**How to easily implement a configuration-first provider pattern in Rust 🦀**

[[Dotan Nahum](https://jondot.medium.com/?source=post_page-----fe005aa06d02--------------------------------)](https://jondot.medium.com/?source=post_page-----fe005aa06d02--------------------------------)

[Dotan Nahum](https://jondot.medium.com/?source=post_page-----fe005aa06d02--------------------------------)

·

Follow

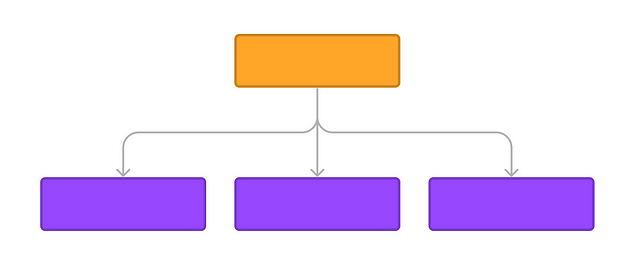
7 min read

·

Oct 17

33

1



In this post we’ll review a way to **build a provider pattern in Rust** which is dynamic, extensible, and fun to maintain, while as statically typed as possible.

**You might need a provider pattern when building a configuration for a system that has plugins**, or different swappable components, logic encoded with no-code approaches, workflow configuration (think: CI/CD YAMLs), and more.

Here’s an example YAML of two “blocks” for some kind of a workflow runner. **It lets you configure which blocks run one after another and their settings**.

Below, we see an env\_block setting environment variables and a shell\_block running a shell command. Each block is configured via the same YAML, and, once loaded, has a single run function.

- kind: env\_block  
 id: env1  
 keys:  
 PATH: foo/bar  
- kind: shell\_block  
 id: sh1   
 cmd: "echo $PATH"

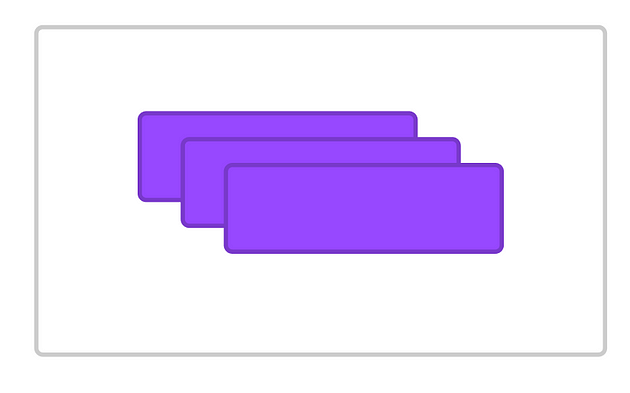
**We’ll implement dynamic block creation from configuration**, or as some would say *plugin system*, or *registry of blocks*, or *strategy design pattern*, or *provider pattern*.

Ultimately we want something like this:

let blocks\_config = load\_yaml(); // box dyn for fexibility  
// do something to turn config into Vec<Box<dyn Block>>  
blocks = figure\_it\_out(blocks\_config);  
blocks.for\_each(|b| b.run());

This code is completely ignorant of the different types of blocks we have, or their different settings.

**The “registry”**



**Remember: we have a collection of blocks**. Each block, has a different concrete implementation, and each block can have its own configuration:

* env\_block sets a bunch of environment variables
* shell\_block runs a shell command
* any other block..

Each block has a run function, available through a Block trait.

trait Block {  
 fn run() -> Result<()>;  
}

Normally we would need **something** that knows about all different kinds of blocks, so maybe something like a blocks registry, registry.rs:

// load YAML  
fn build\_blocks(blocks\_config: ..) -> .. {  
 for block in blocks\_config.blocks {  
 match block.kind {  
 .. do something based on the `kind` field ..  
 .. build a block ..  
 .. push it into a vec ..  
 }  
 }  
}

Some issues with this:

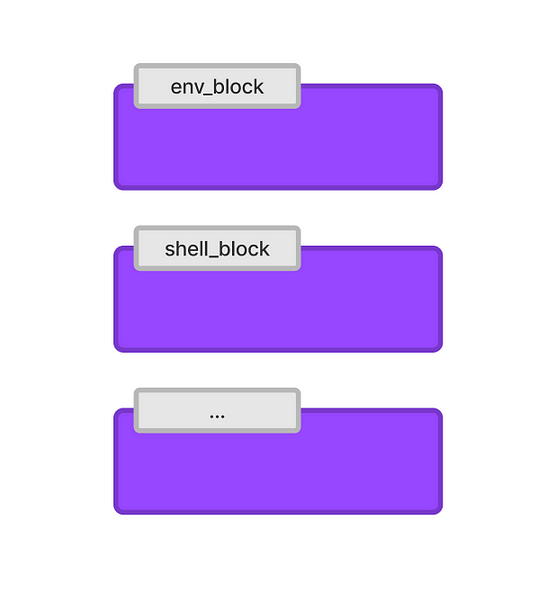
* Late binding: we only know what the rest of the configuration fields mean **after** we deal with the .kind attribute, so we'll have to keep it as serde\_json::Value
* Or, we’ll have to think of a different kind of configuration format which will deserialize into a strongly typed model, such as:

env\_block:  
 ..  
shell\_block:  
 ..

Keep in mind that the above solution loses ordering between env\_block and shell\_block. There are additional solutions, but they revolve around the same idea: encoding types in **configuration structure** so that deserialization has enough power to infer those types.

**What ever solution we’ll find, it’s not as easy to maintain** — when we need to add new kinds of blocks, when some blocks are enabled only by *features*, or when we have complex block creation logic.

**Serde can deserialize traits with**typetag



What if we can **deserialize each block options** into *some****behavior****that can build a block from the options*?

The key here is [typetag](https://github.com/dtolnay/typetag). typetag is a great library that you can use to have serde deserialize trait objects using a single discriminating field for indicating a type, ultimately "hidden" by a trait.

**And once we can deserialize trait objects, we can dial into that dynamic behavior that we wanted.**

Below, our type tag in the YAML is kind. This is all the information that's needed to decide which implementation of block the next sibling fields in the YAML relate to.

- kind: env\_block  
 id: env1  
 keys:  
 PATH: foo/bar  
- kind: shell\_block  
 id: sh1   
 cmd: "echo $PATH"

**Now, we define a BlockBuilder trait**. We say that a BlockBuilder's job is to take a particular block's YAML options and configure a block for us based on those options, remember, it returns a Box<dyn Block> because it needs to build *any kind of block* for us, and behind the scenes it will also magically pick the concrete type for us.

#[typetag::serde(tag = "kind")]  
trait BlockBuilder {  
 fn build() -> Result<Box<dyn Block>>  
}

**The important bit is the tag = "kind" attribute**, which tells typetag to generate the right code to build a specific impl of BlockBuilder that relates to our specific block.

**Why a seperate abstraction for building our block?** because we recognize that a “live” block, may contain fields that are initialized but not serializable, for example a live database connection, while a block configuration section may contain a connection **string**. This way, as a best-practice, we separate the behavior of **constructing** blocks from configuration from the actual block. We’ll show an example later where we can combine the two.

Here’s the data model for our env block options on the Rust side, as defined in the YAML file:

#[derive(Default, Debug, Serialize, Deserialize, Clone)]  
pub struct EnvBlockOpts {  
 id: String,  
 keys: HashMap<String, String>,  
}

We see that it’s an env block, which is one kind of block our of all possible blocks we have.

We implement a BlockBuilder, and **note the specific values we set for the typetag::serde attribute: env\_block**.

#[typetag::serde(name = "env\_block")]  
impl BlockBuilder for EnvBlockOpts {  
 fn build(&self) -> Result<Box<dyn Block>> {  
 Ok(Box::new(EnvBlock::new(self.id.clone(), self.keys.clone())))  
 }  
}  
// The actual env block (which isn't so important for the purpose of this discussion)  
pub struct EnvBlock {  
 id: String,  
 keys: HashMap<String, String>,  
}  
impl EnvBlock {  
 pub fn new(id: String, keys: HashMap<String, String>) -> Self {  
 Self { id, keys }  
 }  
}

Now we can load all block configurators:

// `load\_yaml` just reads and deserializes yaml, no special code  
let block\_configs = load\_yaml(); // Vec<Box<dyn BlockBuilder>>  
// generically call builders to get all needed blocks.   
// we don't know or care about specific block implementations,  
// we're getting a Box<dyn Block> which has a `.run` function and that's perfect for us.  
let blocks = block\_configs.iter().for\_each(|block\_config| block\_config.build()).collect::<Vec<\_>>();

Note that there nothing in the code above indicating a specific kind of block or a specific implementation of a block. **Everything is fully dynamic** and extensible, and our code is ignorant of how many types of blocks there are, how to build them, and so on.

To illustate what’s happening when building an env\_block:

YAML  
-> we deserialize YAML  
 -> uses `kind=env\_block` to resolve EnvBlockOpts  
 -> deserialized EnvBlockOpts into a <dyn BlockBuilder> trait which exposes a `build` function  
 -> we call `.build`   
 -> get a fully configured <dyn Block>

**Nested blocks and nested configuration**

**What if we have a block that contains a block, and nests its YAML configuration as well?**

# a top level ping block  
- kind: ping\_block  
 # billing block  
 billing:  
 addr: ..  
 # auth block  
 auth:  
 addr: ..

Where a ping\_block's .run calls both children blocks's .run in order to accomplish what it needs to do.

The full implementation of dynamically loading ping\_block would be:

#[typetag::serde(tag = "kind")]  
trait BlockBuilder {  
 fn build() -> Result<Box<dyn Block>>  
}  
// note: billing and auth config builders will be automatically resolved by typetag but  
// `.build` will not be automatically called for the nested blocks  
#[derive(Serialize, Deserialize)]  
struct PingBlockOpts {  
 billing: Box<dyn BlockBuilder>,  
 auth: Box<dyn BlockBuilder>  
}  
#[derive(Serialize, Deserialize)]  
struct BillingOpts {  
 addr: String  
}  
#[derive(Serialize, Deserialize)]  
struct AuthOpts {  
 auth: String  
}  
// this is our actual ping block, which is composed out of two other blocks  
struct PingBlock {  
 billing\_ping: Box<dyn Block>,  
 auth\_ping: Box<dyn Block>  
}  
struct BillingBlock {  
 ..  
}  
struct AuthBlock {  
 ..  
}  
  
// individual block builders, with their appropriate opts:  
#[typetag::serde(name = "auth\_block")]  
impl BlockBuilder for AuthOpts {  
 fn build(&self) -> Result<Box<dyn Block>> {  
 ..  
 }  
}  
#[typetag::serde(name = "billing\_block")]  
impl BlockBuilder for BillingOpts {  
 fn build(&self) -> Result<Box<dyn Block>> {  
 }  
 ..  
}  
// The key is implementing a nested `build` in the top level block builder,  
// which calls individual nested block builders manually  
#[typetag::serde(name = "ping\_block")]  
impl BlockBuilder for PingBlockOpts {  
 fn build(&self) -> Result<Box<dyn Block>> {  
 // let its inner blocks config builders build the actual blocks  
 // and save them as fields via constructor  
 Ok(Box::new(PingBlock::new(self.billing.build(), self.auth.build())))  
 }  
}

**Self-building blocks**

**Sometimes a block can have serializable fields** and its construction is very simple.

In that case we can save a bit of typing and have a bloc implement **both** BlockBuilder and the Block trait so it can build itself with no need for a seperate "opts" struct.

// the block struct itself is serializable  
#[derive(Serialize, Deserialize)]  
struct HealthBlock {   
 addr: String  
}  
// note we implement a config builder for HealthBlock itself, there's no separate configuration opts struct  
#[typetag::serde(name = "health\_block")]  
impl BlockBuilder for HealthBlock {  
 fn build(&self) -> Result<Box<dyn Block>> {  
 Ok(Box::new(HealthBlock::new(..)))  
 }  
}  
impl Block for HealthBlock {  
 ..  
}  
// and now:  
let builder = load\_yaml(..block yaml..); // this is the config builder trait  
let block = builder.build(); // which builds itself

This is a compact way to implement this pattern, and you can start with it by default.