Crate schemars

Schemars

build passing | crates.io v1.0.0-alpha.17 | docs passing | schemars rustc 1.60+

Generate JSON Schema documents from Rust code

Basic Usage

If you don't really care about the specifics, the easiest way to generate a JSON schema for your types is to `#[derive(JsonSchema)]` and use the `schema_for!` macro. All fields of the type must also implement `JsonSchema` - Schemars implements this for many standard library types.

```rust
use schemars::{schema_for, JsonSchema};

#[derive(JsonSchema)]
pub struct MyStruct {
    pub my_int: i32,
    pub my_bool: bool,
    pub my_nullable_enum: Option<MyEnum>,
}

#[derive(JsonSchema)]
pub enum MyEnum {
    StringNewType(String),
    StructVariant { floats: Vec<f32> },
}
```

Rust ▼

```rust
println!("{}", serde_json::to_string_pretty(&schema).unwrap());
```

▶ Click to see the output JSON schema...

Serde Compatibility

One of the main aims of this library is compatibility with Serde. Any generated schema *should* match how serde_json would serialize/deserialize to/from JSON. To support this, Schemars will check for any `#[serde(...)]` attributes on types that derive `JsonSchema`, and adjust the generated schema accordingly.

```
use schemars::{schema_for, JsonSchema};
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize, JsonSchema)]
#[serde(rename_all = "camelCase", deny_unknown_fields)]
pub struct MyStruct {
    #[serde(rename = "myNumber")]
    pub my_int: i32,
    pub my_bool: bool,
    #[serde(default)]
    pub my_nullable_enum: Option<MyEnum>,
}

#[derive(Deserialize, Serialize, JsonSchema)]
#[serde(untagged)]
pub enum MyEnum {
    StringNewType(String),
    StructVariant { floats: Vec<f32> },
}

let schema = schema_for!(MyStruct);
println!("{}", serde_json::to_string_pretty(&schema).unwrap());
```

▶ Click to see the output JSON schema...

`#[serde(...)]` attributes can be overriden using `#[schemars(...)]` attributes, which behave identically (e.g. `#[schemars(rename_all = "camelCase")]`). You may find this useful if you want to change the generated schema without affecting Serde's behaviour, or if you're just not using Serde.

Schema from Example Value

If you want a schema for a type that can't/doesn't implement `JsonSchema`, but does implement `serde::Serialize`, then you can generate a JSON schema from a value of that type. However, this schema will generally be less precise than if the type implemented `JsonSchema` - particularly when it involves enums, since schemars will not make any assumptions about the structure of an enum based on a single variant.

```
use schemars::schema_for_value;
use serde::Serialize;

#[derive(Serialize)]
pub struct MyStruct {
    pub my_int: i32,
    pub my_bool: bool,
    pub my_nullable_enum: Option<MyEnum>,
```

```rust
}

#[derive(Serialize)]
pub enum MyEnum {
    StringNewType(String),
    StructVariant { floats: Vec<f32> },
}

let schema = schema_for_value!(MyStruct {
    my_int: 123,
    my_bool: true,
    my_nullable_enum: Some(MyEnum::StringNewType("foo".to_string()))
});
println!("{}", serde_json::to_string_pretty(&schema).unwrap());
```

▶ Click to see the output JSON schema...

Feature Flags

- `derive` (enabled by default) - provides `#[derive(JsonSchema)]` macro
- `impl_json_schema` - implements `JsonSchema` for Schemars types themselves
- `preserve_order` - keep the order of struct fields in `Schema` and `SchemaObject`
- `raw_value` - implements `JsonSchema` for `serde_json::value::RawValue` (enables the serde_json `raw_value` feature)

Schemars can implement `JsonSchema` on types from several popular crates, enabled via feature flags (dependency versions are shown in brackets):

- `chrono` - chrono (^0.4)
- `indexmap1` - indexmap (^1.2)
- `indexmap2` - indexmap (^2.0)
- `either` - either (^1.3)
- `uuid08` - uuid (^0.8)
- `uuid1` - uuid (^1.0)
- `smallvec` - smallvec (^1.0)
- `arrayvec05` - arrayvec (^0.5)
- `arrayvec07` - arrayvec (^0.7)
- `url` - url (^2.0)
- `bytes` - bytes (^1.0)
- `enumset` - enumset (^1.0)

- `rust_decimal` - rust_decimal (^1.0)
- `bigdecimal03` - bigdecimal (^0.3)
- `bigdecimal04` - bigdecimal (^0.4)
- `smol_str` - smol_str (^0.1.17)
- `semver` - semver (^1.0.9)

For example, to implement `JsonSchema` on types from `chrono`, enable it as a feature in the `schemars` dependency in your `Cargo.toml` like so:

```
[dependencies]
schemars = { version = "0.8", features = ["chrono"] }
```

Modules

gen	JSON Schema generator and settings.
schema	JSON Schema types.
visit	Contains the `Visitor` trait, used to recursively modify a constructed schema and its subschemas.

Macros

schema_for	Generates a `RootSchema` for the given type using default settings.
schema_for_value	Generates a `RootSchema` for the given example value using default settings.

Traits

JsonSchema	A type which can be described as a JSON Schema document.

Type Aliases

Map	
MapEntry	
Set	The set type used by schemars types.

Derive Macros

JsonSchema

JsonSchema_repr