Dokumentacja projektu

Skład grupy:

- Wojciech Dąbek - wdabek@student.agh.edu.pl
- Ida Ciepiela - idaciepiela@student.agh.edu.pl

Z grupy laboratoryjnej nr 5.

Tytuł projektu: *Space Crabs*

Temat projektu: Platforma do tworzenia i rezerwowania kosmicznych ekspedycji / wycieczek turystycznych.

Wykorzystywany SZBD: MongoDB

Technologia realizacji aplikacji: Język Rust - framework Rocket.

Funkcjonalności

1. Każdy niezalogowany:
    - wyświetlać dostępne ekspedycje
    - zalogować się
2. Każdy zalogowany:
    - wyświetlać swoje dane logowania
3. Participant:
    - wyświetlać ekspedycje na które jest zapisany
    - zapisać się na ekspedycję
4. Organizer:
    - wyświetlać ekspedycje które są przez niego organizowane
5. Admin:
    - generowanie raportów
    - wyświetlanie danych użytkowników

Struktura bazy daych

Zdecydowaliśmy się na bazę, która będzie się składać z dwóch kolekcji - *users* i *expeditions*.

Kolekcja *users*

Kolekcja ta zawiera informacje o użytkownikach naszego serwisu. Każdy z nich posiada następujące pola:

- [_id]
- [login] - nazwę użytkownika
- [password] - hasło

- [role] - rolę
  Dodatkowo w zależności od tego jaką rolę będzie miał użytkownik takie dodatkowe pola będą w danym dokumencie. Dostępne role oraz ich pola prezentują się następująco:

1. participant
   - [firstname] - imie
   - [lastname] - nazwisko
   - [my_expeditions] - ekspedycje, na które zapisany jest użytkownik. Każda ekspedycja jest obiektem o następujących polach:
     - [exp_id]
     - [name] - nazwa eksedycji
     - [start_date] - data rozpoczęcia
     - [reserved] - status rezerwacji
       - Z tego zrezygnowaliśmy, ale w bazie w "starych" `user`'ach to pole można zauważyć jako występujące. Nie wpływa to na funkcjonowanie systemu.
     - [paid] - status opłacenia rezerwacji
2. organizer
   - [company_name] - nazwa firmy, z którą powiązany jest użytkownik
   - [contact] - numer kontaktowy
   - [organized_expeditions] - ekspedycje, którą są organizowane przez danego użytkownika. Każda ekspedycja jest obiektem o następujących polach:
     - [exp_id]
     - [name] - nazwa eksedycji
     - [start_date] - data rozpoczęcia
3. admin

Dany użytkownik może więcej niż jedną rolę.

Przykładowe dokumenty z kolekcji *users*:

```
{
  "_id": {
    "$oid": "665dc57f329cd9bc49872586"
  },
  "login": "graceDavis",
  "password": "davisGrace444",
  "role": [
    "participant"
  ],
  "firstname": "Grace",
  "lastname": "Davis",
  "my_expeditions": [
    {
      "exp_id": "665dbcd0e0843652d2629235",
      "name": "Stellaris",
      "start_date": {
        "$numberLong": "2464"
      },
      "reserved": false,
      "paid": false
    }
  ]
}
```

```
{
  "_id": {
    "$oid": "661e39492e1f19d1b89964b7"
  },
  "login": "admin",
  "password": "admin",
  "role": "admin"
}
```

```
{
  "_id": {
    "$oid": "665d9cce329cd9bc4987252b"
  },
  "login": "peterParker",
  "password": "spideyPass456",
  "role": [
    "organizer"
  ],
  "company_name": "ExpeditionExtreme",
  "organized_expeditions": [
    {
      "exp_id": "665dbfaae0843652d2629242",
      "name": "Galactic Trail",
      "start_date": {
        "$numberLong": "2472"
      }
    }
  ],
  "contact": "345-678-901"
}
```

```
{
  "_id": {
    "$oid": "661e39152e1f19d1b89964b6"
  },
  "login": "jasiu",
  "password": "haslo1234",
  "firstname": "Jan",
  "lastname": "Kowalski",
  "role": ["organizer", "participant"],
  "company_name": "Crunchy Cola",
  "contact": "123-456-789",
  "p_expeditions": [
    {
      "exp_id": "234567890",
          "name": "ABC",
          "start_date":2421,
      "reserved": true,
      "paid": false
    },
    {
      "exp_id": "34567890987",
          "name": "MoonRiding",
          "start_date":2433,
      "reserved": true,
      "paid": true
    }
  ],
  "o_expeditions": [
    {
      "exp_id": "234567890",
          "name": "Mars-o-Polo",
          "start_date":2400
    }
  ]
}
```

Kolekcja *expeditions*

Kolekcja ta zawiera informacje na temat ekspedycji
Każdy z dokumentów zawiera następujące pola:

- [_id]
- [name] - nazwa ekspedycji
- [stops] - lista przystanków podczas ekspedycji.
- [max_no_participants] - maksymalna liczba uczestninków
- [guide] - dane przewodnika
  - [firstname]
  - [lastname]
  - [age]
  - [experience]
- [organizer] - dane organizatora
  - [org_id]
  - [company_name]
- [start_time] - czas rozpoczęcia
- [end_time] - czas zakończenia
- [home_station] - stacja macierzysta
- [participants] - lista uczestników. Każdy z uczestników jest obiektem o następujących polach:
  - [user_id]
  - [firstname]
  - [lastname]

    - [paid] - status opłacenia rezerwacji
- [price] - cena ekspedycji

Przykładowe dokumenty z kolekcji *expeditions*:

```
  {
  "_id": {
    "$oid": "665dbcd0e0843652d2629235"
  },
  "name": "Stellaris",
  "stops": [
    "Venus",
    "Alpha Centauri",
    "Jupiter"
  ],
  "max_no_participants": {
    "$numberLong": "150"
  },
  "guide": {
    "firstname": "Elara",
    "lastname": "Nova",
    "age": {
      "$numberLong": "32"
    },
    "experience": "Navigator"
  },
  "organizer": {
    "org_id": "665d9cce329cd9bc49872537",
    "name": "GuardiansTravels"
  },
  "start_time": {
    "$numberLong": "2464"
  },
  "end_time": {
    "$numberLong": "2473"
  },
  "home_station": "Luna Base",
  "participants": [
    {
      "user_id": "665dc57f329cd9bc49872586",
      "firstname": "Grace",
      "lastname": "Davis",
      "paid": false
    }
  ],
  "price": {
    "$numberLong": "15000"
  }
}
```

Dokumenty z kolekcji expeditions przetrzymywane są w strukturach w naszym kodzie

```
#[derive(Debug, Serialize, Deserialize, Clone)] //generate implementation support
for formatting the output, serializing, and deserializing the data structure.
pub struct Expedition {
    #[serde(rename = "_id", skip_serializing_if = "Option::is_none")]
    pub id: Option<ObjectId>,
    pub name: String,
    pub stops: Vec<String>,
    pub max_no_participants: i64,
    pub guide: Guide,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub organizer: Option<Organizer>,
    pub start_time: i64,
    pub end_time: i64,
    pub home_station: String,
    pub participants: Vec<Participant>,
    pub price: i64,
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Guide {
    pub firstname: String,
    pub lastname: String,
    pub age: i64,
    pub experience: String,
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Organizer {
    pub org_id: String,
    pub name: String,
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Participant {
    pub user_id: String,
    pub firstname: String,
    pub lastname: String,
    pub paid: bool,
}
```

Dokumenty z kolekcji users przetrzymywane są w następujących strukturach w naszym kodzie:

```
#[derive(Debug, Serialize, Deserialize, Clone)] //generate implementation support
for formatting the output, serializing, and deserializing the data structure.
pub struct User {
    #[serde(rename = "_id", skip_serializing_if = "Option::is_none")]
    pub id: Option<ObjectId>,
    pub login: String,
    pub password: String,
    pub role: Vec<String>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub firstname: Option<String>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub lastname: Option<String>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub company_name: Option<String>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub my_expeditions: Option<Vec<MyExpedition>>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub organized_expeditions: Option<Vec<MyExpedition>>,
    #[serde(skip_serializing_if = "Option::is_none")]

    pub contact: Option<String>
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct MyExpedition {
    pub exp_id: String,
    pub name: String,
    pub start_date: i64,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub reserved: Option<bool>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub paid: Option<bool>,
}
```

Funkcje korzystające z bazy danych:

```
    pub fn create_expedition(&self, new_expedition: Expedition) ->
Result<InsertOneResult, Error> {
        let new_doc = Expedition {
            id: None, //mongoDB will create unique id
            name: new_expedition.name,
            stops: new_expedition.stops,
            max_no_participants: new_expedition.max_no_participants,
            guide: new_expedition.guide,
            organizer: new_expedition.organizer,
            start_time: new_expedition.start_time,
            end_time: new_expedition.end_time,
            home_station: new_expedition.home_station,
            participants: new_expedition.participants,
            price: new_expedition.price,
        };

        let expedition = self
            .expedition_col
            .insert_one(new_doc, None)
            .ok()
            .expect("Error creating expedition");
        Ok(expedition)
    }
```

```

        pub fn get_expedition(&self, id: &String) -> Result<Expedition, Error> {
        let obj_id = ObjectId::parse_str(id).unwrap();
        let filter = doc! {"_id": obj_id};
        let expedition_detail = self
            .expedition_col
            .find_one(filter, None)
            .ok()
            .expect("Error getting expeditions's detail");
        Ok(expedition_detail.unwrap())
    }
```

```


    pub fn add_expedition_to_organizator(
        &self,
        user_id: &String,
        new_expedition: Expedition
    ) -> Result<UpdateResult, Error> {

        let expedition_id = match self.create_expedition(new_expedition.clone()){
            Ok(expedition) =>
expedition.inserted_id.as_object_id().unwrap().to_hex(),
            Err(err) => return Err(err),
        };

        let new_user_doc = doc! {
            "$push": {
                "organized_expeditions": {
                    "exp_id": expedition_id,
                    "name": &new_expedition.name,
                    "start_date": &new_expedition.start_time,
                }
            }
            };

            let user_id = ObjectId::parse_str(user_id).unwrap();
            let user_filter = doc!{"_id":user_id};

            let updated_user_doc = self
            .user_col
            .update_one(user_filter, new_user_doc, None)
            .ok()
            .expect("Error updating user");

        Ok(updated_user_doc)
    }
```

```rust
pub fn add_expedition_to_user(
        &self,
        user_id: &String,
        expedition_id: &String
    ) -> Result<UpdateResult, Error> {

        let expedition_detail = match self.get_expedition(expedition_id) {
            Ok(expedition) => expedition,
            Err(err) => return Err(err),
        };

        let user_detail = match self.get_user(user_id) {
            Ok(user) => user,
            Err(err) => return Err(err),
        };


        let new_user_doc = doc! {
            "$push": {
                "my_expeditions": {
                    "exp_id": expedition_id,
                    "name": &expedition_detail.name,
                    "start_date": &expedition_detail.start_time,
                    "reserved": false,
                    "paid": false
                }
            }
        };

        let new_exp_doc = doc! {
            "$push": {
                "participants":{
                    "user_id": user_id,
                    "firstname": user_detail.firstname,
                    "lastname": user_detail.lastname,
                    "paid": false
                }
            }
        };

        let expedition_id = ObjectId::parse_str(expedition_id).unwrap();
        let user_id = ObjectId::parse_str(user_id).unwrap();
        let user_filter = doc!{"_id":user_id};
        let expedition_filter = doc!{"_id":expedition_id};

        let updated_expedition_doc = self
        .expedition_col
        .update_one(expedition_filter, new_exp_doc, None)
        .ok()
        .expect("Error updating expedition");
        self
        .user_col
        .update_one(user_filter, new_user_doc, None)
        .ok()
        .expect("Error updating user");

        Ok(updated_expedition_doc)
    }
```

```
    pub fn mark_expedition_as_paid(
        &self,
        user_id: &String,
        expedition_id: &String
    ) -> Result<UpdateResult, Error> {

        let expedition_id = match ObjectId::parse_str(expedition_id) {
            Ok(id) => id,
            Err(err) => return Err(Error::from(err)),
        };

        let user_id = match ObjectId::parse_str(user_id) {
            Ok(id) => id,
            Err(err) => return Err(Error::from(err)),
        };

        let user_filter = doc! {
            "_id": user_id,
            "my_expeditions.exp_id": expedition_id,
        };

        let expedition_filter = doc! {
            "_id": expedition_id,
            "participants.user_id": user_id,
        };

        let update_user_doc = doc! {
            "$set": {
                "my_expeditions.$.paid": true,
            },
        };

        let update_expedition_doc = doc! {
            "$set": {
                "participants.$.paid": true,
            },
        };


        let updated_user_doc = self
        .user_col
        .update_one(user_filter, update_user_doc, None)
        .ok()
        .expect("Error updating user");

        self
        .expedition_col
        .update_one(expedition_filter, update_expedition_doc, None)
        .ok()
        .expect("Error updating expedition");


        Ok(updated_user_doc)
    }
```

```
pub fn make_user_participant(
        &self,
        id: &String,
        firstname: &String,
        lastname: &String
    ) -> Result<UpdateResult, Error> {

        let obj_id = ObjectId::parse_str(id).unwrap();
        let filter = doc! {"_id": obj_id};
        let new_doc = doc! {
            "$set":
                {
                    "firstname":firstname,
                    "lastname":lastname,
                    "my_expeditions":[],
                },
            "$push":
            {
                "role":"Participant"
            },
        };
        let updated_doc = self
        .user_col
        .update_one(filter, new_doc, None)
        .ok()
        .expect("Error updating user");

        Ok(updated_doc)
    }
```

```
#[post("/user/paid/<expedition_id>/<user_id>")]
pub fn mark_expedition_as_paid(db:
&State<MongoRepo>,expedition_id:String,user_id:String) ->
Result<Json<UpdateResult>, Status> {
    let result = db.mark_expedition_as_paid(&expedition_id, &user_id);
    match result{
        Ok(res) => Ok(Json(res)),
        Err(_) =>Err(Status::InternalServerError),
    }
}
```

```


    pub fn make_user_organizer(
        &self,
        id: &String,
        company_name: &String,
        contact: &String
    ) -> Result<UpdateResult, Error> {

        let obj_id = ObjectId::parse_str(id).unwrap();
        let filter = doc! {"_id": obj_id};
        let new_doc = doc! {
            "$set":
            {
                "company_name":company_name,
                "contact":contact,
                "organized_expeditions":[],
            },
            "$push":
            {
                "role":"Organizer"
            },
        };
        let updated_doc = self
        .user_col
        .update_one(filter, new_doc, None)
        .ok()
        .expect("Error updating user");

        Ok(updated_doc)
    }
```

```


    pub fn update_expedition(
        &self,
        id: &String,
        updated_expedition: Expedition
    ) -> Result<UpdateResult, Error> {

        let obj_id = ObjectId::parse_str(id).unwrap();
        let filter = doc! {"_id": obj_id};
        let new_doc = doc! {
            "$set":
                {
                    "name": updated_expedition.name,
                    "stops": updated_expedition.stops,
                    "max_no_participants": updated_expedition.max_no_participants,
                    "guide": {
                        "firstaname":updated_expedition.guide.firstname,
                        "lastname":updated_expedition.guide.lastname,
                        "age":updated_expedition.guide.age,
                        "experience":updated_expedition.guide.experience,},
                    "start_time": updated_expedition.start_time,
                    "end_time": updated_expedition.end_time,
                    "home_station": updated_expedition.home_station,
                    "price": updated_expedition.price,
                },
        };
        let updated_doc = self
            .expedition_col
            .update_one(filter, new_doc, None)
            .ok()
            .expect("Error updating expedition");

        Ok(updated_doc)
    }
```

```


    pub fn delete_expedition(&self, id: &String) -> Result<DeleteResult, Error> {
        let obj_id = ObjectId::parse_str(id).unwrap();
        let filter = doc! {"_id": obj_id};
        let expedition_detail = self
            .expedition_col
            .delete_one(filter, None)
            .ok()
            .expect("Error deleting expedition");
        Ok(expedition_detail)
    }
```

```


    pub fn get_all_expeditions(&self) -> Result<Vec<Expedition>, Error> {
        let cursors = self
            .expedition_col
            .find(None, None)
            .ok()
            .expect("Error getting list of expeditions");
        let expeditions = cursors.map(|doc| doc.unwrap()).collect();
        Ok(expeditions)
    }
```

```


    pub fn create_user(&self, new_user: User) -> Result<InsertOneResult, Error> {
        let new_doc = User {
            id: None, //mongoDB will create unique id
            login: new_user.login,
            password: new_user.password,
            role: new_user.role,
            firstname: new_user.firstname,
            lastname: new_user.lastname,
            company_name: new_user.company_name,
            my_expeditions: new_user.my_expeditions,
            organized_expeditions: new_user.organized_expeditions,
            contact: new_user.contact,
        };

        let user = self
            .user_col
            .insert_one(new_doc, None)
            .ok()
            .expect("Error creating user");
        Ok(user)
    }
```

```
    pub fn get_user(&self, id: &String) -> Result<User, Error> {
        let obj_id = ObjectId::parse_str(id).unwrap();
        let filter = doc! {"_id": obj_id};
        let user_detail = self
            .user_col
            .find_one(filter, None)
            .ok()
            .expect("Error getting user's detail");
        Ok(user_detail.unwrap())
    }
```

```
    pub fn delete_user(&self, id: &String) -> Result<DeleteResult, Error> {
        let obj_id = ObjectId::parse_str(id).unwrap();
        let filter = doc! {"_id": obj_id};
        let user_detail = self
            .user_col
            .delete_one(filter, None)
            .ok()
            .expect("Error deleting user");
        Ok(user_detail)
    }
```

```
    pub fn get_all_users(&self) -> Result<Vec<User>, Error> {
        let cursors = self
            .user_col
            .find(None, None)
            .ok()
            .expect("Error getting list of users");
        let user = cursors.map(|doc| doc.unwrap()).collect();
        Ok(user)
    }
```

```


    pub fn find_user(&self, login: &String) -> Result<User, Error> {
        let stored_user = self
            .user_col
            .find_one(doc! { "login": &login }, None)
            .ok()
            .expect("Error finding user");
        Ok(stored_user.unwrap())
    }
```

```


    pub fn get_contacts(&self, stop: &String) ->
Result<Vec<ContactOrganizator>,Error>{
        let contact_pipeline = vec![
        doc! {
            "$match": {
              "stops": stop
            }
          },
          doc! {
            "$lookup": {
              "from": "users",
              "let": { "orgId": { "$toObjectId": "$organizer.org_id" } },
              "pipeline": [
                {
                  "$match": {
                    "$expr": { "$eq": ["$_id", "$$orgId"] }
                  }
                }
              ],
              "as": "organizer_details"
            }
        },
          doc! {
            "$unwind": "$organizer_details"
          },
          doc! {
            "$project": {
              "_id": 0,
              "expedition_name": "$name",
              "organizer_name": "$organizer_details.login",
                "contact": "$organizer_details.contact"
              }
            }
];
let cursors = self
            .expedition_col
            .aggregate(contact_pipeline, None)
            .ok()
            .expect("Error getting list of contacts");
        let results: Result<Vec<ContactOrganizator>, Error> = cursors.map(|doc| {

            let expedition_name =
doc.clone().unwrap().get_str("expedition_name").unwrap_or("").to_string();
            let organizer_name =
doc.clone().unwrap().get_str("organizer_name").unwrap_or("").to_string();
            let contact =
doc.clone().unwrap().get_str("contact").unwrap_or("").to_string();

            Ok(ContactOrganizator {
                expedition_name,
                organizer_name,
                contact,
            })
        }).collect();

        results
    }
```

Podstawowe endopointy:

```
#[post("/expedition", data = "<new_expedition>")]
pub fn create_expedition(
    db: &State<MongoRepo>,
    new_expedition: Json<Expedition>,
) -> Result<Json<InsertOneResult>, Status> {

    let expedition: Expedition = new_expedition.into_inner(); //change from
Json<Expedition> to Expedition
    let organiser_id = expedition.organizer.clone().expect("Organiser not
provided!").org_id;
    let expedition_detail = db.create_expedition(expedition.clone());
    let result = match expedition_detail {
        Ok(expedition) => Ok(Json(expedition)),
        Err(_) => Err(Status::InternalServerError),
    };

    let result2 = match db.add_expedition_to_organizator(&organiser_id,
expedition) {
        Ok(_) => (),
        Err(_) => return Err(Status::InternalServerError)
    };
    result
}
```

```rust
#[get("/expedition/<path>")]
pub fn get_expedition(
    db: &State<MongoRepo>,
    path: String
) -> Result<Template, Status> {

    let id = path;
    if id.is_empty() {
        return Err(Status::BadRequest);
    };
    let expedition_detail = db.get_expedition(&id);
    match expedition_detail {
        Ok(expedition) => {
            let mut context = HashMap::new();
            context.insert("expedition", expedition);
            Ok(Template::render("expedition", &context))
        },
        Err(_) => Err(Status::InternalServerError),
    }
}
```

```rust
#[put("/expedition/<path>", data = "<new_expedition>")]
pub fn update_expedition(
    db: &State<MongoRepo>,
    path: String,
    new_expedition: Json<Expedition>,
) -> Result<Json<Expedition>, Status> {

    let id = path;
    if id.is_empty() {
        return Err(Status::BadRequest);
    };
    let data: Expedition = new_expedition.into_inner();
    let update_result = db.update_expedition(&id, data);
    match update_result {
        Ok(update) => {
            if update.matched_count == 1 {
                let updated_expedition_info = db.get_expedition(&id);
                return match updated_expedition_info {
                    Ok(expedition) => Ok(Json(expedition)),
                    Err(_) => Err(Status::InternalServerError),
                };
            } else {
                return Err(Status::NotFound);
            }
        }
        Err(_) => Err(Status::InternalServerError),
    }
}
```

```rust
#[delete("/expedition/<path>")]
pub fn delete_expedition(
    db: &State<MongoRepo>,
    path: String
) -> Result<Json<&str>, Status> {

    let id = path;
    if id.is_empty() {
        return Err(Status::BadRequest);
    };
    let result = db.delete_expedition(&id);
    match result {
        Ok(res) => {
            if res.deleted_count == 1 {
                return Ok(Json("Expedition successfully deleted!"));
            } else {
                return Err(Status::NotFound);
            }
        }
        Err(_) => Err(Status::InternalServerError),
    }
}
```

```rust
#[get("/expeditions")]
pub fn get_all_expeditions(db: &State<MongoRepo>) -> Result<Template, Status> {
    let maybe_expeditions = db.get_all_expeditions();
    match maybe_expeditions {
        Ok(expeditions) => {
            let api_expeditions: Vec<ApiExpedition> = expeditions.iter()
                .map(|exp| ApiExpedition {
                    str_id: exp.id.unwrap().to_hex(),
                    expedition: exp.clone()
                }).collect();
            let mut context = HashMap::new();
            context.insert("api_expeditions", api_expeditions);
            Ok(Template::render("expeditions", &context))
        },
        Err(_) => Err(Status::InternalServerError),
    }
}
```

```rust
#[post("/raports/contacts", data = "<stop_form>")]
pub fn get_contact_raport(
    db: &State<MongoRepo>,
    stop_form: Form<HashMap<String, String>>
) -> Result<Template, Status> {
    let stop = stop_form.get("stop").cloned().unwrap_or_default();
    match db.get_contacts(&stop) {
        Ok(contacts) => {
            let mut context = HashMap::new();
            context.insert("contacts", contacts);
            Ok(Template::render("contacts", &context))
        },
        Err(_) => Err(Status::InternalServerError)
    }
}
```

```


#[post("/user", data = "<new_user>")]
pub fn create_user(
    db: &State<MongoRepo>,
    new_user: Json<User>,
) -> Result<Json<InsertOneResult>, Status> {

    let user: User = new_user.into_inner(); //change from Json<User> to User
    let user_detail = db.create_user(user);
    match user_detail {
        Ok(user) => Ok(Json(user)),
        Err(_) => Err(Status::InternalServerError),
    }
}
```

```

#[get("/user/<path>")]
pub fn get_user(
    db: &State<MongoRepo>,
    path: String
) -> Result<Template, Status> {

    let id = path;
    if id.is_empty() {
        return Err(Status::BadRequest);
    };
    let user_detail = db.get_user(&id);
    match user_detail {
        Ok(user) => {
            let mut context = HashMap::new();
            context.insert("user", user);
            Ok(Template::render("user", &context))
        },
        Err(_) => Err(Status::InternalServerError),
    }
}
```

```rust
#[delete("/user/<path>")]
pub fn delete_user(
    db: &State<MongoRepo>,
    path: String
) -> Result<Json<&str>, Status> {

    let id = path;
    if id.is_empty() {
        return Err(Status::BadRequest);
    };
    let result = db.delete_user(&id);
    match result {
        Ok(res) => {
            if res.deleted_count == 1 {
                return Ok(Json("Expedition successfully deleted!"));
            } else {
                return Err(Status::NotFound);
            }
        }
        Err(_) => Err(Status::InternalServerError),
    }
}
```

```rust
#[get("/users")]
pub fn get_all_users(db: &State<MongoRepo>) -> Result<Template, Status> {
    let maybe_users = db.get_all_users();
    match maybe_users {
        Ok(users) => {
            let api_users: Vec<ApiUser> = users.iter()
                .map(|usr| ApiUser {
                    str_id: usr.id.unwrap().to_hex(),
                    user: usr.clone()
                }).collect();
            let mut context = HashMap::new();
            context.insert("api_users", api_users);
            Ok(Template::render("users", &context))
        },
        Err(_) => Err(Status::InternalServerError),
    }
}
```