Aplicaciones con RUST

Dr. Casimiro Gómez González
Departamento de I+D, SMARTEST
correo: casimiro.gomez@smartest.mx
Tel: 222 707 4118

Primavera 2022

Prólogo

El presente proyecto presenta las bases del desarrollo de un **API** basado en RUST. Los conceptos básicos del API han sido desarrollado a lo largo del presente documento desde sus bases hasta la primera aplicación de envío de correos. ´

El autor
Casimiro Gómez González
Departamento de I+D, **SMARTEST**

Índice general

Capítulo 1

Diseño de la estructura APP

1.1. Estructura del Código APP

La estructuración del código es una parte importante del desarrollo de cualquier aplicación web. Debido a esto, tenemos que sentirnos cómodos dividiendo un problema en componentes que Rust pueda administrar y ejecutar. Para nuestro ejercicio, crearemos un programa de tareas simple en el que podemos crear, actualizar y eliminar elementos de tareas a través de una línea de comandos. Esta es una aplicación sencilla. El proceso aquí es explorar cómo construir un código bien estructurado que sea escalable sin meterse en la complejidad de la lógica de la aplicación. Para construir esto bien en Rust, tendremos que dividir los procesos en partes:

- Cree estructuras para elementos pendientes y tareas pendientes.
- Cree una fábrica(***factory***) que permita que las estructuras se construyan en el módulo **tienda_app**.
- Cree rasgos(***traits***) que permitan que una estructura elimine, cree, edite y obtenga los elementos de tareas pendientes. Luego, estos se importan a la fábrica para que las estructuras pendientes y terminadas puedan implementarlos.
- Cree un módulo de lectura y escritura en archivo para que lo utilicen otros módulos.
- Cree un módulo de configuración que pueda alterar el comportamiento de la aplicación según el entorno.

Antes de comenzar a abordar estos puntos, pongamos en marcha nuestra aplicación. Navegue hasta el directorio deseado e inicie un nuevo proyecto **Cargo** llamado ***tienda_app***. Una vez hecho esto, pondremos nuestra lógica relacionada con las tareas pendientes en un módulo ***tienda***. Esto se puede lograr creando un directorio ***tienda*** y colocando un archivo **mod.rs** en la base:

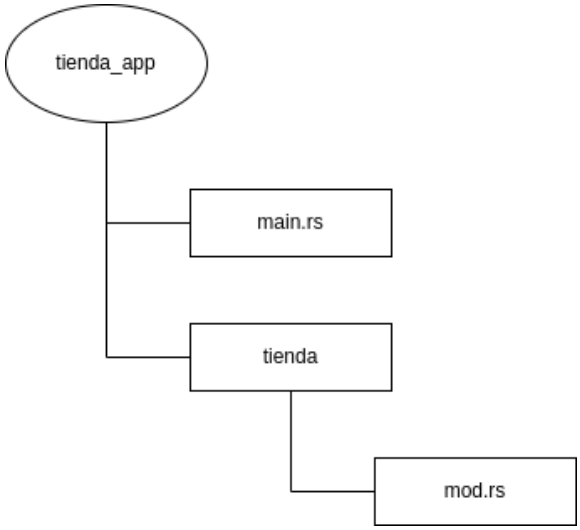

Figura 1.1: Estructura básica de programa

Con esto, estamos listos para comenzar a construir nuestras estructuras de tareas para poder importarlas y usarlas en la función principal.

1.2. Construyendo la estructura de la tienda

En este momento, tenemos dos tipos diferentes de elementos pendientes: un elemento pendiente y un elemento terminado. Ambos tendrán los mismos atributos, título y estado. Sin embargo, como recordamos con las características, sería ventajoso para nosotros tener dos estructuras diferentes, ya que queremos la máxima flexibilidad para definir la funcionalidad de cada tipo de elemento.

También es posible que deseemos agregar un tipo de elemento de tarea diferente en el futuro. Debido a esto, es lógico tener una estructura base que contenga los atributos comunes y luego dos estructuras que hereden la estructura base: una para el elemento pendiente y otra para el elemento terminado. Para lograr esto, necesitamos la siguiente estructura de directorios en nuestro módulo:

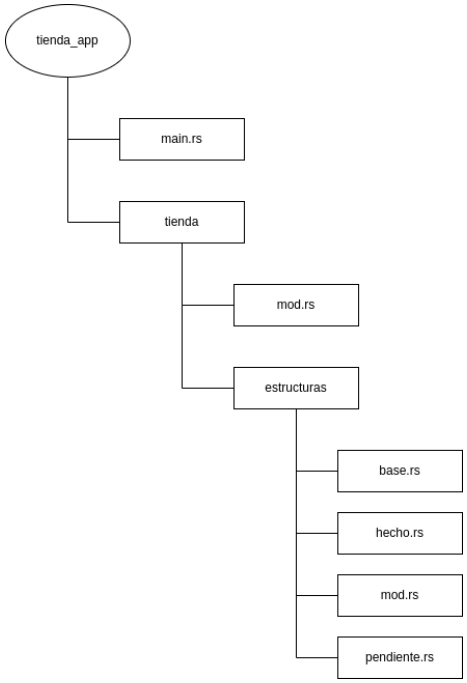

Figura 1.2: Estructura básica de la tienda

Podemos ver que hemos creado otro directorio, **estructuras**, y hemos alojado todas nuestras estructuras allí. En **base.rs**, definimos la estructura base:

```
1  pub struct Base{
2          pub titulo: String,
3          pub estatus: String
4  }
5  impl Base {
6          pub fn nuevo(titulo_entrada: &str, estatus_entrada: &str) ->
               Base {
7                  return Base {titulo: titulo_entrada.to_string(),
8                          estatus: estatus_entrada.to_string()}
9          }
10 }
```

Aquí tenemos una estructura estándar con un constructor. También debemos tener en cuenta que hay una palabra clave **pub** antes de las definiciones de función, estructura y atributo. Esto se debe a que nuestro objetivo es utilizar esta estructura fuera del archivo. Si no los declaráramos como públicos, el compilador se negaría a compilar si los usáramos externamente.

Ahora que lo hemos definido como público, tenemos que declararlo en nuestro

archivo **tienda_app/estructuras/mod.rs**:

```
mod base
```

Esto permite que otros archivos dentro del módulo accedan al archivo base. Sin embargo, debido a que solo queremos que nuestra estructura base se use dentro del módulo, no la hacemos pública.

Queremos usar nuestra estructura base dentro del módulo pero no externamente. Ahora que hemos hecho que nuestra clase base sea accesible para el resto del módulo, podemos definir nuestro elemento de tarea pendiente en el archivo **hecho.rs**:

```
user super::base::Base;

pub struct Hecho{
        pub super_estructura: Base
}
impl Hecho {
        pub fn nuevo(titulo_entrada: &str) -> Hecho {
                let base: Base = Base::nuevo(titulo_entrada, "hecho"
                    );
                return Hecho{super_estructura: base}
        }
}
```

Aquí, accedemos a la estructura base a través del archivo tienda/estructuras/-mod.rs usando **super** en la línea de importación en la parte superior del archivo. También bloqueamos el estado en el constructor (la nueva función) y construimos nuestra estructura base con esto. Hacemos esto para asegurarnos de que una estructura Terminada no defina otro estado aparte de Terminado.

Hacemos lo mismo para nuestro elemento de tarea pendiente en nuestro archivo **pendiente.rs**:

```
use super::base::Base;

pub struct Pendiente{
        pub super_estructura: Base
}
impl Pendiente{
        pub fn nuevo(titulo_entrada: &str) -> Pendiente {
                let base: Base = Base::nuevo(titulo_entrada, "
                    pendiente");
                return Pendiente{super_estructura: base}
        }
}
```

Ahora que tenemos las estructuras que necesitamos, podemos definirlas públicamente en nuestro archivo tienda/estructuras/mod.rs para permitir que el archivo principal las use:

```
mod base
pub mod hecho
pub mod pendiente
```

Ahora que nuestras estructuras están listas, podemos hacer que estén disponibles, definiéndolas públicamente en el archivo **tienda/mod.rs**:

```
pub mod estructuras;
```

Ahora nuestro módulo está listo para usar en la función principal con lo siguiente:

```
mod tienda;

use tienda::estructuras::hecho::Hecho;
use tienda::estructuras::pendiente::Pendiente;

fn main() {
        let hecho: Hecho = Hecho::nuevo("shopping");
        println!("{}", hecho.super_struct.titulo);
        println!("{}", hecho.super_struct.estatus);
}
```

Aquí tenemos una estructura estándar con un constructor. También debemos tener en cuenta que hay una palabra clave **pub** antes de las definiciones de función, estructura y atributo. Esto se debe a que nuestro objetivo es utilizar esta estructura fuera del archivo. Si no los declaráramos como públicos, el compilador se negaría a compilar si los usáramos externamente.

Ahora que lo hemos definido como público, tenemos que declararlo en nuestro archivo tienda/estructuras/mod.rs:

```
mod base;
```

Esto permite que otros archivos dentro del módulo accedan al archivo base. Sin embargo, debido a que solo queremos que nuestra estructura base se use dentro del módulo, no la hacemos pública.

Queremos usar nuestra estructura base dentro del módulo pero no externamente. Ahora que hemos hecho que nuestra clase base sea accesible para el resto del módulo, podemos definir nuestro elemento de tarea pendiente en el archivo hecho.rs:

```
use super::base::Base;

pub struct Hecho {
        pub super_struct: Base
```

```
}
impl Hecho {
        pub fn new(input_title: &str) -> Hecho {
                let base: Base = Base::nuevo(input_title,
                "done");
                return Hecho{super_struct: base}
        }
}
```

Aquí, accedemos a la estructura base a través del archivo **tienda/estructuras/mod.rs** usando **super** en la línea de importación en la parte superior del archivo. También bloqueamos el estado en el constructor (la nueva función) y construimos nuestra estructura base con esto. Hacemos esto para asegurarnos de que una estructura **Terminada** no defina otro estado aparte de **Terminado**.

Hacemos lo mismo para nuestro elemento de tarea pendiente en nuestro archivo **pendiente.rs**:

```
use super::base::Base;

pub struct Pendiente{
        pub super_estructura: Base
}
impl Pendiente{
        pub fn nuevo(titulo_entrada: &str) -> Pendiente {
                let base: Base = Base::nuevo(titulo_entrada, "
                   pendiente");
                return Pendiente{super_estructura: base}
        }
}
```

Ahora que tenemos las estructuras que necesitamos, podemos definirlas públicamente en nuestro archivo **tienda/estructuras/mod.rs** para permitir que el archivo principal las use:

```
mod base;
pub mod hecho;
pub mod pendiente;
```

Ahora que nuestras estructuras están listas, podemos hacer que estén disponibles definiéndolas públicamente en el archivo **tienda/mod.rs**:

```
pub mod estructuras;
```

Ahora nuestro módulo está listo para usar en la función principal con lo siguiente:

```
mod tienda;

use tienda::estructuras::hecho::Hecho;
```

```
use tienda::estructuras::pendiente::Pendiente;


fn main() {
        let hecho: Hecho = Hecho::nuevo("Comprando");
        println!("{}", hecho.super_estructura.titulo);
        println!("{}", hecho.super_estructura.estatus);
        let pendiente: Pendiente = Pendiente::nuevo("Lavandaria");
        println!("{}", pendiente.super_estructura.titulo);
        println!("{}", pendiente.super_estructura.estatus);
}
```

Aquí, podemos ver que definimos nuestro módulo y luego importamos las dos estructuras que queremos usar. Luego los definimos y luego los imprimimos.

Esto es útil, sin embargo, a medida que el programa crece, podríamos terminar con largas listas de importación a medida que importamos cada estructura pública que alberga el módulo. Esto tampoco es escalable. Si necesitáramos usar nuestro módulo en otro módulo, también tendríamos que reescribir muchas importaciones. Otros desarrolladores también podrían implementar nuestro módulo de forma incorrecta. Para evitar que ocurran estos problemas, podemos construir una interfaz. Vamos a discutir esto más en la siguiente sección.

1.3. Manejando estructuras con fábricas

Podemos construir nuestra interfaz con el patrón de fábrica. Aquí es donde seleccionamos la estructura correcta en función de la entrada, la construimos y la devolvemos. Esto se puede hacer en el archivo **tienda/mod.rs**:

```
pub mod estructuras;

use estructuras::hecho::Hecho;
use estructuras::pendiente::Pendiente;

pub enum TiposItem {
        Pendiente(Pendiente),
        Hecho(Hecho)
}
pub fn fabrica_tienda(tipo_item: &str, titulo_item: &str) -> Result<
    TiposItem, &'static␣str>␣{
␣␣␣␣␣␣␣␣if␣tipo_item␣==␣"pendiente"␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣item_pendiente␣=␣Pendiente::nuevo(titulo_item);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Ok(TiposItem::Pendiente(item_pendiente))
␣␣␣␣␣␣␣␣}
```

```
15 ␣␣␣␣␣␣␣␣else␣if␣tipo_item␣==␣"hecho"␣{
16 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣item_hecho␣=␣Hecho::nuevo(titulo_item);
17 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Ok(TiposItem::Hecho(item_hecho))
18 ␣␣␣␣␣␣␣␣}
19 ␣␣␣␣␣␣␣␣else␣{
20 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Err("Esto␣no␣es␣aceptado")
21 ␣␣␣␣␣␣␣␣}
22 }
```

Aquí, bloqueamos las estructuras eliminando la definición de publicación, ya que solo permitiremos que se use a través de la interfaz, que es la función **fabrica_tienda**. En esta función, verificamos el tipo de entrada y construimos la estructura según ese tipo. También empaquetamos un error si pasamos un tipo que no tenemos. También podemos ver que hemos utilizado una enumeración para habilitar la devolución de los dos tipos de elementos.

En este punto, hay una oportunidad de refactorización. Se podría argumentar que solo necesitábamos una estructura y que el tipo podía manejarse en la fábrica, reduciendo la necesidad de múltiples estructuras. Esta es una observación verdadera. Sin embargo, planeamos comenzar a agregar características a nuestras estructuras. En este momento, las estructuras múltiples pueden parecer un poco excesivas, pero debemos mantener la flexibilidad en nuestro código.

Ahora que nuestra interfaz está definida, podemos utilizar esto en la función principal llamando a la fábrica con algunos parámetros y usando una declaración de coincidencia:

```
1  mod tienda;
2  use tienda::TiposItem;
3  use tienda::fabrica_tienda;
4
5  fn main() {
6          let tienda_item: Result<TiposItem, &'static␣str>␣=␣
               fabrica_tienda("pendiente",␣"hacer");
7  ␣␣␣␣␣␣␣␣match␣tienda_item.unwrap()␣{
8  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣TiposItem::Pendiente(item)␣=>␣println!("Esto␣es␣un␣
       item␣pendiente␣con␣titulo:␣{}",␣item.super_estructura.titulo),
9  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣TiposItem::Hecho(item)␣=>␣println!("Esto␣es␣un␣item␣
       hecho␣con␣titulo:␣{}",␣item.super_estructura.titulo)
10 ␣␣␣␣␣␣␣␣}
11 }␣␣␣␣␣␣␣␣
```

Esto puede parecer excesivo por ahora: la simple verificación de parámetros y la obtención de la estructura podrían haberse realizado en la función principal. Sin embargo, a medida que crece la complejidad, **main** se volverá inmanejable. Es mejor mantener la lógica que rodea la definición y la construcción de tareas pendientes en

su propio módulo.

En este momento, nuestros elementos de tareas pendientes no hacen nada. Albergan diferentes atributos, pero no pueden eliminar, guardar, actualizar u obtener. Todo lo que pueden hacer son los atributos de la casa. Para definir la funcionalidad, necesitamos darle a nuestras estructuras algunos rasgos.

1.4. Definiendo funcionalidad con rasgos (traits)

Ahora que tenemos nuestros rasgos, es hora de que tengan alguna funcionalidad. Vamos a hacer esto con rasgos. Para escalabilidad y flexibilidad, mantendremos nuestros rasgos tan simples y aislados como sea posible. Definir un rasgo para cada tipo de proceso nos brinda la capacidad de ajustar la funcionalidad para cada tipo.

Para lograr esto, debemos agregar los rasgos del directorio dentro del directorio **estructuras**. Dentro del directorio de rasgos, tenemos un archivo para cada proceso y un archivo **mod.rs** para declararlos (ver figura 1.3).

Primero, definamos funciones que impriman el proceso. Interactuaremos con el entorno de lectura y escritura en un archivo en la siguiente sección. Podemos deducir que las funciones de creación, eliminación, edición y obtención de características requerirán un título. Por lo tanto, en el archivo **traer**, podemos definir el siguiente rasgo:

```
1 pub trait Traer {
2         fn traer(&self, titulo: &str) {
3                 println!("{} esta siendo recuperada", titulo);
4         }
5 }
```

Aquí, vinculamos la función **traer** con el parámetro **&self**, que permite que la estructura llame a la función directamente como **some_struct.traer(&String::from("something"))**. También tomamos el título. Nuevamente, por ahora, solo imprimiremos una declaración para garantizar que funcione el mapeo de rasgos en múltiples archivos.

Con el proceso de edición, podemos tener múltiples funciones para diferentes procesos. En este momento, todo lo que necesitamos es una función para configurar un elemento como hecho y otra función para configurar un elemento como pendiente. Podemos definir estos en el archivo **editar.rs**:

```
1 pub trait Editar {
2         fn ajustar_a_hecho(&self, titulo: &str) {
3                 println!("{} esta siendo ajustado a hecho", titulo);
4         }
5         fn ajustar_a_pendiente(&self, titulo: &str) {
```

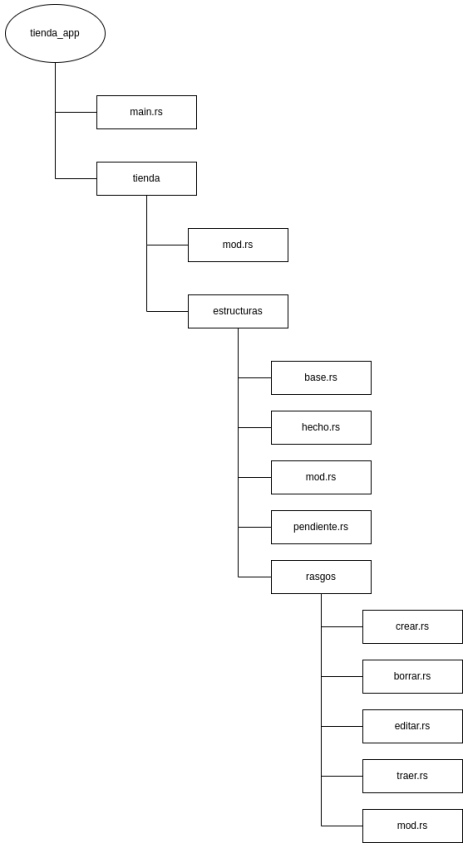

Figura 1.3: Estructura básica de la tienda con rasgos

```
6                 println!("{} esta siendo ajustado a pendiente",
                      titulo);
7         }
8 }
```

En términos de **Crear**, solo hay una función que necesitamos, que es **crear**. Esto se puede definir en el archivo **crear.rs**:

```
1 pub trait Crear {
2         fn crear(&self, titulo: &str) {
3                 println!("{} esta siendo creado", titulo);
4         }
5 }
```

Similar a esto, todo lo que necesitamos es una sola función de eliminación en el rasgo Borrar en el archivo **borrar.rs**:

```
1 pub trait Borrar {
```

```
2         fn borrar(&self, titulo: &str) {
3                 println!("{} esta siendo borrado", titulo);
4         }
5 }
```

Ahora que tenemos todos nuestros rasgos, tenemos que definirlos públicamente en el archivo mod del directorio de rasgos para que se pueda acceder a ellos desde el exterior:

```
1 pub mod crear;
2 pub mod borrar;
3 pub mod editar;
4 pub mod traer;
```

Ahora que son accesibles, tenemos que hacerlos accesibles para las estructuras definiéndolos públicamente en el archivo **mod.rs** en el directorio de estructuras:

```
1 pub mod rasgos;
2 mod base;
3 pub mod hecho;
4 pub mod pendiente;
```

Esto permite que nuestras estructuras usen **super** para acceder a los rasgos. Para la estructura **Hecho**, debemos permitir que el programa obtenga, edite y elimine implementando estos rasgos en la estructura **Hecho**:

```
1 use super::base::Base;
2 use super::rasgos::traer::Traer;
3 use super::rasgos::borrar::Borrar;
4 use super::rasgos::editar::Editar;
5
6 pub struct Hecho{
7         pub super_estructura: Base
8 }
9 impl Hecho {
10        pub fn nuevo(titulo_entrada: &str) -> Hecho {
11                let estatus_entrada: String = String::from("hecho");
12                let base: Base = Base::nuevo(titulo_entrada, "hecho"
                      );
13                return Hecho{super_estructura: base}
14        }
15 }
16 impl Traer for Hecho {}
17 impl Borrar for Hecho {}
18 impl Editar for Hecho {}
```

Para la estructura **Pendiente**, vamos a habilitarla para crear, editar, obtener y eliminar:

```
use super::base::Base;
use super::rasgos::crear::Crear;
use super::rasgos::editar::Editar;
use super::rasgos::traer::Traer;
use super::rasgos::borrar::Borrar;

pub struct Pendiente{
        pub super_estructura: Base
}
impl Pendiente{
        pub fn nuevo(titulo_entrada: &str) -> Pendiente {
                let base: Base = Base::nuevo(titulo_entrada, "
                   pendiente");
                return Pendiente{super_estructura: base}
        }
}
impl Crear for Pendiente{}
impl Editar for Pendiente{}
impl Traer for Pendiente{}
impl Borrar for Pendiente{}
```

Ahora que las estructuras están mejoradas con nuestros rasgos, podemos ver cuán escalable es esto. Si añadimos otra estructura de tareas pendientes, podemos colocar una variedad de rasgos para darle instantáneamente la funcionalidad que necesitamos. También podemos eliminar/agregar rasgos de/a nuestras estructuras existentes con facilidad. Esto demuestra el poder que nos dan las estructuras. Ahora que tenemos nuestros rasgos, podemos demostrar cómo se pueden usar en main con lo siguiente:

```
mod tienda;
use tienda::TiposItem;
use tienda::fabrica_tienda;
use tienda::estructuras::rasgos::crear::Crear;

fn main() {
        let tienda_item: Result<TiposItem, &'static␣str>␣=␣
           fabrica_tienda("pendiente",␣"lavando");
␣␣␣␣␣␣␣␣match␣tienda_item.unwrap()␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣TiposItem::Pendiente(item)␣=>␣item.crear(
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣&item.super_estructura.titulo),
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣TiposItem::Hecho(item)␣=>␣println!(
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"Esta␣hecho␣el␣item␣con␣titulo:␣{}",
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣item.super_estructura.titulo)␣␣␣
␣␣␣␣␣␣␣␣}␣␣␣␣␣␣␣␣
}
```

Debe tenerse en cuenta que importamos el rasgo Create en main. Aunque el rasgo Crear se implementa para la estructura Pendiente, no podrá activarse si no se importa porque el compilador no lo encontrará.

Lo que hemos hecho aquí es construir nuestro propio módulo, que contiene un punto de entrada. Luego lo importamos a la función principal y lo ejecutamos. Ahora que la estructura básica está construida y funcionando, necesitamos que el módulo interactúe con el entorno pasando variables y escribiendo en un archivo para que sea útil.

1.5. Interacción con el ambiente

Para interactuar con el entorno, tenemos que manejar dos cosas. Primero, necesitamos cargar, guardar y editar el estado de las tareas pendientes. En segundo lugar, también tenemos que aceptar la entrada del usuario para editar y mostrar datos. Nuestro programa puede lograr esto ejecutando los siguientes pasos para cada proceso:

- Recopilar argumentos del usuario.
- Defina un comando (obtener, editar, eliminar y crear) y defina un título de tarea a partir de los comandos.
- Cargue un archivo **JSON** que almacene los elementos pendientes de ejecuciones anteriores del programa.
- Ejecute una función de obtención, edición, eliminación o creación basada en el comando pasado al programa, guardando el resultado del estado en un archivo **JSON** al final.
- Podemos comenzar a hacer posible este proceso de cuatro pasos cargando inicialmente nuestro estado con la caja de **serde**.

1.5.1. Leyendo y escribiendo archivos JSON

Para instalar el **crate serde**, definimos la dependencia en el archivo **Cargo.toml** en la sección [dependencias]:

```
1 [dependencies]
2 serde_json = {version = "1.0.79", default-features = false, features
    = ["alloc"]}
```

Para administrar la lectura y escritura en el archivo JSON, tiene sentido crear nuestro propio módulo, ya que habrá varios lugares donde escribiremos datos en el archivo. Sin embargo, el módulo en sí será bastante pequeño y consistirá solo en funciones de lectura y escritura. No sería práctico dedicarle un directorio completo. Podemos crear nuestro propio módulo con solo un archivo en el mismo directorio del archivo principal. En nuestro archivo **src/estado.rs**, definimos los métodos de lectura y escritura

```
use std::fs::File;
use std::fs;
use std::io::Read;
use serde_json::Map;
use serde_json::value::Value;
use serde_json::json;

pub fn leer_archivo(nombre_archivo: &str) -> Map<String, Value> {
        let mut archivo = File::open(nombre_archivo.to_string()).
            unwrap();
        let mut datos = String::new();
        archivo.read_to_string(&mut datos).unwrap();
        let json: Value = serde_json::from_str(&datos).unwrap();
        let estado: Map<String, Value> = json.as_object().unwrap().
            clone();
        return estado
}
```

En la función de lectura, tomamos la ruta del archivo como una cadena y usamos la biblioteca estándar para abrirlo. Directamente lo desenvolvemos. Si hay un error aquí, entonces no tiene sentido continuar con el programa. Luego definimos una cadena mutable con el nombre de datos y leemos el archivo en esa cadena (recuerde que las cadenas son referencias a cadenas literales).

Luego usamos **serde** para convertir esa cadena en un valor JSON y luego definimos ese valor como un objeto y lo clonamos para obtener un mapa **serde JSON**. Si no lo clonamos, simplemente estaremos devolviendo una referencia. Nos tomamos la molestia de convertirlo en un mapa para obtener la funcionalidad adicional.

Más abajo en el archivo **src/estado.rs**, definimos el archivo de escritura:

```
pub fn escribir_a_archivo(nombre_archivo: &str, estado: &mut Map<
    String, Value>) {
        let nuevo_dato = json!(estado);
        fs::write(nombre_archivo.to_string(), nuevo_dato.to_string()
            ).expect("No␣se␣pudo␣escribir␣el␣archivo");
}
```

Aquí, aceptamos la ruta del archivo y el mapa, volvemos a convertir el mapa a JSON usando la macro y luego lo convertimos en una cadena para escribir en el archivo.

Para verificar si esto funciona, debemos importarlo a **main**, leer un archivo **JSON**, obtener algunos parámetros del usuario y escribir una nueva entrada en el archivo:

```
mod estado;

use std::env;
use estado::{escribir_a_archivo, leer_archivo};
use serde_json::value::Value;
use serde_json::{Map, json};

fn main() {

        let args: Vec<String> = env::args().collect();
        let estatus: &String = &args[1];
        let titulo: &String = &args[2];
        let mut estado: Map<String, Value> = leer_archivo("./estado.
           json");
        println!("{:?}", estado);
        estado.insert(titulo.to_string(), json!(estatus));
        escribir_a_archivo("./estado.json", &mut estado);
}
```

Aquí, recopilamos los argumentos del entorno pasados por el usuario y los recopilamos en un vector de cadenas. Luego definimos los comandos del vector args. Una vez que hayamos hecho eso, cargamos los datos del archivo JSON y los imprimimos usando la notación de depuración. Un resultado de ejemplo (según el contenido del archivo JSON) de esta instrucción de impresión es el siguiente:

```
{"compras": String("pendiente"), "lavado": String("hecho")}
```

Luego insertamos la nueva entrada y luego escribimos en un archivo. Nuestra ruta raíz será donde está el archivo **Cargo.toml**, por lo que definimos un archivo JSON vacío llamado **estado.json** junto al archivo **Cargo.toml**. Para permitir que nuestras tareas pendientes interactúen con el estado, debemos habilitar nuestros rasgos de tareas pendientes para manipular el estado.

Se crea el archivo estado.rs con el siguiente contenido:

```
{"lola":"hola","shopping":"pending","washing":"done"}
```

1.6. Mejorando los rasgos

Teniendo en cuenta que se ha definido el estado, ahora tenemos una mejor idea de lo que realmente necesitan nuestras funciones de rasgos. Inicialmente podemos comenzar con nuestro rasgo más simple, que es obtener. Aquí, tenemos que obtener un elemento pendiente por el título del estado e imprimirlo. Si no está, imprimimos que el título no estaba en el mapa:

```
use serde_json::Map;
use serde_json::value::Value;

pub trait Traer {
        fn traer(&self, titulo: &str, estado: &Map<String, Value>) {
                let item: Option<&Value> = estado.traer(titulo);
                match item {
                        Some(resultado) => {
                                println!("\n\nItem: {}", titulo);
                                println!("Estatus: {}\n\n",
                                   resultado)
                        },
                        None => println!("item: {} no fue encontrado
                           ", titulo)
                }
        }
}
```

Aquí, tomamos el estado, llamamos a la función get para el mapa y luego lo administramos con una declaración de coincidencia e imprimimos el resultado.

El rasgo que podemos abordar es el rasgo Crear. En este rasgo, necesitamos insertar nuestra nueva entrada en el estado y luego usar nuestra función **escribir_a_archivo** para escribir el estado actualizado en el archivo JSON que estamos usando para almacenar nuestros elementos pendientes:

```
use serde_json::Map;
use serde_json::value::Value;
use serde_json::json;
use crate::estado::escribir_a_archivo;
pub trait Crear {
        fn crear(&self, titulo: &str, estatus: &String, estado: &mut
            Map<String, Value>) {
                estado.insert(titulo.to_string(), json!(estatus));
                escribir_a_archivo("./estado.json", estado);
                println!("{} esta siendo creado", titulo);
        }
}
```

Aquí, podemos ver que usamos el módulo de estado definido en ***main*** con el comando use ***crate***. El rasgo de eliminación tiene el mismo enfoque pero con la función de eliminación se llama en el mapa:

```
use serde_json:.Map;
use serde_json::value::Value;
use crate::estado::escribir_a_archivo;
pub trait Borrar {
        fn borrar(&self, titulo: &str, estado: &mut Map<String,
           Value>) {
                estado.remove(titulo);
                escribir_a_archivo("./estado.json", estado);
                println!("{} esta siendo borrado", titulo);
        }
}
```

Para el rasgo Editar, necesitamos dos funciones: una para establecer una tarea pendiente como pendiente y otra para configurar una tarea pendiente como lista en el estado, como se muestra:

```
use serde_json::Map;
use serde_json::value::Value;
use serde_json::json;
use crate::estado::escribir_a_archivo;
pub trait Editar {
        fn ajustar_a_hecho(&self, titulo: &str, estado:: &mut Map<
           String, Value>) {
                estado.insert(titulo.to_string(), json!(String::from
                   ("hecho")));
                escribir_a_archivo("./estado.json", estado);
                println!("{} esta siendo ajustado a hecho", titulo);
        }
        fn ajustar_a_pendiente(&self, titulo: &str, estado: &mut Map
           <String, Value>) {
                estado.insert(titulo.to_string(), json!(String::from
                   ("pendiente")));
                escribir_a_archivo("./estado.json", estado)
                println!("{} esta siendo ajustado a pendiente",
                   titulo);
        }
}
```

Nuestros rasgos ahora pueden interactuar con el archivo JSON y realizar los procesos para los que están destinados. Sin embargo, la simple interacción directa con los rasgos en el archivo principal no es escalable. A medida que el programa crezca, es posible que necesitemos usar estos rasgos en otro lugar. Para protegernos

contra esto, podemos crear nuestra propia interfaz para administrar el uso de estos rasgos creando nuestro propio módulo de proceso.

1.7. Procesando rasgos y estructuras

Al igual que nuestro módulo de estado, el módulo de proceso solo necesitará algunas funciones. Por lo tanto, podemos definir el módulo en un archivo **src/procesos.rs**. El propósito de este módulo es dirigir el flujo de los comandos. Necesitamos un punto de entrada para procesar la entrada y dirigirla a la función correcta para procesar el artículo. En primer lugar, importemos todas las estructuras y rasgos que necesitamos:

```
use serde_json::Map;
use serde_json::value::Value;
use super::tienda::TiposItem;
use super::tienda::estructuras::hecho::Hecho;
use super::tienda::estructuras::pendiente::pendiente;
use super::tienda::estructuras::rasgos::traer::Traer;
use super::tienda::estructuras::rasgos::crear::Crear;
use super::tienda::estructuras::rasgos::borrar::Borrar;
use super::tienda::estructuras::rasgos::editar::Editar;
```

Luego definimos las funciones que nos permiten procesar estructuras Terminadas y Pendientes:

```
fn proceso_pendiente(item: Pendiente, comando: String, estado: &Map<String, Value>) {
        let mut estado = estado.clone();
        match comando.as_str() {
                "traer" =>  item.traer(&item.super_struct.title, &estado),
                "crear" =>  item.crear(&item.super_struct.title, &item.super_struct.status, &mut estado),
                "borrar" => item.borrar(&item.super_struct.title, &mut estado),
                "editar" => item.ajustar_a_hecho(&item.super_struct.title, &mut estado),
                _ => println!("comando: {} no soportado", comando)

        }
}
fn proceso_hecho(item: Done, comando: String, estado: &Map<String, Value>) {
        let mut estado = estado.clone();
```

```
14          match comando.as_str() {
15                  "traer" => item.traer(&item.super_struct.title, &
                        estado),
16                  "borrar" => item.borrar(&item.super_struct.title, &
                        mut estado),
17                  "editar" => item.ajustar_a_pendiente(&item.
                        super_struct.title, &mut estado),
18                  _ => println!("comando: {} no soportado", comando)
19          }
20  }
```

Ahora que hemos definido funciones que procesan nuestras estructuras pendientes, podemos crear un punto de entrada que tome una estructura, un estado de memoria y un comando para que podamos canalizar la estructura hacia la función correcta:

```
1  pub fn proceso_entrada(item: ItemTypes, comando: String, estado: &
       Map<String, Value>) {
2           match item {
3                   TiposItem::Pendiente(item) => process_pending(item,
                        comando, estado),
4                   TiposItem::Hecho(item) => process_done(item, comando
                        , estado)
5           }
6   }
```

Lo que tenemos aquí es esencialmente una declaración de coincidencia que se asigna a otras declaraciones de coincidencia. Esto nos da mucha flexibilidad. Si vamos a agregar un tipo, todo lo que tenemos que hacer es agregar una línea en la declaración de coincidencia de la función proceso_entrada (nuestro punto de entrada). También podemos agregar declaraciones condicionales adicionales en las funciones. Podemos eliminar y agregar comandos rápidamente porque cualquier cosa agregada que no coincida es capturada por el operador _.

Cabe señalar que tenemos que importar los rasgos al archivo. Aunque los rasgos han sido implementados por las estructuras en el módulo de tareas pendientes, el archivo que llama al rasgo vinculado a la estructura aún debe importarse para ser reconocido. Esta interfaz se puede pasar por el programa y utilizarse en cualquier lugar. Si otro desarrollador necesita otro punto de entrada donde se procesa una tarea pendiente, entonces sabemos que procesará las tareas pendientes de manera estandarizada.

Ahora que todos los módulos son completamente funcionales y funcionan entre sí, podemos implementarlos en la función principal:

```
1  mod estado;
2  mod tienda;
```

```
3  mod procesos;
4
5  use std::env;
6  use estado::leer_archivo;
7  use serde_json::value::Value;
8  use serde_json::Map;
9  use tienda::fabrica_tienda;
10 use procesos::proceso_entrada;
11
12 fn main() {
13         let args: Vec<String> = env::args().collect();
14         let comando: &String = &args[1];
15         let titulo: &String = &args[2];
16         let estado: Map<String, Value> = leer_archivo("./estado.json
               ");
17         let estatus: String;
18
19         match &estado.get(*&titulo) {
20                 Some(resultado) => {
21                         estatus = resultado.to_string().replace
                           ('\"', "");
22                 }
23                 None=> {
24                         estatus = "pendiente".to_string();
25                 }
26         }
27         let item = fabrica_tienda(&estatus, titulo).expect(&estatus)
               ;
28         proceso_entrada(item, comando.to_string(), &estado);
29 }
```

Aquí, todo lo que hemos aprendido está en exhibición. Obtenemos los argumentos del entorno, definimos los comandos y leemos los datos del archivo JSON para obtener el estado de la lista de tareas pendientes. Luego utilizamos lo que sabemos sobre los ámbitos, definiendo el estado como una cadena fuera del bloque de coincidencia para que podamos confiar en el estado fuera del bloque de coincidencia. En el bloque de coincidencia, hacemos una suposición. Si el elemento no existe en el estado, definimos el estado como pendiente. No tendría sentido crear un nuevo elemento terminado. Luego pasamos el estado a la interfaz o nuestro módulo de tareas pendientes. Finalmente, pasamos nuestra estructura, comando y estado de elementos al punto de entrada de nuestro módulo de proceso.

Ahora que hemos construido nuestra aplicación, podemos probarla completamente a través del siguiente procedimiento:

```
1 cargo run
```

Nuestro programa falla porque el índice está fuera de los límites. Esto se debe a que no pusimos ningún comando. Si ejecutamos lo siguiente:

```
cargo run crear lavando
```

Recibimos un mensaje de que se está creando el lavado y nuestro archivo JSON vacío ahora se ve así:

```
{"lavando":"pendiente"}
```

Ejecutar el comando **traer** (**cargo run traer lavando**) nos da la siguiente impresión:

```
Articulo: lavado
Estado: Pendiente"
```

Ejecutando el comando de edición (**cargo run editar lavando**), obtenemos una impresión que nos dice que el lavado se ha configurado como hecho, y nuestro archivo **JSON** se ve así:

```
{"lavado":"hecho"}
```

Ejecutar el comando de eliminación (**cargo run borrar lavando**) elimina el elemento de lavado en el archivo **JSON**.

Básicamente, lo que hemos hecho aquí es crear un programa que acepte algunas entradas de la línea de comandos, interactúe con un archivo y lo edite según el comando y los datos de ese archivo. Los datos son bastante simples: un título y un estado.

Podríamos haber hecho todo esto en la función principal con múltiples sentencias de coincidencia y bloques if, else if y else. Sin embargo, esto no es escalable. En cambio, construimos estructuras que heredaron otras estructuras, que luego implementaron rasgos. Luego empaquetamos la construcción de estas estructuras en una fábrica que permite que otros archivos usen toda esa funcionalidad en una sola línea de código.

Luego construimos una interfaz de procesamiento para que la entrada de comando, el estado y la estructura pudieran procesarse, lo que nos permitió agregar funcionalidad adicional y cambiar el flujo del proceso con unas pocas líneas de código. Nuestra función principal solo tiene que centrarse en recopilar los argumentos de la línea de comandos y coordinar cuándo llamar a las interfaces del módulo. Ahora hemos explorado y utilizado cómo Rust administra los módulos, brindándonos los componentes básicos para crear programas del mundo real que pueden resolver problemas y agregar funciones sin verse perjudicados por la deuda tecnológica y una función principal creciente. Ahora que podemos hacer esto, estamos listos para comenzar a crear aplicaciones web escalables que pueden crecer.

En el próximo capítulo, aprenderemos sobre el marco web Actix para poner en funcionamiento un servidor web básico.

Capítulo 2

Diseño del servidor WEB

El poder de **Rust** es que permite a los usuarios desarrollar rápidamente con estructuras seguras de memoria de alto nivel con bajo consumo de memoria y tiempos de ejecución rápidos. Sin embargo, también permite un control de grano fino si es necesario. Si un desarrollador realmente quiere, puede desactivar la seguridad de la memoria en Rust y continuar desarrollando y ejecutando programas Rust (aunque no se recomienda). Los **crates** de Rust no son una excepción a esto. El **framework Actix Web** nos expone a algunos conceptos subyacentes del servidor web, invitándonos a modificarlos de forma segura si es necesario. El lanzamiento de un servidor web básico con el marco web de **Actix** introducirá algunos conceptos nuevos que debemos abordar antes de comenzar a desarrollar nuestras funciones.

2.1. Lanzamiento de un servidor web Actix básico

Para admitir un servidor **web Actix**, necesitamos crear un nuevo proyecto Cargo. Para habilitar la ejecución de un servidor **Actix**, debemos definir las siguientes dependencias en el archivo **Cargo.toml**:

```
1 [dependencies]
2 actix-web = "4.0.1"
3 actix-rt = "2.7.0"
```

actix-web es el marco principal que alberga las estructuras que definen las rutas y el servidor. **actix-rt** nos permite ejecutar todo en el subproceso actual.

El siguiente código es la implementación de ejemplo estándar que pone en marcha un servidor rápidamente mientras nos muestra las características del marco de forma concisa:

```
use actix_web::{web, App, HttpRequest, HttpServer, Responder};
async fn saludo(req: HttpRequest) -> impl Responder {
        let nombre = req.match_info().get("nombre").unwrap_or("Mundo
            ");
        format!("Hola {}!", nombre)
}

#[actix_rt::main]
async fn main() -> std::io::Result<()> {
        HttpServer::new(|| {
                App::new()
                .route("/", web::get().to(saludo))
                .route("/{nombre}", web::get().to(saludo))
        })
        .bind("127.0.0.1:8000")?
        .run()
        .await
}
```

Aquí, usamos las estructuras web de **Actix** para definir una vista que extrae datos de la solicitud. Luego, redefinimos nuestra función principal como una función asíncrona utilizando la macro de la caja **actix-rt**. Sin esta macro, el programa no se compilaría ya que las funciones principales no pueden ser asíncronas. Luego construimos un nuevo servidor y definimos las rutas asignándolas a la función que queremos. Luego nos vinculamos a una dirección, ejecutamos y luego esperamos el resultado.

Si bien la funcionalidad asíncrona es nueva, nos centraremos en esto más adelante en el capítulo. Por ahora, deberíamos centrar nuestra atención en el cierre que se pasa a la nueva función de la estructura **HttpServer**.

2.2. Comprender los cierres

Los cierres son esencialmente funciones; sin embargo, tienen algunas diferencias. Cabe señalar que las funciones se pueden definir sobre la marcha a través de || corchetes en lugar de () corchetes. Un ejemplo simple de esto es imprimir un parámetro:

```
let test_function: fn(String) = |string_input: &str| {
        println!("{}", string_input);
};

test_function("test");
```

Lo que sucede aquí es que asignamos la variable **test_function** a nuestra función que imprime la entrada. El tipo es un tipo **fn**. Hay algunas ventajas en esto. Debido a que es una variable que se asigna, podemos explotar los ámbitos.

Las funciones normales definidas por sí mismas están disponibles a través del archivo/módulo en el que se importan o definen. Sin embargo, habrá momentos en el desarrollo web en los que deseamos que la disponibilidad de la función se restrinja a una determinada duración. Cambiar el cierre a un alcance interno puede lograr esto fácilmente:

```
{
        let test_function: fn(String) = |string_input: &str| {

                println!("{}", string_input);
        };
}

test_function("test");
```

Aquí, la llamada de nuestra función está fuera del alcance, lo que dará como resultado una función que no se encuentra en este error de alcance.

Las definiciones de cierre tienen una sintaxis similar. Sin embargo, pueden ingresar parámetros e interactuar con variables externas en su alcance:

```
let test = String::from("test");
let test_function = || {
        println!("{}", test);
};

test_function();
```

Tenga en cuenta que no hemos definido un tipo para la variable **test_function**. Esto se debe a que un cierre es un tipo anónimo único que no se puede escribir. La analogía más cercana a un cierre es una estructura que alberga variables capturadas.

Los cierres también pueden tener valores de retorno con los que se puede interactuar como una función normal:

```
let test = String::from("test");
let test_function = || {
        println!("{}", test);
        return test + &String::from(" case")
};

let outcome: String = test_function();
```

Aquí, el resultado denotará la cadena devuelta del cierre bajo la variable **test_function**.

Ahora que tenemos una mayor comprensión de los cierres, podemos mirar hacia atrás a nuestra función principal con confianza. Sabemos que se está llamando a un cierre en la función **HttpServer::new**. Dado que la estructura de la aplicación es la línea final del cierre, la aplicación debe devolverse del cierre para que se activen

las funciones de enlace y ejecución. Con esta información sobre los cierres, podemos estar un poco más seguros con la creación de nuestro servidor **HTTP**:

```
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
        HttpServer::new(|| {
                println!("function esta disparada");
                let app = App::new()
                .route("/", web::get().to(saludo))
                .route("/{name}", web::get().to(saludo));
                return app
        })
        .bind("127.0.0.1:8000")?
        .workers(3)
        .run()
        .await
}
```

Aquí, definimos la aplicación y la devolvemos, y arrojamos una declaración de impresión. Podemos hacer lo que queramos en el cierre siempre que devolvamos una estructura de aplicación construida. También agregamos una función de trabajadores con el parámetro 3. Cuando ejecutamos esto, podemos ver que obtenemos el siguiente resultado:

```
    Finished dev [unoptimized + debuginfo] target(s) in 35.66s
Running `target/debug/web_rust`
function esta disparada
function esta disparada
function esta disparada
```

Esto nos dice que el cierre fue disparado tres veces. Alterar el número de trabajadores nos muestra que existe una relación directa entre este y el número de veces que se dispara el cierre. Si la función de los trabajadores se omite, el cierre se activa en relación con la cantidad de núcleos que tiene su sistema.

Ahora que comprendemos los matices en torno a la construcción de la estructura de la aplicación, es hora de ver el cambio principal en la estructura del programa, la programación asíncrona.

2.3. Entender la programación asíncrona

Hasta este capítulo, hemos estado escribiendo código de manera secuencial. Esto es lo suficientemente bueno para scripts estándar. Sin embargo, en el desarrollo web, la programación asíncrona es importante, ya que hay múltiples solicitudes a los servidores y las llamadas a la API introducen tiempo de inactividad. En algunos otros

lenguajes, como Python, podemos construir servidores web sin tocar ningún concepto asíncrono. Si bien los conceptos asincrónicos se utilizan en estos marcos web, la implementación se define bajo el capó. Esto también es cierto para el marco de trabajo de Rust Rocket. Sin embargo, como hemos visto, se implementa directamente en Actix Web.

Cuando se trata de utilizar código asíncrono, hay dos conceptos principales que debemos entender:

- **Procesos**: Un proceso es un programa que se está ejecutando. Tiene su propia pila de memoria, registros para variables y código.
- **Hilos(o subprocesos)**: un hilo o subproceso es un proceso ligero que un planificador gestiona de forma independiente. Sin embargo, comparte datos con otros hilos y el programa principal.

En esta sección, veremos los hilos para ver el efecto que tienen en nuestro código. Una de las mejores formas de explorar **subprocesos** en cualquier lenguaje de programación es codificar una breve pausa en cada **subproceso** y tiempo para procesar el programa en general. Podemos cronometrar nuestro programa Rust con el siguiente código:

```
use std::{thread, time};

fn do_something(number: i8) -> i8 {
        println!("number {} is running", number);
        let two_seconds = time::Duration::new(2, 0);
        thread::sleep(two_seconds);
        return 2
}

fn main() {
        let now = time::Instant::now();
        let one: i8 = do_something(1);
        let two: i8 = do_something(2);
        let three: i8 = do_something(3);
        println!("time elapsed {:?}", now.elapsed());
        println!("result {}", one + two + three);
}
```

Aquí, definimos una función estándar que duerme. Debe tenerse en cuenta que aunque estamos llamando a la función de suspensión desde el módulo de subprocesos de la biblioteca estándar, no hay nada en este código que genere un subproceso todavía. En la función principal, iniciamos el temporizador, activamos tres funciones,

luego detenemos el temporizador e imprimimos la suma de los resultados después. Con esto, obtenemos la siguiente salida:

```
number 1 is running
number 2 is running
number 3 is running
time elapsed 6.000599893s
result 6
```

Esto no es sorprendente. Tenemos las funciones ejecutándose secuencialmente. Cada función duerme durante 2 segundos, y el tiempo total transcurrido al final de todo el proceso es de poco más de 6 segundos.

Ahora vamos a crear un hilo para cada función:

```
use std::{thread, time};

use std::thread::JoinHandle;

fn do_something(number: i8) -> i8 {
        println!("number {} is running", number);
        let two_seconds = time::Duration::new(2, 0);
        thread::sleep(two_seconds);
        return 2
}

fn main() {
        let now = time::Instant::now();
        let thread_one: JoinHandle<i8> = thread::spawn(||
            do_something(1));
        let thread_two: JoinHandle<i8> = thread::spawn(||
            do_something(2));
        let thread_three: JoinHandle<i8> = thread::spawn(||
            do_something(3));
        let result_one = thread_one.join();
        let result_two = thread_two.join();
        let result_three = thread_three.join();
        println!("time elapsed {:?}", now.elapsed());
        println!("result {}", result_one.unwrap() + result_two.
            unwrap() + result_three.unwrap());
}
```

Aquí puede ver que pasamos un cierre a través de cada hilo. Si intentamos y simplemente pasamos la función **do_something** a través del hilo, obtenemos un error quejándose de que el compilador esperaba un cierre FnOnce<()>y encontró un i8 en su lugar. Esto se debe a que un cierre estándar implementa el rasgo público FnOnce<()>, mientras que nuestra función **do_something** simplemente devuelve

i8.

Cuando se implementa **FnOnce<()>**, el cierre solo se puede llamar una vez. Esto significa que cuando creamos un hilo, podemos asegurarnos de que el cierre solo se puede llamar una vez, y luego, cuando regresa, el hilo termina. Como nuestra función **do_something** es la línea final del cierre, se devuelve i8. Sin embargo, debe tenerse en cuenta que el hecho de que se implemente el rasgo **FnOnce<()>** no significa que no podamos llamarlo varias veces. Este rasgo solo se llama si el contexto lo requiere. Esto significa que si tuviéramos que llamar al cierre fuera del contexto del hilo, podríamos llamarlo varias veces.

Una vez que hemos separado los tres subprocesos, obtenemos una estructura **JoinHandle** de las funciones de generación de cada subproceso que creamos. Llamamos a la función de unión para cada uno de ellos. La función de unión espera a que finalice el subproceso asociado. Luego imprimimos el tiempo transcurrido. Las funciones de combinación devuelven una estructura de resultado, por lo que deben desenvolverse para acceder al valor de retorno del cierre pasado al hilo. Luego sumamos estos resultados para imprimir el resultado final. Si no llamamos a la función de unión, el proceso principal se ejecutará y finalizará antes de que finalicen los subprocesos. Este código de subprocesamiento da el siguiente resultado:

```
number 1 is running
number 2 is running
number 3 is running
time elapsed 2.000523086s
result 6
```

Como podemos ver, todo el proceso tardó poco más de 2 segundos en ejecutarse. Esto se debe a que los tres subprocesos se ejecutan simultáneamente. También podemos notar que el hilo tres se dispara antes que el hilo dos. No se preocupe si obtiene una secuencia de 1, 2, 3. Los hilos terminan en un orden indeterminado. La programación es determinista, sin embargo, hay miles de eventos que ocurren bajo el capó que requieren que la CPU haga algo. Como resultado, los intervalos de tiempo exactos que obtiene cada subproceso nunca son los mismos. Estos pequeños cambios se suman. Debido a esto, no podemos garantizar que los hilos terminen en un orden determinado.

La generación de subprocesos nos brinda una comprensión práctica de la programación asíncrona. Sin embargo, recordamos que el servidor web de Actix no utiliza esta sintaxis; define funciones con una sintaxis async y await. Para sentirnos más cómodos con el marco web, debemos analizar esta sintaxis.

2.4. Entendiendo async y await

La sintaxis async y await maneja los mismos conceptos cubiertos en la sección anterior, sin embargo, hay algunos matices. En lugar de simplemente generar hilos, creamos futuros y luego los manipulamos cuando sea necesario.

En informática, un futuro es un cálculo sin procesar. Aquí es donde el resultado aún no está disponible, pero cuando llamamos o esperamos, el futuro se completará con el resultado del cálculo. Los futuros también pueden denominarse promesas, retrasos o diferidos. Para explorar futuros, crearemos un nuevo proyecto Cargo y utilizaremos los futuros creados en el archivo **Cargo.toml**:

```
[dependencies]
futures = "0.3.21"
```

Ahora que tenemos nuestro futuro, podemos definir nuestra propia función asíncrona en el archivo **main.rs**:

```
async fn do_something(number: i8) -> i8 {
        println!("number␣{}␣is␣running", number);
        let two_seconds = time::Duration::new(2, 0);
        thread::sleep(two_seconds);
        return 2
}
```

Esta es la función de subproceso estándar que definimos anteriormente para señalar que el subproceso se está ejecutando y devuelve un valor del subproceso. La única diferencia es que tenemos una palabra clave asíncrona antes de la definición fn. La forma más sencilla de manejar esta función es llamarla y luego bloquear el programa hasta que finalice el cálculo:

```
use futures::executor::block_on;
use std::{thread, time};

fn main() {
        let now = time::Instant::now();
        let future_one = do_something(1);
        let outcome = block_on(future_one);
        println!("time␣elapsed␣{:?}", now.elapsed());
        println!("Here␣is␣the␣outcome:␣{}", outcome);
}
```

Aquí, la variable **future_one** es un futuro. Ejecutar esto nos da el siguiente resultado:

```
number 1 is running
time elapsed 2.000196102s
Here is the outcome: 2
```

Esto es de esperar ya que es lo mismo que el resultado del subproceso en la sección anterior. El resultado de la función asíncrona también se puede extraer usando await. Para hacer esto, necesitamos tener un bloque asíncrono:

```
// Agregar al programa main.rs
let future_two = async {
        return do_something(2).await
};
let future_two = block_on(future_two);
println!("Here␣is␣the␣outcome:␣{:?}", future_two);
```

Esto parece un poco detallado ya que está haciendo lo mismo que simplemente llamar a la función block_on en el futuro desde la llamada a la función **do_something**. Sin embargo, se puede llamar a **await** dentro de otra función asíncrona. También podemos ser más flexibles dentro del bloque asíncrono. Por ejemplo, podemos empaquetar dos futuros y devolverlos:

```
let future_two = async {
        let outcome_two = do_something(2).await;
        let outcome_three = do_something(3).await;
        return [outcome_two, outcome_three]
};
let future_two = block_on(future_two);
println!("Here␣is␣the␣outcome:␣{:?}", future_two);
```

Esto da la siguiente salida:

```
number 2 is running
number 3 is running
Here is the outcome: [2, 2]
```

Sin embargo, si lo cronometramos, podemos ver que las funciones se activan de forma secuencial, lo que lleva poco más de 4 segundos. Esto no es muy útil, ya que es mejor que no nos molestemos con la sintaxis asíncrona si vamos a obtener los mismos resultados de tiempo que la programación secuencial normal.

Unirse parecía funcionar en la sección anterior. Dado que los futuros también tienen una función de combinación, tiene sentido utilizarla para dividir el tiempo necesario a la mitad haciendo que dos futuros se ejecuten al mismo tiempo:

```
use futures::join;
...
let future_two = block_on(future_two);
println!("Here␣is␣the␣outcome:␣{:?}", future_two);
let second_outcome = async {
        let future_two = do_something(2);
        let future_three = do_something(3);
        return join!(future_two, future_three)
```

```
};
let now = time::Instant::now();
let result = block_on(second_outcome);
println!("time␣elapsed␣{:?}", now.elapsed());
println!("here␣is␣the␣result:␣{:?}", result);
```

Sin embargo, esto no nos da el resultado que esperábamos:

```
number 1 is running
time elapsed 2.000109916s
Here is the outcome: 2
number 2 is running
number 3 is running
Here is the outcome: [2, 2]
```

Los futuros se están ejecutando secuencialmente. La única diferencia es que devolvemos el resultado de los futuros en una tupla. A pesar de que la función **futures::join** es contraria a la intuición, podemos usar otra caja para crear nuestra propia función de unión asíncrona usando la caja **async_std**. Antes de hacer esto, podemos definir la caja en la sección de dependencias del archivo **Cargo.toml**:

```
async-std = "1.10.0"
```

Con esta caja, ahora podemos ejecutar nuestros futuros de forma asíncrona:

```
use std::vec::Vec;
use async_std;
use futures::future::join_all;
...
        let mut futures_vec = Vec::new();
let future_four = do_something(4);
let future_five = do_something(5);
futures_vec.push(future_four);
futures_vec.push(future_five);
let handles = futures_vec.into_iter().map(async_std::task::spawn).
    collect::<Vec<_>>();
let results = join_all(handles).await;
return results
};
let now = time::Instant::now();
let result = block_on(third_outcome);
println!("time␣elapsed␣for␣join␣vec␣{:?}", now.elapsed());
println!("Here␣is␣the␣result:␣{:?}", result);
```

Aquí, lo que hemos hecho es definir nuestros futuros en un bloque asíncrono y luego agregarlos a un vector denominado **futures_vec**. Luego obtenemos nuestro vector poblado de futuros y llamamos a la función **into_iter** en él. Esto devuelve un iterador, que podemos usar para recorrer los futuros.

También podemos devolver un iterador simplemente llamando a la función iter. Sin embargo, llamar a esto producirá **&T**. Simplemente hacer referencia a un futuro no es un futuro. Necesitamos el directorio futuro si vamos a aplicarle la función de generación. Nuestra función **into_iter** nos permite obtener **T**, **&T** o **&mut T**, según el contexto y las necesidades.

Luego aplicamos la función **async_std::task::spawn** a cada futuro en el vector usando la función de mapa. La función **async_std::task::spawn** parece familiar para la función **thread::spawn** que usamos anteriormente, así que, nuevamente, ¿por qué molestarse con todo este dolor de cabeza adicional? Podríamos simplemente recorrer el vector y generar un hilo para cada tarea. La diferencia aquí es que la función **async_std::task::spawn** genera una tarea asíncrona en el mismo hilo. Por lo tanto, ¡estamos ejecutando simultáneamente ambos futuros en el mismo hilo!

Luego usamos la función de recopilación para recopilar los resultados de este mapeo en un vector llamado identificadores. Una vez hecho esto, pasamos este vector a la función **join_all** para unir todas las tareas asíncronas y esperar a que se completen usando **await**. Con esto, obtenemos la siguiente salida:

```
number 1 is running
time elapsed 2.000157327s
Here is the outcome: 2
number 2 is running
number 3 is running
Here is the outcome: [2, 2]
number 2 is running
number 3 is running
time elapsed 4.000374497s
here is the result: (2, 2)
number 4 is running
number 5 is running
time elapsed for join vec 2.005101774s
Here is the result: [2, 2]
```

¡Hemos logrado ejecutar dos tareas asíncronas al mismo tiempo en el mismo hilo, lo que resultó en que ambos futuros se ejecutaran en poco más de 2 segundos!

Como podemos ver, generar hilos y tareas asíncronas en Rust es bastante sencillo. Sin embargo, debemos tener en cuenta que pasar variables a subprocesos y tareas asíncronas no es sencillo. El mecanismo de préstamo de Rust garantiza la seguridad de la memoria. Tenemos que pasar por pasos adicionales al pasar datos a un hilo. Una mayor discusión sobre los conceptos generales detrás del intercambio de datos entre subprocesos no es propicia para nuestro proyecto web. Sin embargo, podemos indicar brevemente qué tipos permiten esto:

```
std::sync::Arc: Este tipo permite que los subprocesos hagan
```

```
   referencia a datos externos:

use std::sync::Arc;
use std::thread;

let names = Arc::new(vec!["dave", "chloe", "simon"]);
let reference_data = Arc::clone(&names);
let new_thread = thread::spawn(move || {
        println!("{}", reference_data[1]);
});

std::sync::Mutex: Este tipo permite a los subprocesos mutar datos
   externos:

use std::sync::Mutex;
use std::thread;

let count = Mutex::new(0);
let new_thread = thread::spawn(move || {
        *count.lock().unwrap() += 1;
});
```

Dentro del hilo aquí, eliminamos la referencia del resultado del bloqueo, lo desenvolvemos y lo mutamos. Debe tenerse en cuenta que solo se puede acceder al estado compartido una vez que se mantiene el bloqueo.

Mirando hacia atrás en el servidor web básico que construimos en la sección anterior, hemos explorado casi todos los nuevos conceptos que se introdujeron. El último concepto que debe entenderse es la definición de la función principal. Podemos ver que la función principal es una función asíncrona. Sin embargo, si simplemente tratamos de definir la función principal como una función asíncrona, devolverá un futuro en lugar de ejecutar el programa.

Esto es posible gracias a la macro #**[actix_rt::main]** proporcionada por la caja **atix-rt**. Esta es una implementación de tiempo de ejecución y permite que todo se ejecute en el subproceso actual. La macro #**[actix_rt::main]** marca la función asíncrona (que en este caso es la función principal) para que la ejecute el sistema **Actix**.

A riesgo de meterse en problemas aquí, la caja Actix ejecuta un cálculo concurrente basado en el modelo actor. Aquí es donde un actor es un cómputo. Los actores pueden enviar y recibir mensajes entre ellos. Los actores pueden alterar su propio estado, pero solo pueden afectar a otros actores a través de mensajes, lo que elimina la necesidad de una sincronización basada en bloqueos (el mutex que cubrimos está basado en bloqueos). Una mayor exploración de este modelo no nos ayudará a desarrollar

aplicaciones web. Sin embargo, la caja Actix tiene buena documentación sobre la codificación de sistemas concurrentes con Actix en **https://actix.rs/book/actix/sec-0-quick-start.html**.

Hemos cubierto mucho aquí. No te sientas estresado si no sientes que lo has retenido todo. Hemos cubierto brevemente una variedad de temas relacionados con la programación asíncrona. No necesitamos entenderlo de adentro hacia afuera para comenzar a construir aplicaciones basadas en el **framework Actix Web**.

Sin embargo, este recorrido rápido es invaluable cuando se trata de depurar y diseñar aplicaciones. Para ver un ejemplo en la naturaleza, podemos ver esta solución inteligente de desbordamiento de pila para ejecutar varios servidores en un solo archivo: **https://stackoverflow.com/questions/59642576/run-multiple-actix-app-on- different-ports**.

Las vistas básicas se definen para cada servidor. Como podemos ver, incluso si son dos servidores diferentes, no se necesita notación adicional:

```
use actix_web::{web, App, HttpServer, Responder};
use futures::future;
async fn utils_one() -> impl Responder {
        "Utils␣one␣reached\n"
}
async fn health() -> impl Responder {
        "All␣good\n"
}
```

Una vez que se definen las vistas, los dos servidores se definen en la función principal:

```

async fn main() -> std::io::Result<()> {
        let s1 = HttpServer::new(move || {
                App::new().service(web::scope("/utils").route(
                "/one", web::get().to(utils_one)))
        })
        .bind("0.0.0.0:3006")?
        .run();
        let s2 = HttpServer::new(move || {
                App::new().service(web::resource(
                "/health").route(web::get().to(health)))
        })
        .bind("0.0.0.0:8080")?
        .run();
        future::try_join(s1, s2).await?;
        Ok(())
}
```

Podemos deducir con confianza que s1 y s2 son futuros que devuelve la función de ejecución. Luego unimos estos dos futuros y esperamos a que terminen. Con este ejemplo de la vida real, podemos ver que nuestra comprensión de la programación asíncrona nos permite desglosar lo que sucede cuando se construyen y ejecutan varios servidores.

Con nuestra nueva confianza en la programación asíncrona, no hay nada que nos impida crear 20 futuros y meterlos en un **try_join**. macro para ejecutar 20 servidores, aunque esto no se recomienda, ya que aumentar la cantidad de servidores aumentará el uso de recursos con rendimientos decrecientes en el rendimiento.

Ahora que nos sentimos realmente cómodos con el marco **Actix Web**, prácticamente podemos considerar la creación de una aplicación escalable con él. Comenzaremos administrando las vistas de nuestra aplicación.

2.5. Gestión de vistas mediante el marco Actix Web

En la sección Lanzamiento de un servidor **web Actix básico**, definimos nuestras vistas en el archivo **main.rs**. Sin embargo, esto no es escalable. Si continuamos definiendo todas nuestras rutas en **main**, terminaremos con muchas importaciones y definiciones de ruta en un archivo. Esto hace que sea difícil de navegar y administrar. Si queremos cambiar un prefijo de **URL** para un bloque de vistas, la edición en este contexto es propensa a errores. Lo mismo ocurre con la desactivación de un bloque de vistas.

Para administrar nuestras vistas, necesitamos crear nuestros propios módulos para cada conjunto de vistas. Para administrar nuestras vistas, creamos un nuevo proyecto de Cargo llamado **administrando_vistas**. Luego definimos la siguiente estructura del proyecto, ver figura 2.1.

El archivo **main.rs** alberga nuestra definición del servidor. Luego definimos una estructura auxiliar de ruta de URL para todas nuestras vistas en el archivo **path.rs**. Definimos nuestras vistas de inicio y cierre de sesión en los archivos **login.rs** y **logout.rs**. Luego construimos las rutas a través de una función de fábrica en el archivo **vistas/auth/mod.rs**. Luego organizamos el disparo de la fábrica en una función de fábrica en el archivo **vistas/mod.rs**.

Ahora que nuestra estructura ha sido definida, podemos construir nuestro archivo **main.rs**:

```
1 use actix_web::{App, HttpServer};
2 mod vistas;
3 #[actix_rt::main]
4 async fn main() -> std::io::Result<()> {
```

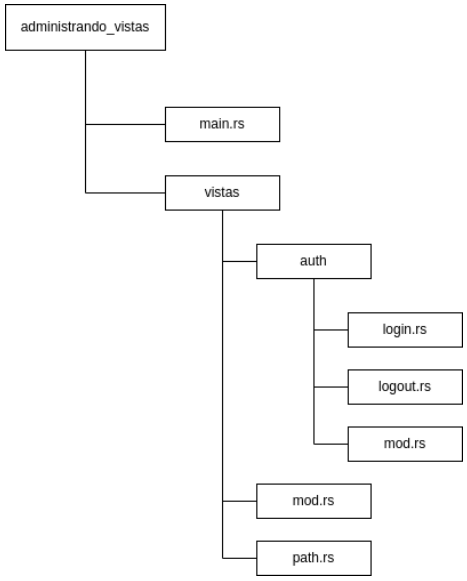

Figura 2.1: Estructura básica de programa

```
5         HttpServer::new(|| {
6                 let app = App::new();
7                 return app
8         })
9         .bind("127.0.0.1:8000")?
10        .run()
11        .await
12 }
```

Aquí, importamos nuestro módulo de vistas vacío, definimos nuestra aplicación en el cierre y la ejecutamos. Ejecutarlo ahora mismo nos dará un servidor sin vistas, por lo que podemos comenzar a trabajar en el módulo de vistas.

El primer archivo en el que podemos trabajar es el archivo vistas/path.rs, que alberga una estructura que define una ruta. Comenzamos aquí porque esta estructura no tiene dependencias. Esta estructura se utilizará para definir un prefijo estándar para una URL, que se fusiona con una cadena que se pasa a la función de definición para crear una cadena de URL:

```
1 pub struct Path {
2         pub prefijo: String
3 }
4 impl Path {
5         pub fn definir(&self, siguiendo_path: String) -> String {
6                 return self.prefijo.to_owned() + &siguiendo_path
```

```
7         }
8 }
```

En nuestra función de definición, tomamos la referencia de la estructura como un parámetro &self para que la misma instancia de estructura se pueda usar varias veces para definir varias URL con el mismo prefijo. Cabe señalar que las funciones con tales firmas se parecen a los métodos, lo que significa que se utilizan como métodos. También debe tenerse en cuenta que usamos una función **to_owned** en la referencia al prefijo de la instancia de estructura. La función **to_owned** crea datos propios a partir de datos prestados mediante la clonación. Queremos que nuestra función de definición devuelva una **URL** de cadena que se pueda pasar a otras funciones, sin embargo, también queremos que nuestra estructura **Path** conserve el atributo de prefijo para que pueda usarse nuevamente para otras vistas.

Ahora que hemos definido nuestra ruta, podemos pasar a nuestras vistas básicas de inicio y cierre de sesión. Nos acercamos a esto a continuación porque las vistas tampoco tienen dependencias. Teniendo en cuenta que este capítulo se centra en la gestión de vistas en lugar de iniciar y cerrar sesión, estas vistas simplemente devolverán una cadena. El procesamiento de datos en vistas se trata en el siguiente capítulo. La autenticación se trata en Gestión de sesiones de usuario.

En nuestro archivo **vistas/auth/login.rs**, definimos la siguiente vista de inicio de sesión:

```
1 pub async fn login() -> String {
2         format!("Vista Login")
3 }
```

Aquí, tenemos una función asíncrona estándar que devuelve una cadena. También podemos definir nuestra vista de cierre de sesión en el archivo **vistas/auth/-logout.rs** de la misma manera:

```
1 pub async fn logout() -> String {
2         format!("Vista␣Logout")
3 }
```

Ahora que hemos definido nuestras vistas, necesitamos crearlas en una función de fábrica en el archivo **vistas/auth/mod.rs**:

```
1 use actix_web::web;
2 mod login;
3 mod logout;
4 use super::path::Path;
5 pub fn fabrica_auth(app: &mut web::ServiceConfig) {
6         let path_base: Path = Path{prefijo: String::from("/auth")};
7         app.route(&path_base.definir(String::from("/entrar")),
8         web::get().to(login::login))
```

```
9           .route(&path_base.definir(String::from("/salir")),
10          web::get().to(logout::logout));
11  }
```

En las importaciones, podemos notar que obtenemos la estructura **Path** del directorio principal del directorio **auth** usando **super**. Antes, hemos estado usando **super** para ingresar al archivo **mod.rs** en el mismo directorio. Sin embargo, si usamos **super** en un archivo **mod.rs**, entonces importamos archivos en el archivo **mod.rs** del directorio principal. Si quisiéramos, podríamos importar la estructura **Path** al archivo **vistas/auth/login.rs** usando **use super::super::path;**.

También podemos ver que una vez que hemos importado todo lo que necesitamos, definimos una función de fábrica que no devuelve nada y toma la aplicación para definir rutas en ella. Sin embargo, en lugar de pasar **actix_web::App**, pasamos una estructura **actix_web::web::ServiceConfig**. Incluso si intentamos pasar la estructura **actix_web::App**, **actix_web** no nos lo permitirá.

Una de las estructuras necesarias para definir el tipo de la función para pasar la aplicación es privada. La estructura **actix_web::web::ServiceConfig** nos permite configurar más la aplicación. Usamos esto para definir rutas, sin embargo, podemos establecer datos de aplicaciones, registrar un servicio HTTP o registrar un recurso externo para recursos de generación de URL usando esta estructura.

Una vez que hemos pasado la estructura de configuración, definimos las rutas como lo haríamos si estuviéramos definiendo una ruta en el archivo **main.rs** donde está la definición del servidor. También podemos ver cómo se usa la estructura **Path**. Hay una ligera ventaja en el uso de la estructura **Path**. El prefijo de **URL** se define una vez, lo que reduce la posibilidad de que ocurra algún error tipográfico en el prefijo si estamos definiendo muchas rutas. También facilita el mantenimiento. Si vamos a cambiar un prefijo para un conjunto de vistas, solo tenemos que cambiarlo una vez en la construcción de la estructura **Path** en la fábrica.

Ahora que tenemos nuestro módulo de **vistas/autenticación** completamente operativo, simplemente podemos pasar la estructura de configuración a través de la fábrica para construir todas las rutas para la autenticación. En el futuro, también construiremos otros módulos para vistas. Debido a esto, necesitamos otra fábrica que pueda orquestar las fábricas de vistas múltiples. Esto se puede definir en el archivo **vistas/mod.rs**:

```
1 use actix_web::web;
2 mod path;
3 mod auth;
4 pub fn fabrica_vistas(app: &mut web::ServiceConfig) {
5         auth::fabrica_auth(app);
6 }
```

Como podemos ver, importamos el módulo de autenticación y lo usamos para construir nuestras vistas en la fábrica. Observamos que pasamos la estructura de configuración a la fábrica y luego la pasamos a **fabrica_auth**. El parámetro de la aplicación se puede pasar a varias fábricas una tras otra, ya que es una referencia y las fábricas se llaman en orden secuencial. También importamos el archivo de ruta. Si bien esto no se usa en el archivo, lo necesitamos para la súper llamada en la fábrica de autenticación. Es por eso que no recibimos una advertencia cuando ejecutamos el código.

Ahora tenemos una forma escalable y bien estructurada de administrar nuestras vistas, todo lo que tenemos que hacer es importar esta fábrica de vistas y llamarla en el archivo **main.rs**:

```
use actix_web::{App, HttpServer};
mod vistas;
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
        HttpServer::new(|| {
                let app = App::new().configure(vistas::vistas
                    fabrica)
                return app
        })
        .bind("127.0.0.1:8000")?
        .run()
        .await
}
```

Como podemos ver, esto es más o menos lo mismo. Todo lo que tenemos que hacer es llamar a la función de configuración en la estructura de la aplicación. Luego pasamos la fábrica de vistas a la función de configuración, que pasará la estructura de configuración a nuestra función de fábrica por nosotros. Como la función de configuración devuelve Self, es decir, la estructura de la aplicación, podemos devolver el resultado al final del cierre.

¡Ahora tenemos un servidor en funcionamiento que crea vistas de forma escalable! Simplemente podemos cortar todas nuestras vistas de autenticación simplemente comentando la siguiente línea en el archivo **vistas/mod.rs**:

```
auth::fabrica_auth(app);
```

Esto también nos da mucha flexibilidad. Al definir nuestras propias fábricas para cada módulo de vistas, no hay nada que nos impida agregar parámetros adicionales a las fábricas individuales para personalizar la compilación. Por ejemplo, si por alguna razón queremos deshabilitar la función de cierre de sesión en función de una variable de entorno o configuración, simplemente podemos agregar un condicional en nuestra

fábrica en el archivo **vistas/auth/mod.rs**:

```
use actix_web::web;
mod login;
mod logout;
use super::path::Path;
pub fn fabrica_auth(app: &mut web::ServiceConfig, logout: bool) {
        let path_base: Path = Path{prefijo: String::from("/auth")};
        let app = app.route(&path_base.definir(String::from("/entrar
           ")),
        web::get().to(login::login))
        if logout {
                app.route(&path_base.definir(String::from("/salir"))
                   ,
                web::get().to(logout::logout));
        }
}
```

Todo lo que tenemos que agregar es un parámetro de cierre de sesión. Luego asignamos el resultado de la función de ruta a la variable de la aplicación. Luego llamamos a la función de ruta en esa variable si la variable de cierre de sesión es verdadera. Recuerda, no tenemos que devolver nada; solo necesitamos llamar a las funciones.

Es importante mantener la lógica aislada. La lógica para construir las vistas para el módulo de autenticación permanece en la fábrica de autenticación. Sin embargo, la colección de variables alrededor de la configuración debe definirse en **vistas/mod.rs**:

```
use actix_web::web;
mod path;
mod auth;
use std::env;
pub fn fabrica_vistas(app: &mut web::ServiceConfig) {
        let args: Vec<String> = env::args().collect();
        let param: &String = &args[args.len()-1];
        if param.as_str() == "cancel_logout" {
                println!("Vista␣logout␣no␣fue␣configurado");
        } else {
                println!("vista␣logout␣esta␣siendo␣configurada");
                auth::fabrica_auth(app, true);
        }
}
```

Aquí, recopilamos los parámetros del entorno. Si es **cancel_logout**, la vista de cierre de sesión no se configurará. Mantener la lógica de los parámetros en la fábrica de **vistas/mod.rs** aumenta la flexibilidad al permitirnos configurar varias fábricas con un solo parámetro. También podemos revisar la estructura **Path** y la ventaja

que ofrece aquí. Si tuviéramos que cambiar el prefijo de un conjunto de vistas sobre la marcha, solo necesitaríamos una declaración condicional o de coincidencia para la estructura Path al comienzo de la función de fábrica en lugar de cada función de definición de ruta.

Nuestro objetivo era crear una aplicación básica que sirva y gestione las vistas de forma escalable. Aquí, tenemos una aplicación que sirve vistas. Estas vistas pueden entrar y salir y definirse en sus propios módulos. Para ejecutar sin la vista de cierre de sesión, usamos la siguiente línea de comando:

```
cargo run cancel_logout
```

Si no, entonces solo ejecutamos la ejecución de carga. Si ejecutamos nuestra aplicación con la vista de cierre de sesión configurada, tenemos las siguientes URL y salidas:

```
http:
```

Esto da la siguiente cadena:

```
Login view
```

También da esto:

```
http:/ /127.0.0.1:8000/auth/salir
```

Nosotros obtenemos la siguiente cadena:

```
Logout view
```

2.6. Poniendo todo junto

Hemos cubierto mucho para obtener algunas vistas básicas en funcionamiento en un servidor web Actix. Podríamos haber hecho todo esto en una página:

```
use actix_web::{web, App, HttpRequest, HttpServer, Responder};
pub async fn logout() -> String {
        format!("Vista␣de␣Salida")
}
pub async fn login() -> String {
        format!("Vista␣de␣entrada")
}
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
        HttpServer::new(|| {
                let app = App::new()
                .route("/auth/entrar", web::get().to(login))
```

```
13                  .route("/auth/salir", web::get().to(logout));
14                  return app
15          })
16          .bind("127.0.0.1:8000")?
17          .run()
18          .await
19  }
```

Capítulo 3

Procesando Solicitudes HTTP

Hasta este momento, hemos utilizado el marco web Actix para proporcionar vistas básicas. Sin embargo, esto solo puede llevarnos hasta cierto punto cuando se trata de extraer datos de la solicitud y devolverlos al usuario.

3.1. Conociendo la configuración inicial

En la figura 3.1 se observa la estructura de la tienda.

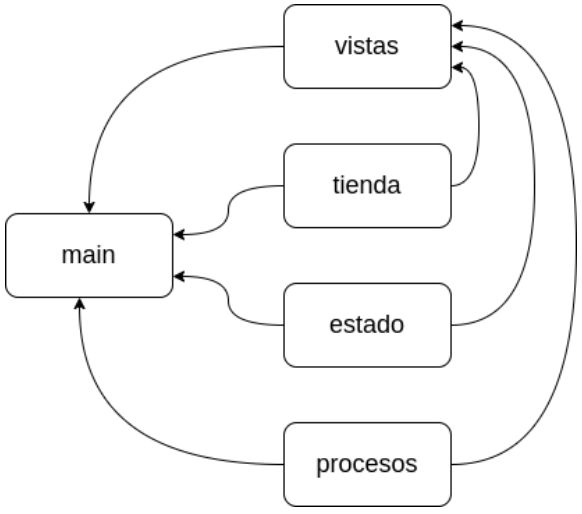

Figura 3.1: Estructura básica de la tienda

Aquí, registraremos todos los módulos en el archivo principal y luego colocaremos

todos estos módulos en las vistas que se utilizarán. Básicamente, estamos intercambiando la interfaz de línea de comandos, Diseño de su aplicación web en Rust, con vistas web. La combinación de estos módulos nos da los siguientes archivos en el código base:

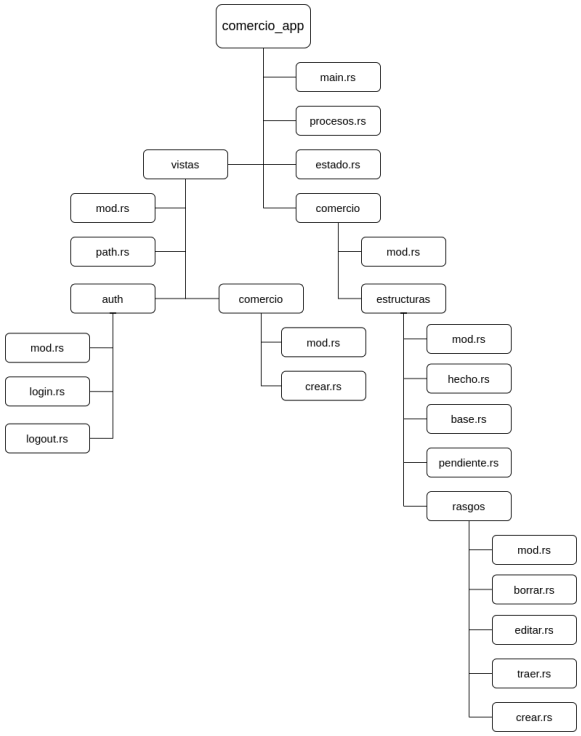

Figura 3.2: Estructura básica de los archivos de la tienda

Tenga en cuenta que aunque estamos agregando módulos, hay un archivo que será nuevo para nosotros, y este es **main.rs**. Aquí, tenemos un código cruzado que estamos empalmando:

```
1  use actix_web::{App, HttpServer};
2  mod estado;
3  mod procesos;
4  mod comercio;
5  mod vistas;
6
7
8  #[actix_rt::main]
9  async fn main() -> std::io::Result<()> {
10         HttpServer::new(|| {
```

```
11                  let app = App::new().configure(
12                  vistas::fabrica_vistas);
13                  return app})
14          .bind("127.0.0.1:8000")?
15          .run()
16          .await
17 }
```

Aquí, definimos los módulos y luego definimos nuestro servidor. Debido a que el servidor utiliza **fabrica_vistas**, no tendremos que modificar este archivo durante el resto de este capítulo. En su lugar, encadenaremos las funciones de fábrica que se llamarán en la función **fabrica_vistas**.

En este punto, podemos sentarnos y apreciar los dividendos de todo el arduo trabajo que hicimos en los capítulos anteriores. El aislamiento de principios y módulos bien definidos nos permitió insertar nuestra lógica desde nuestro programa de línea de comandos en nuestra interfaz de servidor con un esfuerzo mínimo. Ahora, todo lo que tenemos que hacer es conectarlo a nuestro módulo de vistas y luego pasar parámetros a esas vistas.

3.2. Pasando parámetros

Ahora que estamos familiarizados con las vistas básicas, vamos a pasar parámetros y datos a la vista. Nuestro módulo de vistas de comercio tendrá la estructura que se muestra en la rama izquierda de la figura 3.2.

Para demostrar esto, vamos a construir una vista básica que toma un parámetro de la **URL** y crea un elemento de tarea. Para ello, tendremos que hacer lo siguiente:

- Cargue el estado actual de la lista de personas pendientes.
- Obtenga el título del nuevo elemento pendiente de la URL.
- Pase el título y la cadena pendiente a través de **fabrica_comercio**.
- Pase el resultado del paso anterior, junto con la cadena de creación y el estado, a la interfaz del módulo de proceso.
- Devuelve una cadena al usuario para indicar que el proceso ha finalizado.

Debido a que el proceso anterior consiste principalmente en interactuar con interfaces de módulos cuidadosamente empaquetadas, todo esto se puede lograr con esta función simple, que se puede encontrar en el archivo **vistas/comercio/crear.rs**:

```
use serde_json::value::Value;
use serde_json::Map;
use actix_web::HttpRequest;
use crate::comercio;
use crate::estado::leer_archivo;
use crate::procesos::entrada_proceso;
pub async fn create(req: HttpRequest) -> String {
        let estado: Map<String, Value> = read_file(String::from(
        "./estado.json"));
        let titulo: String = req.match_info().get("titulo"
        ).unwrap().to_string();
        let titulo_referencia: String = titulo.clone();
        let item = comercio::fabrica_comercio(&String::from("
           pendiente"),
        titulo).expect("crear ");
        entrada_proceso(item, "crear".to_string(), &estado);
        return format!("{} crear", titulo_referencia)
}
```

Este código demuestra que la lógica dentro de nuestras vistas futuras consistirá principalmente en reorganizar estas interfaces en un orden que tenga sentido para el propósito de la vista. Para que esta vista esté disponible para el servidor, tendremos que empaquetarla como una función de fábrica directa en el archivo **vistas/comercio/mod.rs**:

```
use actix_web::web;
mod crear;
use super::path::Path;

pub fn fabrica_item(app: &mut web::ServiceConfig) {
        let path_base: Path = Path{prefijo: String::from("/item")};
        app.route(&path_base.definir(String::from("/crear/{titulo}")
           ),
        web::get().to(crear::crear));
}
```

Aquí, podemos ver que nuestra fábrica adopta el mismo enfoque que la fábrica de vistas de autenticación, utilizando las estructuras **Path** y **ServiceConfig**. También podemos ver que nuestro parámetro de título está definido con corchetes, **título**, que se extrae a través de la estructura **HttpRequest** en nuestra vista de creación mediante la función **match_info().get("título")**. Ahora, en nuestro archivo **src/vistas/mod.rs**, necesitamos limpiar parte de la lógica anterior e introducir nuestro **fabrica_item**:

```
use actix_web::web;
mod path;
```

```
3 mod auth;
4 mod comercio;
5
6 pub fn views_factory(app: &mut web::ServiceConfig) {
7         auth::fabrica_auth(app);
8         comercio::fabrica_item(app);
9 }
```

También hemos eliminado el parámetro de definición de cierre de sesión para que se pueda compilar. También tendremos que limpiar nuestra **fabrica_auth** en el archivo **vistas/auth/mod.rs**:

```
1 use actix_web::web;
2 mod login;
3 mod logout;
4 use super::path::Path;
5
6 pub fn fabrica_auth(app: &mut web::ServiceConfig) {
7         let base_path: Path = Path{prefix: String::from("/auth")};
8         let app = app.route(&base_path.define(String::from("/login")
              ),
9         web::get().to(login::login))
10        .route(&base_path.define(String::from("/logout")),
11        web::get().to(logout::logout));
12 }
```

Ahora, nuestra aplicación es completamente funcional y podemos interactuar con ella usando el comando de ejecución de carga. http://127.0.0.1:8000/item/crear/codigp%20in%20rust nos da el siguiente resultado en una ventana del navegador web:

```
1 codigo en rust creado
```

Además de esto, nuestro archivo **estado.json** contiene el siguiente contenido:

```
1 {"codigo␣en␣rust":"pendiente"}
```

¡Funcionó! Ahora tenemos un servidor que acepta una solicitud **GET**, extrae parámetros de la **URL**, crea un nuevo elemento pendiente y luego lo almacena en nuestro archivo JSON. Cabe señalar que, si bien vamos a utilizar un archivo JSON para el almacenamiento de datos, definiremos una base de datos para la aplicación, Persistencia de datos con **PostgreSQL**. Además, tenga en cuenta que un %20 en la **URL** indica un espacio.

El método **GET** funciona para nosotros, pero no es el método más apropiado para crear una tarea pendiente. Los métodos **GET** se pueden almacenar en caché, marcar, guardar en el historial del navegador y tener restricciones en cuanto a su longitud. Agregarlos a favoritos, almacenarlos en el historial del navegador o almacenarlos en

caché no solo presenta problemas de seguridad, sino que también aumenta el riesgo de que el usuario vuelva a hacer la misma llamada accidentalmente. Debido a esto, no es una buena idea alterar los datos con una solicitud **GET**. Para protegernos contra esto, podemos usar solicitudes **POST**, que no se almacenan en caché, terminan en el historial del navegador y no se pueden marcar.

Nuestra función de creación se puede convertir en un método **POST** cambiando la función de obtención a una función de publicación en nuestro módulo de vistas pendientes en el archivo **vistas/comercio/mod.rs**:

```
1 use actix_web::web;
2 mod crear;
3 use super::path::Path;
4
5 pub fn fabrica_item(app: &mut web::ServiceConfig) {
6         let path_base: Path = Path{prefijo: String::from("/item")};
7         app.route(&path_base.definir(String::from("/crear/{titulo}")
              ),
8         web::post().to(crear::crear));
9 }
```

El cambio está en la última línea de la función **fabrica_item**. Si ejecutamos esto nuevamente, nuestra URL que creó un elemento pendiente ya no funciona en el navegador. En su lugar, recibimos un error 404 porque no se puede encontrar la página. Esto tiene sentido ya que la URL del navegador es una solicitud GET. Podemos realizar una función POST usando Postman (ver figura 3.3):

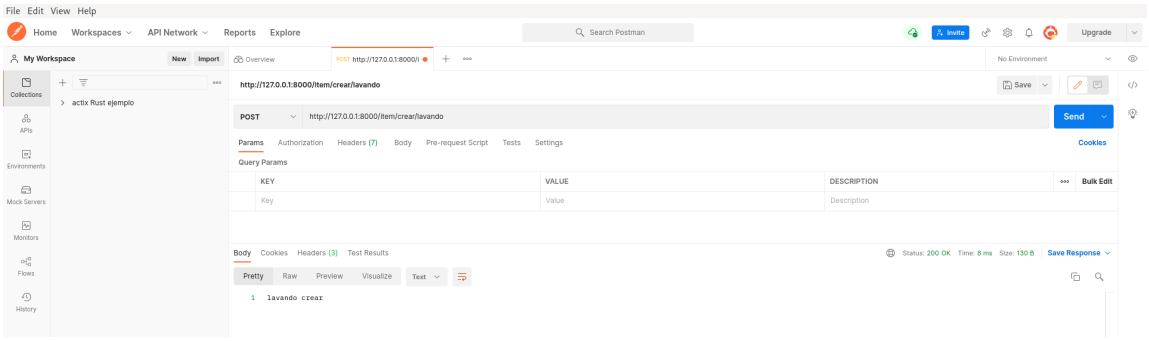

Figura 3.3: Revisando POST con postman

Aquí, podemos ver que creamos un elemento de lavado para hacer con el mismo patrón de URL. Podemos ver que obtenemos un código 200 y luego una cadena creada por lavado en el cuerpo. La verificación de nuestro archivo de estado JSON nos muestra que el sistema aún funciona, ya que tenemos el código oxidado y lavado

para hacer elementos, ambos pendientes. Podemos hacer que el mismo patrón de URL acepte los métodos POST y GET, simplemente llamando a la función de ruta dos veces en la fábrica de vistas pendientes con las funciones de publicación y obtención:

```
use actix_web::web;
mod crear;
use super::path::Path;

pub fn fabrica_item(app: &mut web::ServiceConfig) {
        let path_base: Path = Path{prefijo: String::from("/item")};
        app.route(&path_base.definir(String::from("/crear/{titulo}")
           ),
        web::post().to(crear::crear));
        app.route(&path_base.definir(String::from("/crear/{titulo}")
           ),
        web::get().to(crear::crear));
}
```

Teniendo en cuenta las diferencias que cubrimos anteriormente entre los métodos GET y POST, es sensato tener solo un método POST para nuestra función de creación.

Mirando hacia atrás en nuestra **GUI** de **Postman**, tenemos que pensar en el futuro. Con nuestra función de creación, una línea de texto es suficiente para decirnos que se ha creado el elemento. De hecho, ni siquiera tenemos que devolver nada en el cuerpo; el número de estado de devolución es suficiente para decirnos que el artículo ha sido creado. Sin embargo, cuando se trata de obtener una lista de tareas pendientes, necesitaremos datos estructurados. Para lograr esto, tendremos que serializar los datos **JSON** y devolvérnoslos.

3.3. Uso de macros para la serialización JSON

La serialización **JSON** se admite directamente a través de la crate **Actix-web**. Podemos demostrar esto creando una vista **GET** que devuelve todos nuestros elementos pendientes en el archivo **vistas/comercio/traer.rs**:

```
use actix_web::{web, Responder};
use serde_json::value::Value;
use serde_json::Map;
use crate::estado::leer_archivo;
pub async fn traer() -> impl Responder{
        let estado: Map<String, Value> = leer_archivo(String::from("
           ./estado.json"));
        return web::Json(estado);
```

```
}
```

Aquí, simplemente leemos nuestro archivo JSON y lo devolvemos, lo pasamos a la estructura web::Json y luego lo devolvemos. La estructura web::Json implementa el rasgo Responder. Tenemos que definir esta nueva vista agregando la definición del módulo al archivo **vistas/comercial/mod.rs**, y luego agregar la definición de ruta a la función de fábrica:

```
mod traer
...
app.route(&path_base.definir(String::from("/traer")),
        web::get().to(traer::traer));
```

Ejecutar **http://127.0.0.1:8000/item/traer** nos brinda los siguientes datos JSON en el cuerpo de la respuesta:

```
{"code in rust":"pending","washing":"pending"}
```

Si bien esto hace el trabajo, no es flexible. Es posible que necesitemos dos listas diferentes: una para los elementos terminados y otra para los pendientes. También podrían querer un recuento de la cantidad de elementos y datos estructurados. Por ejemplo, es posible que necesitemos agregar una marca de tiempo para cuando se creó o terminó el elemento. Tener un cuerpo **JSON** simple para el elemento como título y tener el estado como valor no nos permite escalar la complejidad cuando sea necesario.

Para tener más control sobre el tipo de datos que vamos a devolver al usuario, vamos a tener que construir nuestras propias estructuras de serialización. Nuestra estructura de serialización presentará dos listas: una para artículos completados y otra para artículos pendientes. Estas listas se completarán con objetos que consisten en un título y un estado.

Como recordará nuestras estructuras de elementos **pendientes** y **Listos** se heredan a través de la composición de una estructura Base. Por lo tanto, tenemos que acceder al título y al estado desde la estructura **Base**. Sin embargo, nuestra estructura **Base** no es accesible al público. Tendremos que hacerlo accesible para que podamos serializar los atributos de cada elemento pendiente, como se muestra en la figura 3.4.

Esto se puede hacer cambiando la declaración del módulo base en el archivo **comercio/estructuras/mod.rs** de **mod base;** a la base **mod pub;**. Ahora que la estructura **Base** está directamente disponible fuera del módulo, podemos construir nuestro propio módulo **json_serialización** en el directorio **src** con la estructura de la figura 3.5.

Nuestro nuevo módulo debe tener las siguientes dependencias agregadas al archivo **Cargo.toml**:

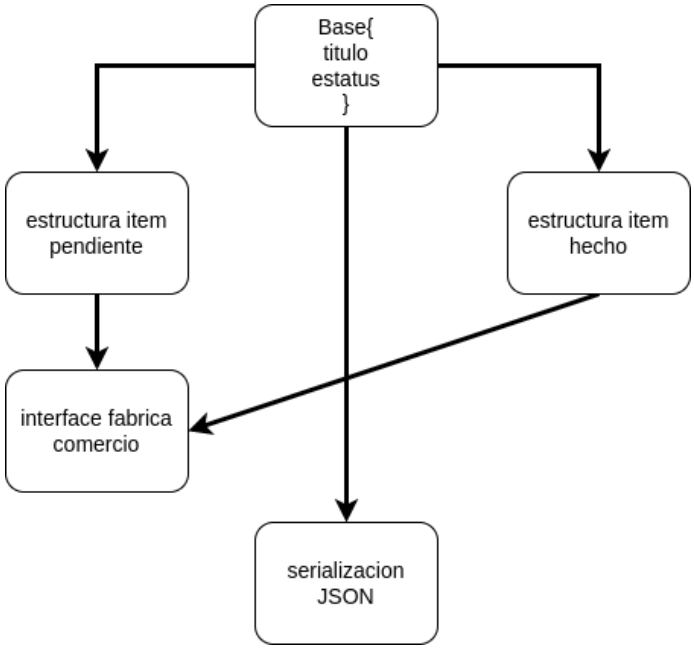

Figura 3.4: Relación que tienen nuestras estructuras pendientes con nuestras interfaces

```
futures = "0.3.21"
serde = "1.0.136"
```

Ahora que tenemos todo, podemos definir un esquema JSON en nuestro archivo **src/json_serializacion/comercio_items.rs** y definir los parámetros y tipos para un cuerpo **JSON**:

```
use serde::Serialize;
use crate::comercio::TiposItem;
use crate::comercio::estructuras::base::Base;
#[derive(Serialize)]
pub struct ComercioItems {
        pub items_pendiente: Vec<Base>,
        pub items_hecho: Vec<Base>,
        pub contador_item_pendiente: i8,
        pub contador_item_hecho: i8
}
```

Aquí, todo lo que hemos hecho es definir una estructura pública estándar con parámetros. Luego usamos la macro de derivación para implementar el rasgo Serialize. Esto permite que los atributos de la estructura se serialicen en JSON con el nom-

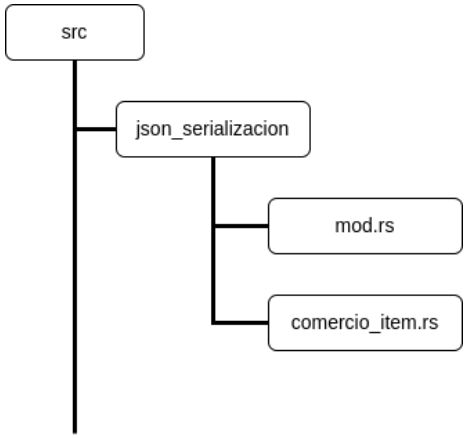

Figura 3.5: Estructura de serialización

bre del atributo como clave. Por ejemplo, si la estructura **ComercioItems** tuviera un **contador_item_hecho** de uno, entonces el cuerpo **JSON** lo denotaría como **"contador_item_hecho": 1**.

Ahora que se ha definido la serialización, tenemos que considerar el procesamiento de los datos. No sería escalable si tuviéramos que ordenar los datos y contarlos antes de llamar a la estructura. Esto agregaría código innecesario a la vista con respecto al procesamiento de datos para serialización en lugar de la lógica que pertenece a la vista en cuestión. También habilitaría el código duplicado. Solo habrá una forma de ordenar, contar y serializar los datos. Si otras vistas necesitaran devolver la lista de elementos, entonces tendríamos que duplicar el código nuevamente.

Teniendo esto en cuenta, tiene sentido construir un constructor para la estructura, donde ingerimos un vector de elementos pendientes, los clasificamos en los atributos correctos y luego los contamos:

```
impl ComercioItems {
        pub fn nuevo(items_entrada: Vec<TiposItem>) -> ComercioItems
            {
                let mut pendiente_array_buffer = Vec::new();
                let mut hecho_array_buffer = Vec::new();
                for item in items_entrada {
                        match item {
                                TiposItem::Pendiente(paquete) =>
                                   pendiente_array_buffer.push(
                                paquete.super_estructura),
                                TiposItem::Hecho(paquete) =>
                                   hecho_array_buffer.push(
                                paquete.super_estructura)
```

```
11                          }
12                          let contar_hecho: i8 = hecho_array_buffer.
                               len() as i8;
13                          let contar_pendiente: i8 =
                               pendiente_array_buffer.len() as i8;
14                          return ComercioItems{
15                                  items_pendiente:
                                       pendiente_array_buffer,
                                       contador_item_hecho: contar_hecho
                                       ,
16                                  contador_item_pendiente:
                                       contar_pendiente, items_hecho:
                                       hecho_array_buffer
17                          }
18                  }
19          }
20  }
```

Lo que hacemos aquí es aceptar un vector de **TiposItem**. Luego los desempaquetamos con una declaración de coincidencia y los insertamos (agregamos) en el vector mutable correcto. Luego llamamos a la función **len** en cada vector. La función **len** devuelve un tamaño de uso, que es un tipo de entero sin signo del tamaño de un puntero. Debido a esto, lo proyectamos como un **i8** y luego redefinimos y devolvemos nuestra estructura, que está lista para serializarse en **JSON**.

Para utilizar nuestra estructura, tenemos que definirla en nuestra vista **TRAER**, en nuestro archivo **vistas/comercio/traer.rs**, y luego devolverlo:

```
1  use actix_web::{web, Responder};
2  use serde_json::value::Value;
3  use serde_json::Map;
4  use crate::estado::leer_archivo;
5  use crate::comercio::{TiposItem, fabrica_comercio};
6  use crate::serializacion_json::comercio_items::ComercioItems;
7
8  pub async fn traer() -> impl Responder {
9          let estado: Map<String, Value> = leer_archivo("./estado.json
               ");
10         let mut array_buffer = Vec::new();
11         for (key, value) in estado {
12                 let item_type: String = String::from(value.as_str().
                       unwrap());
13                 let item: TiposItem = fabrica_comercio(&item_type, &
                       key).unwrap();
14                 array_buffer.push(item);
15         }
```

```
16          let regresar_paquete: ComercioItems = ComercioItems::nuevo(
                array_buffer);
17          return web::Json(regresar_paquete);
18  }
```

Aquí hay otro momento en el que todo encaja: usamos nuestra interfaz **leer_archivo** para obtener el estado del archivo JSON. Luego recorremos el mapa, convirtiendo el tipo de elemento en una cadena y introduciéndolo en nuestra interfaz **fabrica_comercio**. Una vez que tenemos el elemento construido de fábrica, lo agregamos a un vector y alimentamos ese vector en nuestra estructura de serialización JSON.

Para que nuestro módulo **serializacion_json** funcione en nuestra aplicación, debemos declararlo en nuestro archivo **main.rs** con el **mod serializacion_json**; línea de código. También tenemos que derivar la serialización en la estructura base en nuestro módulo **comercio** agregando la macro que definimos en nuestro archivo **src/comercio/estructuras/base.rs**, de la siguiente manera:

```
1  use serde::Serialize;
2
3  #[derive(Serialize)]
4  pub struct Base{
5          pub titulo: String,
6          pub estatus: String
7  }
8  impl Base {
9          pub fn nuevo(titulo_entrada: &str, estatus_entrada: &str) ->
               Base {
10                 return Base {titulo: titulo_entrada.to_string(),
11                         estatus: estatus_entrada.to_string()}
12         }
13 }
```

Esta es una respuesta limpia y bien estructurada que se puede ampliar y editar si es necesario a medida que se desarrolla nuestra aplicación. Podemos detenernos aquí, pero tenga en cuenta que nuestra vista **TRAER** devolvió una implementación del rasgo Responder. Esto significa que si nuestra estructura ComercioItems también implementa esto, se puede devolver directamente en una vista. Podemos hacer esto en nuestro archivo **serializacion_json/comercio_items.rs** con el siguiente bloque impl:

```
1  impl Responder for ComercioItems {
2          type Error = Error;
3          type Future = Ready<Result<HttpResponse, Error>>;
4          fn respond_to(self, _req: &HttpRequest) -> Self::Future {
5                  let body = serde_json::to_string(&self).unwrap();
6                  ready(Ok(HttpResponse::Ok()
```

```
7                   .content_type("application/json")
8                   .body(body)))
9           }
10  }
```

La función **respond_to** se activa cuando se devuelve la función de vista. Aquí, devolvemos un tipo que nosotros mismos definimos llamado **Future**. Esto se compone de una estructura **Ready** del **crate de futuros**, lo que indica que el futuro está inmediatamente listo con un valor. Dentro de esto hay una estructura de resultado, que puede ser un error o una **HttpRequest**.

Serializamos la estructura propia usando **&self** y la adjuntamos al cuerpo. Ahora, simplemente devolver nuestra estructura sin hacer ningún otro procesamiento en nuestra función **TRAER** se puede hacer creando un nuevo elemento pendiente y simplemente devolviéndolo. Esto se demuestra en el siguiente bloque de código:

```
1  let regresar_paquete: ComercioItems = ComercioItems::nuevo(
       array_buffer);
2  return regresar_paquete;
```

¡Esto nos da la misma respuesta que recibimos anteriormente! Ahora, estamos en el punto donde podemos refactorizar. Es razonable suponer que todas nuestras vistas de comercio requerirán una lista actualizada de comercio una vez finalizada la operación. Esta refactorización se puede realizar elevando todo el código de la función **TRAER** a una función llamada **regresar_estado** en un archivo **utils** en el directorio **vistas/comercio/utils.rs** (recuerde definir el archivo en **mod.rs**). La función **regresar_estado** devuelve la estructura **ComercioItems**. Esto luego acorta nuestra vista **TRAER** a lo siguiente: