# Chapter 29. Configuration file

In this chapter we'll define a configuration file for the Today program, using the TOML file format. This allows us to control the workings of the program without changing the source code and recompiling. In the process we will also learn about trait objects.

## Using a TOML file to store the configuration

Since the Today program has expanded a lot, it is a good idea to provide a way for users (including ourselves) to configure the operation of the program. To some extent this could be done with command-line arguments, but we will reserve those to give options specific to one particular run (in Chapter 33, *Handling command-line arguments*), and use a *configuration file* for more permanent options, such as the event providers to use.

We can take a page from Cargo's playbook and use TOML as the format of our configuration file. We'll call the file `today.toml` and put it in the configuration directory of the program. This directory is operating-system specific, and can be found using the `dirs` as discussed in Chapter 28, *Configuration directories*.

There is an established crate in Rust for parsing TOML files, simply called `toml` and available on crates.io. The contents of the TOML-format configuration file for the Today program will initially look something like this:

**Example 29.1. The configuration file in TOML format**

```
[[providers]]
name = "history"
kind = "csv"
resource = "history.csv"

[[providers]]
name = "programming"
kind = "text"
resource = "programming.txt"

[[providers]]
name = "computing"
```

```
kind = "text"
resource = "computing.txt"
```

The `[[providers]]` sections list all the event providers that the program should use. The `kind` key specifies the kind of event provider is being configured. The program's documentation should list the allowable values.

The `resource` key is used to locate a file that is needed by the provider, or to supply a web address. Values that do not have a protocol prefix like `http://` or `https://` should be taken as local files relative to the configuration directory, whereas values that do have a protocol are treated as network resources. The interpretation of the resource string will depend completely on the event provider.

The configuration file can have as many providers as possible, but their names should be unique. If a duplicate provider name is found, the program must stop parsing the configuration file, report an error and quit. In Chapter 28, *Configuration directories* we had both a text file and a CSV file event provider with the name `compsci`, but that will not be allowed anymore.

## Parsing the TOML file

For TOML file parsing we need the `toml` crate, along with the `serde` crate. Serde is a well-established parsing library for parsing the JSON, YAML, and TOML formats, among others. It also has features to define attributes that help in serializing from and deserializing to Rust data structures. The clue is in the name; *ser* stands for serialization (writing out), and *de* stands for deserialization (reading in).

The `toml` crate can be added to the `Cargo.toml` in the usual fashion. The `serde` crate, however, needs some additional configuration to bring in the `derive` attributes:

**Example 29.2. Project file with `toml` and `serde` crates added**

```
[dependencies]
serde = { version = "1.0.228", features = ["derive"] }
toml = "1.1.2"
```

This tells Cargo to pull in the `derive` feature of the `serde` crate.

Next we will define some data types to hold the configuration, and put them in `src/lib.rs`:

**Example 29.3. Configuration data structures**

```
use serde::Deserialize;

#[derive(Deserialize, Debug)]
pub struct ProviderConfig {
    name: String,
    kind: String,
    resource: String,
}

#[derive(Deserialize, Debug)]
pub struct Config {
    providers: Vec::<ProviderConfig>,
}
```

These correspond to the format of our TOML file. Here we derive the deserialization support of the `serde` crate, along with the `Debug` trait.

We'll change the signature of the `run` function to take a parameter of type `Config`:

**Example 29.4. Updated `run` function signature**

```
pub fn run(config: &Config, config_path: &Path) -> Result<(), Box<dyn Error>> {
    // etc.
```

Later we will add code to act on the configuration, but first we need to take a trip back to the `main` function to actually parse it.

In Chapter 28, *Configuration directories* we developed a function called `get_config_path` which gets us the path for the configuration file as `Some(path)`, or `None` if it is not found. We will treat the configuration file as optional, so if neither the configuration path nor the TOML file exists, we just do nothing. The program could have internal event providers that can't be configured in the TOML file, so if the configuration file is found, we will just add its contents to the list of event providers.

First, we need to bring the `Config` struct into scope, so that we can parse the TOML file into an instance of it. Then we'll act upon any positive result from the `get_config_path` function:

**Example 29.5. Parsing the TOML configuration file in `main.rs`**

```
use std::path::{Path, PathBuf};
use std::fs;
use today::{run, Config};

fn main() {
    const APP_NAME: &str = "today";
    let config_path = get_config_path(APP_NAME);
    match config_path {
        Some(path) => {
            let toml_path = path.join(format!("{}.toml", APP_NAME));
            println!("Looking for configuration file '{}'", &toml_path.display());
            let config_str = fs::read_to_string(&toml_path)
                .expect("configuration file should exist");
            let config: Config = toml::from_str(&config_str)
                .expect("configuration file should be valid");
            println!("config: {:#?}", config);

            if let Err(e) = today::run(&config, &path) {
                eprintln!("Error running program: {}", e);
            }
        },
        None => {
            eprintln!("Unable to configure the application");
            return;
        }
    }
```

```
}
```

There are some debug printouts that will not be included in the final program, but at this stage they can be useful so that we know what is going on. If we run the program with the configuration file found earlier in this chapter, placed in the correct location according to our chosen operating system, we should see the following result in the terminal (this example is from macOS):

```
Looking for configuration file '/Users/me/Library/Application Support/today/today.toml'
config: Config {
    providers: [
        ProviderConfig {
            name: "history",
            kind: "csv",
            resource: "history.csv",
        },
        ProviderConfig {
            name: "programming",
            kind: "text",
            resource: "programming.txt",
        },
        ProviderConfig {
            name: "computing",
            kind: "text",
            resource: "computing.txt",
        },
    ],
}
```

Based on this example, the configuration file has been parsed successfully from TOML, and all the event providers mentioned have been registered. Then it's mostly a matter of actually instantiating them.

## Instantiating the event providers

After the information about event providers has been read from the configuration file, the providers need to be instantiated. We'll use the `kind` key in a

`[[providers]]` entry found in the TOML file to determine what kind of provider to instantiate. If we specify a provider kind that is not recognized by the implementation, it should result in at least a warning, if not an error.

In order to keep the `main` function as short as possible, we will delegate the provider instantiation to the `run` function in `src/lib.rs`. That requires us to pass the parsed configuration to the `run` function, but we have already changed the signature of the function in anticipation of that.

The `run` function can in turn delegate the provider instantiation to a new function called `create_providers`, which returns a vector of providers. However, all the providers we have created so far are structs of different types, and a vector can only hold elements that are all the same type. Initially this seems like a problem, but the event providers have one thing in common: they all implement the `EventProvider` trait.

In Rust we can use *trait objects* to take advantage of *polymorphism*, one of the most important concepts of object-oriented programming. While Rust is not an object-oriented programming language, it does have many of the concepts of object-oriented programming, if not all the terminology.

When we implement our new `create_providers` function, we make it return a vector whose elements are of type `Box<dyn EventProvider>`. This allows us to put structs that implement the `EventProvider` trait into the vector, even though they are of different types. This is a new concept, so let's unpack it.

First of all, a `Box<T>` in Rust is a generic type that gives us a constant-size pointer to some value, much like `std::unique_ptr` in C++. We can instantiate our various event providers and then put each of them in a box of type `dyn EventProvider`. This then lets us put the *boxes* in a vector with an element type of `Box<dyn EventProvider>`. Just to be clear, the type of the vector is `Vec<Box<dyn EventProvider>>`, so there are actually nested generic types here.

The `dyn` keyword is needed to tell the compiler that any calls to methods on the associated trait (in this case the `EventProvider` trait) are *dynamically dispatched*. Because each of the event providers has its own implementation of the `get_events`, doing wildly different things, it needs to be decided somehow which method should be called at any given time. Dynamic dispatch means figuring this out at runtime.

This is how the instantiation of the event providers is done in the `create_providers` function:

**Example 29.6. Instantiating the event providers in `src/lib.rs`**

```
fn create_providers(config: &Config, config_path: &Path) -> Vec::<Box<dyn EventProvider>> {
    // Try to create all the event providers specified in `config`.
    // Put them in a vector of trait objects.
    let mut providers: Vec::<Box<dyn EventProvider>> = Vec::new();
    for cfg in config.providers.iter() {
        let path = config_path.join(&cfg.resource);
        match cfg.kind.as_str() {
            "text" => {
                let provider = TextFileProvider::new(&cfg.name, &path);
                providers.push(Box::new(provider));
            },
            "csv" => {
                let provider = CSVFileProvider::new(&cfg.name, &path);
                providers.push(Box::new(provider));
            },
            _ => {
                eprintln!("Unknown provider kind in {:?}", cfg);
            }
        }
    }

    providers
}
```

We iterate through `config.providers`, construct a path for each resource, and then match on the `kind` value. So far we only have two event provider kinds, but when more appear, we can just bolt them on to this `match` expression as additional arms. If we don't recognize the provider kind, we print out a message to the standard error stream.

We create an instance of a specific kind of provider, put it in a `Box` as the parameter to the `Box::new` method, and then push the resulting trait object into the vector. At the end of the function we return the vector of trait objects to the caller.

The upshot of this is that using polymorphism we can have as many trait objects as we want, each implementing the `EventProvider` trait, and the `run` function does not need to care about which specific event providers it is dealing with. It just calls the `get_events` method, passing a vector, and the vector may or may not gain new elements as a result.

The new implementation of the `run` function could look like this:

**Example 29.7. Revised `run` function implementation**

```
pub fn run(config: &Config, config_path: &Path)
        -> Result<(), Box<dyn Error>> {
    birthday::handle_birthday();

    let mut events: Vec<Event> = Vec::new();

    let providers = create_providers(config, config_path);
    let mut count = 0;
    for provider in providers {
        provider.get_events(&mut events);  // polymorphism at work!
        let new_count = events.len();
        println!(
            "Got {} events from provider '{}'",
            new_count - count,
            provider.name());
        count = new_count;
    }

    let today: NaiveDate = Local::now().date_naive();
    let today_month_day = MonthDay::new(today.month(), today.day());

    for event in events {
        if today_month_day == event.month_day() {
            println!("{}", event);
        }
    }

    Ok(())
}
```

This implementation first instantiates all the event providers using the `create_providers` function, then iterates through them all to call their `get_events` methods, accumulating any events they provide in the `events`. To be able to monitor things, this implementation also shows how many events each provider contributed to the total. This is something we may want to leave out of the final program, but it is useful information at the development stage.

## Preventing duplicate event providers

The initial implementation of the `create_providers` does not satisfy the requirement stated at the beginning of this chapter: event providers should not have duplicate names. It would have cluttered the happy path in the code above, so we'll deal with it separately now.

To find out if there already is an event provider with a given name in the `providers` vector we need to do a quick check using an iterator:

**Example 29.8. Checking for existing event providers by name**

```
fn create_providers(config: &Config, config_path: &Path) -> Vec::<Box<dyn EventProvider>> {
    let mut providers: Vec::<Box<dyn EventProvider>> = Vec::new();
    for cfg in config.providers.iter() {
        let found = providers.iter().any(|p| p.name() == cfg.name);  // note the closure!
        if found {
            eprintln!("Event provider {} already exists", &cfg.name);
            continue;
        }

        // ...and so on
    }
}
```

We get an iterator to the `providers` using its `iter` method. The `any` method of an iterator applies a closure to each item, and returns `true` if that closure returns `true` for any item. The items in the `providers` vector all implement the `EventProvider`, so they all have the `name` method. The closure compares the result of calling that method to the name found in the configuration item currently being handled, and returns `true` if they match exactly. We will look at closures

again in Chapter 32, *Filtering events at the source* and Chapter 37, *Grouping and sorting the events*.