         let current_count
170                             =
171                             success_count.fetch_add(1,
172                             std::sync::atomic::Ordering::SeqCst);
173                         let progress =
174                             ((current_count + 1) as f32 /
175                             total_count as f32) * 100.0;
176                         println!
177                             ("Download progress: {:.2}%",
178                             progress);
179                         Ok(())
180                     }
181                     Err(e) => {
182                         eprintln!("Failed
183                             to download {}: {}", url, e);
184                         Err(url)
185                     }
186                 }
187             }
188         ));
189     }

```

```

167     let results =
168         join_all(tasks).await;
169         for result in results {
170             if let Ok(Err(url)) = result
171         {
172             failed_urls.push(url);
173         }
174     }
175     println!("{}", "已完成!", title);
176     if !failed_urls.is_empty() {
177         let html_content = format!(
178             r#"<html>
179
180                 <body>
181                     <h1><a href="{}">
182 {}</a></h1>
183
184                     <ul>
185                         {}
186                     </ul>
187                 </body>
188             </html>"#,
189
190             page_url,
191             title,
192             failed_urls
193                 .iter()
194                 .map(|url| format!(
195                     "<li><a href=\"{}\">{}</a></li>", url,
196                     url))
197                 .collect::<Vec<_>>()
198                 .join("")
199         );
200
201     fs::write(dir_path.join("failed_downl
202 oads.html"), html_content)
203     .expect("Failed to write
204 HTML file");
205 } else {
206     let failed_file_path =
207     dir_path.join("failed_downloads.html"
208 );
209     if failed_file_path.exists()
210     {
211         fs::remove_file(failed_file_path).exp
212 ect("Failed to delete
213 failed_downloads.html");
214     }
215 }
216 let _ = Command::new("C:\\Program
217 Files\\Google\\Chrome\\Application\\c
218 hrome.exe")

```

```

171     let results =
172         join_all(tasks).await;
173         for result in results {
174             if let Ok(Err(url)) = result
175         {
176             failed_urls.push(url);
177         }
178     }
179     println!("{}", "已完成!", title);
180     if !failed_urls.is_empty() {
181         let html_content = format!(
182             r#"<html>
183
184                 <body>
185                     <h1><a href="{}">{}
186 </a></h1>
187
188                     <ul>
189                         {}
190                     </ul>
191                 </body>
192             </html>"#,
193
194             page_url,
195             title,
196             failed_urls
197                 .iter()
198                 .map(|url| format!(
199                     "<li><a href=\"{}\">{}</a></li>", url,
200                     url))
201                 .collect::<Vec<_>>()
202                 .join("")
203         );
204
205     fs::write(dir_path.join("failed_downl
206 oads.html"), html_content)
207     .expect("Failed to write
208 HTML file");
209 } else {
210     let failed_file_path =
211     dir_path.join("failed_downloads.html"
212 );
213     if failed_file_path.exists()
214     {
215         fs::remove_file(failed_file_path).exp
216 ect("Failed to delete
217 failed_downloads.html");
218     }
219 }
220 let _ = Command::new("C:\\Program
221 Files\\Google\\Chrome\\Application\\c
222 hrome.exe")

```

```

201     .arg(dir_path.to_str().unwrap())
202     .output();
203     HttpResponse::Ok().body(format!(
204     {} \n已完成! ", title))
  
```

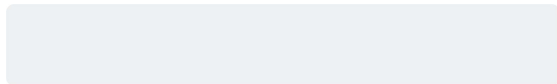
```

205 #[actix_web::main]
206 async fn main() ->
207     std::io::Result<()> {
208     let db_path =
209     "C:\\Users\\aa\\Downloads\\rarbg_db\\
210     rarbg_db.sqlite";
211     let conn =
212     Connection::open(db_path).unwrap();
213
214     let app_state =
215     web::Data::new(AppState {
216     conn: Mutex::new(conn),
217
218     });
219
220     HttpServer::new(move || {
221     App::new()
222     .app_data(app_state.clone())
223
224     .service(get_items)
225     .service(get_items_batch)
  
```

```

205     .arg(dir_path.to_str().unwrap())
206     .output();
207     HttpResponse::Ok().body(format!(
208     {} \n已完成! ", title))
209 }
210
211 async fn init_pool() -> Pool {
212     let mut cfg = Config::new();
213     cfg.host =
214     Some("localhost".to_string());
215     cfg.user =
216     Some("postgres".to_string());
217     cfg.password =
218     Some("4545".to_string());
219     cfg.dbname =
220     Some("your_database_name".to_string()
221     );
222     cfg.create_pool(Some(Runtime::Tokio1)
223     , NoTls).unwrap()
224 }
225
226 #[actix_web::main]
227 async fn main() ->
228     std::io::Result<()> {
229     let sqlite_conn =
230
231     Connection::open("C:\\Users\\aa\\Down
232     loads\\rarbg_db\\rarbg_db.sqlite").un
233     wrap();
234     let pg_pool = init_pool().await;
235
236     let app_state =
237     web::Data::new(AppState {
238     conn:
239     Mutex::new(sqlite_conn),
240
241     });
242
243     let pg_app_state =
244     web::Data::new(PgAppState { pool:
245     pg_pool });
246
247     HttpServer::new(move || {
248     App::new()
249     .app_data(app_state.clone())
250
251     .app_data(pg_app_state.clone())
252
253     .service(get_items)
254     .service(get_items_batch)
  
```

```
217         .service(handle_post)
```



```
218     })
219     .bind("127.0.0.1:46644")?
220     .run()
221     .await
222 }
223
```

```
232         .service(handle_post)
```

```
233     .service(get_items_batch_pq)
```

```
234     })
235     .bind("127.0.0.1:46644")?
236     .run()
237     .await
238 }
239
```