C++ Rest Server

Nebojša Koturović

15. maj 2020.

Sadržaj

1 C++ REST Server

- Ovaj dokument ima za cilj objašnjenje nekih delova projekta.

1.1 Biblioteke korišćene u projektu:

- **fmt-lib** (Uslov za restinio): https://github.com/fmtlib/fmt
- **restinio** (HTTP Server + Express router): https://github.com/Stiffstream/restinio
- **refl-cpp** (Statička refleksija): https://github.com/veselink1/refl-cpp
- **BoostHana** (metaprogramiranje): https://www.boost.org/doc/libs/1_73_0/libs/hana/doc/html/index.html
- **nlohmann::json** (json): https://github.com/nlohmann/json
- **SOCI** (DBAccessLib for SQL - ovde za sqlite): https://github.com/SOCI/soci

2 Model (Predstavljanje podataka)

2.1 Implementacioni detalji

- **VAŽNO:** Svi fajlovi vezani za ovo pitanje se nalaze u folderu **`/include/model`**
- U ovom folderu su 4 header fajla:
  1. **field.hpp** — Field klasa
  2. **constraint.hpp** — Ograničenja (paremetri `Field` klase).
  3. **model.hpp** — Operatori koji se oslanjaju na refleksiju (`to_json`, `from_json`, `to_map`, `operator<<`, ...)
  4. **models.hpp** — Konkretni primeri modela (npr. User) i definicije potrebne za omogućivanje refleksije.

2.2 Koncept i klasa Model (model.hpp)

Bazna klasa `Model` i koncept `CModel` nemaju neku posebnu ulogu. Da ne pišemo `<typename T>` nego `<CModel M>`.

Program 2.1: Dummy klasa Model i dummy CModel koncept

```
struct Model {}; // definicija dummy klase Model
// Dummy koncept CModel
template<typename C>
concept CModel = std::derived_from<C,Model>;
```

Program 2.2: Primer modela User (models.hpp)

```
struct User : Model {
    Field<int,cnstr::Unique> id;
    Field<std::string, cnstr::Unique, cnstr::Length<1,10>, cnstr::Required> username;
    Field<std::string, cnstr::Required, cnstr::Length<6,255>> password;
    Field<std::string, cnstr::Unique,cnstr::Required, cnstr::NotEmpty, cnstr::Length<2,32>> email;
    Field<std::string, cnstr::Required, cnstr::Length<2,64>> firstname;
    Field<std::string, cnstr::Required, cnstr::Length<2,64>> lastname;
    Field<int, cnstr::Required> born; // linux time (since epoch)
    Field<std::string> status;
};
```

2.3 Koncept ograničenja (constraint.hpp)

- Ograničenja su paremetri klase `Field`.
- Ograničenja su **compile time stvar!!!** Veza sa runtime-om su statičke funkcije klase!
- Definicije ograničenja su u `"constraint.hpp"` i namespace im je `cnstr::`.

Program 2.3: Constraint concept

```
/* Compile type concept (trait) for what is Constraint */
template<typename C>
concept Cnstr = requires(typename C::inner_type t) {
    { C::is_satisfied(t) } -> std::same_as<bool>;
    { C::name() } -> std::same_as<const char *>;
    { C::description_en() } -> std::same_as<std::string>;
    { C::description_rs() } -> std::same_as<std::string>;
};
```

Program 2.4: Primer klase koja zadovoljava koncept Cnstr

```
struct NotEmpty {
    using inner_type = std::string_view;
    NotEmpty() = delete; // ne moze se instancirati

    constexpr static bool is_satisfied(std::string_view s) { return !s.empty(); }
    constexpr static const char * name() { return "NotEmpty"; }

    static std::string description_en() {
        return "Field must not be empty";
    }
    static std::string description_rs() {
        return "Polje ne sme biti prazno";
    }

};
```

2.4 Klasa Field (field.hpp)

Program 2.5: Pojednostavljena verzija Field klase

```
template <typename T, /*typename*/ cnstr::Cnstr ...Cs>
struct Field {
    using cs = type_list<Cs...>; // Type lista Constraint-ova

    std::optional<T> value; // Ovde se cuva vrednost

    // Primenjuje funkciju f na sve Cstr-ove koji nisu ispunjeni
    template <class Func, class ... FArgs>
    auto apply_to_unsatisfied_cnstrs(Func && f = Func{}, FArgs&& ... fargs) const;
};
```

- Od posebne važnosti je funkcija `apply_to_unsatisfied_cnstrs(func, fargs)` koja prihvata funkciju `func(fargs)` i primenjuje je na sva neispunjena ograničenja i vraća vektor rezultata.
- Ova funkcija radi na Field-ovima, a želimo i funkciju koja radi na celom Modelu.

2.4.1 Refleksija (model.hpp)

Sada ćemo već pomenutu funkciju `apply_to_unsatisfied_cnstrs` **liftovati**. Do sada je mogla da radi samo sa klasom **Field**, a sada ćemo omogućiti i da radi sa **Model-om** koji je ustvari **proizvod** više **Field-ova**

$$\mathcal{M} = \mathcal{F} \times \mathcal{F} \times ... \times \mathcal{F}$$

To ćemo učiniti pomoću `refl-cpp` biblioteke, koja nam pruža mogućnost iteracije po članskim promenljivama i funkcijama. Za više informacija o refleksiji pogledati `refl-cpp` biblioteku.

Program 2.6: Funkcija koja primenjuje func na nezadovoljena ograničenja svih članskih promenljiva Modela

```
template <CModel M, typename Func, typename ... Args>
auto apply_to_unsatisfied_cnstrs_of_model(const M& model, Func &&f, Args&& ...args)
{
    // Mapa: FieldName -> Vec[f<Unsatisfied Constraint>(args...)]
    std::map<std::string, std::vector<decltype(f.template operator()<cnstr::Required>(args...))>> results_map;
    // Koriscenjem refleksije mozemo da iteriramo kroz clanske promenljive Model-a
    refl::util::for_each(refl::reflect(model).members, [&](auto member) {
        if constexpr (refl::trait::is_field<decltype(member)>())
            if (auto && vec = member(model).apply_to_unsatisfied_cnstrs(f, args...); vec.size())
                results_map[member.name.str()] = std::move(vec);
   });
    return results_map; // rezultat se naknadno moze pretvoriti u json
}
```

Evo primera gde ova funkcija izvlači opise nezadovoljenih Constraint-ova za svako polje unutar modela.

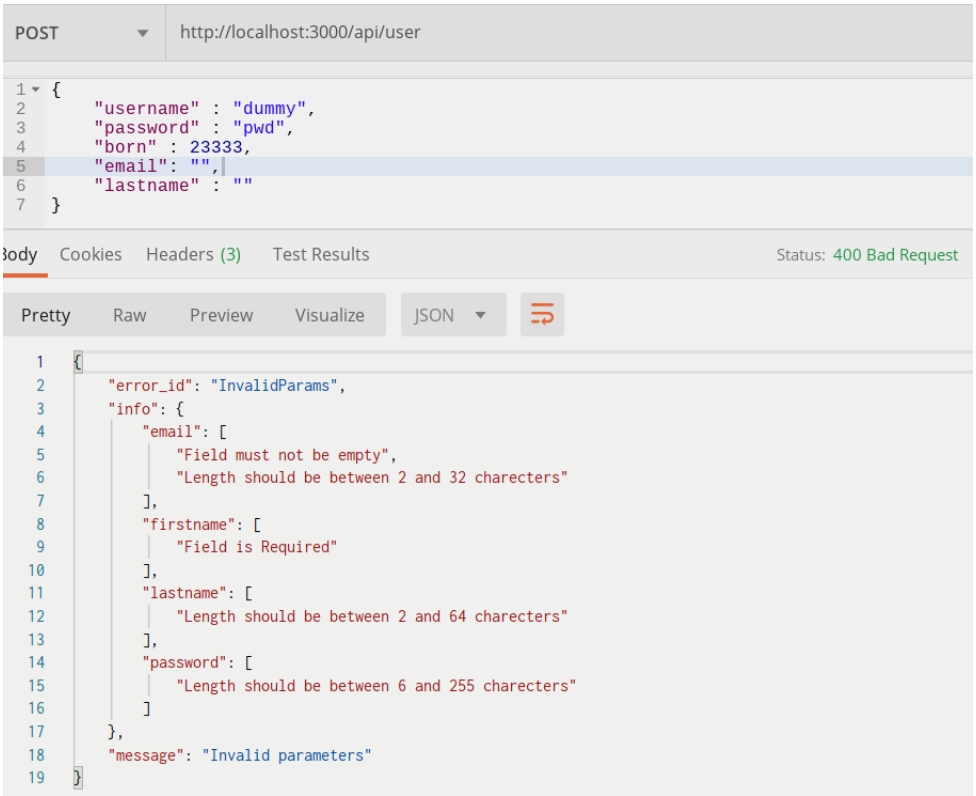
Program 2.7: Izvlačenje opisa nezadovoljenih uslova iz modela

```
/* Lambda Funkcija pomocu koje izvlacimo opis Ogranicenja C */
auto description = []<Cnstr C>(std::string_view lang = "en") -> std::string {
    if (lang == "rs") {
        return C::description_rs();
    } else /* if lang en */ {
        return C::description_en();
    }
};

/* Jedna instanca modela User */
rs::model::User kotur {
    .username = { "kotur" },
    .password = { "pwd" },
    .email = { "" },
    .firstname = { "Nebojsa" },
    .lastname = { "Koturovic" },
    .born = { 813336633 }, /* Linux Time */
};

/* Primena Lambde na sva neispenjena ogranicenja */
if (auto ds_map = rs::model::apply_to_unsatisfied_cnstrs_of_model(kotur, description, "rs"); ds_map.size())
    std::cout << rs::json_t(ds_map).dump(2) << '\n'; // ispis mape opisa u json_formatu
```

Izlaz programa 2.7:

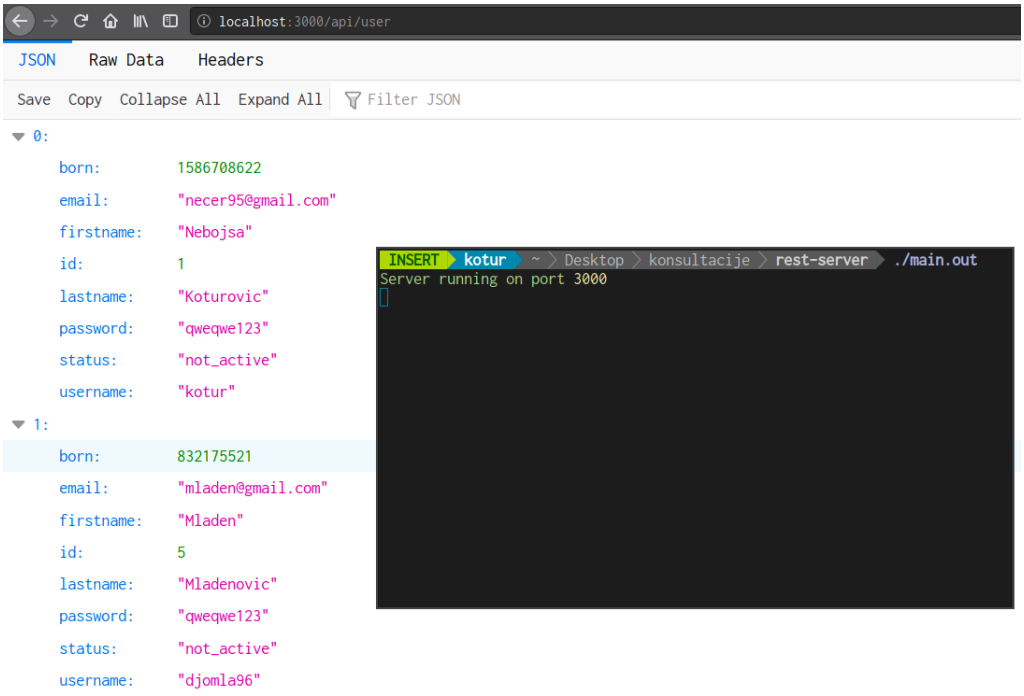
```
{
  "email": [
    "Polje ne sme biti prazno",
    "Duzina mora da bude izmedju 2 i 32 karaktera"
  ],
  "password": [
    "Duzina mora da bude izmedju 6 i 255 karaktera"
  ]
}
```

A ovako neuspešan **POST** zahtev izgleda u realnom primeru (verzija sa `lang="en"`):



Slika 2.1: Neuspešan POST zahtev

Evo i uspešnog **GET** zahteva:



Slika 2.2: Uspešan GET zahtev

2.5 Dodatna zapažanja

- Čini mi se da statička tipiziranost C++-a i `meta programiranje` daju finu potporu.
- Uz pomoć **refl-cpp** su implementirane i funkcionalnosti `from_json`, `to_json`, `from_sql_to_model`, `from_model_to_sql`. Dakle ovde je cilj eliminisati **boiler-plate** koji bi se uporno ponavljao.
- Ideju je potencijalno moguće dalje razvijati, gde će automatski moći da se generiše baza podataka od postojećih modela i slično.
- Primetio sam npr. u Djangu ima slična ideja sa ograničenjima, al naravno sve je runtime.
- U folderu `examples` su data neka dva sitna primera vezana za `json` i `sql`.

2.6 Pitanja

- Da li ima smisla uopšte nešto ovako? Koliko je praktično sve ovo, da li je suludo kucati u C++-u REST server?
- Sugestije da li i na koji način bi bilo logičnije pristupiti rešavanju prezentovanih problema.

compile-time Funkcije/Lambde/Funkcijski-objekti ...: Nisam pronašao najbolji način da šaljem kao argument članske funkcije nespecijalizovane generičke klase.

Sve vrste zamerki i sugestija su i više nego dobrodošle!!