Rust Projekt

API rekomendacji eventów

Wykorzystane technologie

- **Rocket** – web framework
- **neo4rs** – klient do bazy Neo4j
- **Serde** – serializacja/deserializacja struktur danych

Działanie aplikacji

- API rekomendujące wydarzenia (eventy)
- Wykorzystuje bazę grafową **Neo4j**
- Wyszukiwanie, filtrowanie, przypisywanie użytkowników

Struktura aplikacji

```
routes
↓
controllers
↓
services
↓
repo
```

Przykład: prosty GET endpoint

```
#[get("/events")]
async fn get_all(controller: &State<EventController>) -> ApiResponse<Vec<Event>> {
    controller.event_service.get_events().await
}
```

Endpoint z parametrem (automatyczny typ)

```
#[get("/event/<id>")]
async fn get_one(controller: &State<EventController>, id: u16) -> ApiResponse<Event> {
    controller.event_service.get_event(id).await
}
```

Endpoint z dwoma parametrami

```
#[put("/events/<event_id>/attendees/<user_name>")]
async fn assign_user_to_event(
    controller: &State<EventController>,
    event_id: u16,
    user_name: &str,
) -> ApiResponse<String> {
    controller.user_event_service.assign_user_to_event(user_name, event_id).await
}
```

Endpoint z QueryString

```
#[get("/events/filter?<keyword>")]
async fn get_events_by_keywords(
    controller: &State<EventController>,
    keyword: Vec<String>
) -> ApiResponse<Vec<Event>> {
    controller.event_service.get_events_by_keywords(keyword).await
}
```

Pasuje do URLi typu:

```
/events/filter?keyword=test
/events/filter?keyword=test&keyword=sport
```

Pattern matching z `while let`

```
while let Some(row) = match rows.next().await {
    Ok(r) => r,
    Err(e) => return Err(Other(e.to_string())),
} {
    let keyword: String = row.get("k.name")?;
    keywords.push(keyword);
}
```

- Efektywne i eleganckie przetwarzanie danych
- Obsługa błędów przez `Result`

Deserializacja obiektów

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct User {
    name: String
}
```

```
let user: User = row.get("u")?;
```

Serde automatycznie zamienia dane z Neo4j do struktury Rust

`Arc<T>` – współdzielenie połączenia

```
pub struct Neo4jConnection {
    pub graph: Arc<Graph>,
}

let graph = Arc::new(Graph::connect(config).await?);
let user_repo = UserRepository::new(graph.clone());
```

- Klonowanie wskaźnika, nie obiektu
- Bezpieczny dostęp w wielu miejscach aplikacji

Domyślne wartości z `.env`

```
dotenv().ok();
let neo4j_uri = env::var("DB_URI").unwrap_or_else(|| "bolt://neo4j:7687".to_string());
```

- Prosty i skuteczny fallback
- Wygodne konfigurowanie środowiska

Problemy i wyzwania

- Główna trudność: zarządzanie **ownership** i **referencjami**
- Rozwiązanie: użycie `Arc<T>` – współdzielony licznik referencji
- Dodatkowo: dbanie o `Result` i `Error` w wielu warstwach

Podsumowanie

- Rust + Rocket + Neo4j = nowoczesne, bezpieczne API
- Wydajność, bezpieczeństwo typów i pattern matching
- Wyzwania z zarządzaniem własnością – dobrze rozwiązane

Dziękuję za uwagę!