Includes a **40% discount** code for CodeCrafters.io

Rust Projects

Write a Redis Clone

Explore asynchronous programming with the actor model using Rust and Tokio

Leonardo Giordani

Leonardo Giordani

Rust Projects - Write a Redis Clone

1st Edition - Release 1.0.0

Contents

Introduction

Our star runner tonight needs no introduction.
The Running Man (1987)

This book was born out of my desire to understand Rust better, and to learn it while creating something useful.

Being interested in many different things, such as coding, guitar playing, and painting, I find myself constantly in the enviable position of a beginner, eager to sail on an ocean of new concepts.

When I approach something new, whether it's a new programming language or a painting technique, I always realise how true it is that learning happens through experience. It is definitely useful to watch YouTube videos where great coders explain advanced concepts, but it's when I'm in front of an empty editor page that I'm forced to apply what I saw, to make it happen with my own hands.

A key principle is that our brains retain information better when it was challenging to acquire. And while it perfectly fine to copy and paste commands form StackOverflow (or LLM models) to solve one-off situations, I personally want to learn to use tools and languages properly and to be effective without having to look up solutions every other minute.

So, why a book on coding, if I just stated that we learn by doing and not by reading?

As the author, writing this book was an excellent opportunity to organise my notes, review decisions, and deepen my understanding of certain topics.

For anyone else, the book might be a jump start to get used to the language before you start coding on your own. Or it might be used as a comparison, to see an alternative solution to the same steps you already went through. Finally, if you want you can use it to just code along copying my solution, and taking your time to dig deep into the frequent references to the Rust documentation to better understand specific topics.

Or you can do all these things together. Having written, edited, corrected, written again, tested, and read every paragraph at least 3 times I can assure you that it was an amazing journey, full of discoveries and satisfaction. It will also be full of typos and mistakes, though. Forewarned is forearmed.

I hope you will enjoy it as much as I did!

Work in progress

This book is not 100% complete. I'm planning to add more sections while I complete some of the advanced challenges like replication, transactions, and persistence.

CodeCrafters

The idea of writing a Redis clone is all but original. It originates from one of the most interesting projects I encountered in the past decade, CodeCrafters. This company, founded in 2022 by Sarup Banskota and Paul Kuruvilla, provides a very simple service: a set of well-paced challenges that guide you to implement an important piece of software from scratch.

Are you into databases? Here is the Build your own Redis challenge. Do you want to create your own language? No problem, head to Build your own Interpreter. Want to dig into network protocols? Build your own HTTP server and Build your own DNS server to the rescue. And the nice thing is that you can solve it with your favourite programming language, and see the solutions of other coders.

It's TDD applied to integration tests, where CodeCrafters provides the code that tests features of your software, and you provide the code that implements such features.

You don't need CodeCrafters to enjoy the book, but I will follow the challenge "Build your own Redis" strictly, so you might benefit from having an account. You can get a discounted access using the following link `https://app.codecrafters.io/join?via=lgiordani` which gives you a free week of access to the platform and a discount of 40% if you decide to subscribe the paid tier.

Why this book comes for free

The first reason I started writing a technical blog was to share with others my discoveries, and to save them the hassle of going through processes I had already cracked. Moreover, I always enjoy the fact that explaining something forces me to better understand that topic, and writing requires even more study to get things clear in my mind, before attempting to introduce other people to the subject.

Much of what I know comes from personal investigations, but without the work of people who shared their knowledge for free I would not have been able to make much progress. The Free Software Movement didn't start with Internet, and I got a taste of it during the 80s and 90s, but the World Wide Web undeniably gave an impressive boost to the speed and quality of this knowledge sharing.

So this book is a way to say thanks to everybody gave their time to write blog posts, free books, software, and to organise conferences, groups, meetups. This is why I teach people at conferences, this is why I write a technical blog, this is the reason behind this book.

That said, if you want to acknowledge my effort with money, thanks! However, the best thing you can do is to become part of this process of shared knowledge; experiment, learn and share what you learn.

Typographic conventions

Here are some typographic conventions used throughout the book.

Regular text appears like this, while code inline code is displayed `like this`. When code refers to a library a link to the documentation is provided in this form [docs].

When the text mentions concepts that are not directly connected with the project at hand but are worth further investigation **an aside will contain details** or links.

Aside

This is an aside with more information and links on specific topics.

The **code of the project** is presented in a box like this

src/main.rs

```rust
fn main() {
    println!("Hello, world!");
}
```

This means that the given code is added to the file specified in the title.

If the **change affects only specific lines** those will be highlighted. Please note that this is not a diff, so the previous version of the code is not shown.

src/main.rs

```rust
fn main() {
    println!("Good night, world!");
}
```

In certain cases, it's useful to **reason about code structure** or to highlight specific details. The following box with a magnifying glass icon shows parts of the code that are described in the text.

src/main.rs

```rust
fn main() {
    println!("Hello, world!");
}
```

If the **code shown is not correct**, either because it doesn't compile or for other reason, it will be shown like this.

src/main.rs [NOT WORKING]

```rust
fn main() {
    println!("Hello, world!");
}
```

The output of **commands run in the terminal**, such as the compiler, the test suite, or other shell commands is shown in a box like this.

```
$ run -a --command

This is the output of the command in a terminal
```

The code shown in the book can be used to **complete the CodeCrafters challenge** "Build your own Redis". Whenever the code passes a specific stage a box like this will provide details.

CodeCrafters

Stage 0: Take over the world!

This is a pointer to the CodeCrafters stage that is covered by the code.

Finally, the code of the project is **available in a repository** and the current step will be highlighted with a box like this

Source code

https://github.com/lgiordani/sider/

Setup the development environment

As the project consists in a sort of reverse engineering of Redis, I decided to call it "Sider", which is Redis written backwards. To follow the project you need an editor (pick your favourite one and make sure you have support for Rust) and the Rust compiler. You can install Rust in your system following the official guide. To be able to use CodeCrafters you also need to configure a Git repository, but you can follow the instructions in the challenge page.

Create the Rust project (CodeCrafters)

If you are using Codecrafters, follow the instructions they provide. You will clone a repository provided by them that contains the Rust project and a script to run the server.

Create the Rust project (manual)

To create the project, I recommend using Cargo

```
$ cargo new sider
$ cd sider
```

This will create a directory called `sider` that contains the standard files and directories of a basic Rust project. You can test that everything works correctly with

```
$ cargo run
```

Now open the file `src/main.c` with your favourite editor and remove the demo code.

Also, create the file `spawn_redis_server.sh` that contains the following code

`spawn_redis_server.sh`

```
exec cargo run \
    --quiet \
    --release \
    --manifest-path $(dirname $0)/Cargo.toml \
    -- "$@"
```

As you can see, this is just a wrapper around `cargo run` and will be useful to run tests.

How to test code (CodeCrafters)

If you are using CodeCrafters, tests will be run every time you commit and push. Make sure you read the official documentation, in particular if you want to use the CodeCrafters CLI.

How to test code (manual)

Warning

The instructions in this section are not official, and CodeCrafters pointed out that the structure of the repository might change in the future. Keep that in mind!

You can also run the CodeCrafters tests manually on your machine. Clone the tests repository and change the file `internal/test_helpers/pass_all/spawn_redis_server.sh` replacing `redis-server` with the full path of the script in your project

`internal/test_helpers/pass_all/spawn_redis_server.sh`

```
#!/bin/sh
find "." -type f -name "*.rdb" -exec rm {} +
exec /My/PROJECT/PATH/spawn_redis_server.sh --loglevel nothing $@
```

You can then run the tests locally using the provided `Makefile`. There are several tests that correspond to the base challenge and to the extensions. For the base challenge run

```
$ make test_base_with_redis
```

This will however run all the tests in the base challenge, so if you want to check only the steps you completed so far you can copy that rule into a custom one and add steps while you progress, e.g.

`Makefile`

```
test_base_with_redis_prog: build
	CODECRAFTERS_SUBMISSION_DIR=./internal/test_helpers/pass_all \
	CODECRAFTERS_TEST_CASES_JSON="[\
		{\"slug\":\"jm1\",\"tester_log_prefix\":\
                  \"stage-1\",\"title\":\"Stage #1: Bind to a port\"},\
                 {\"slug\"
                  \"stage-2\",\"title\":\"Stage #2: Respond to PING\"},\
         ]" \
         dist/main.out
```

About the book

The book has been written using Mau and rendered with TeX.

The cover photo is by Tengyart and can be found on Unsplash. It represents "rich metal texture with rust, green paint and scratches" and I found it to be both artistically attractive and aptly matching the language we are going to use.

Chapter 1

Initial Steps

We'II go step by step and cut off every bulkhead and every vent untiI we have it cornered.
Alien (1979)

The initial steps are pretty standard in an application like this. We will create a server that first **listens on a specific TCP port**. Then we will change the code to **respond to a single incoming request** and terminate. As a third step, the server will **respond to multiple incoming requests** from the same client, and finally it will **serve multiple clients**.

At the end of the chapter we will have a running application that can be used with the official Redis CLI tool to serve the command `PING`.

Step 1 - Bind to a port

The first thing we need to do is to create a server that binds to TCP port 6379.

The Rust standard library provides `TcpListener` [docs] that can be bound to a socket.

`src/main.rs`

```rust
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").unwrap();

    for stream in listener.incoming() {
        match stream {
            Ok(_stream) => {
                println!("accepted new connection");
            }
            Err(e) => {
                println!("error: {}", e);
            }
        }
    }
}
```

This is the classic initial step for a server. We hard coded the address of the server (`127.0.0.1`) and the port (`6379`) but in the future we might want to make both value configurable through a specific command line option.

Some comments on the code itself:

- `unwrap` [docs] is a crude way to extract the `Ok` value from a `Result`. It's a convenient method for tests, as it panics if the result is an error, but in production code it might not be the best solution.
- `println!` [docs] is how we print text on the standard output. While there are more sophisticated ways to debug programs, printing is still a good way to understand what is going on in a system, and to log events.
- A concept somehow hidden in the code above is that of `Iterator` [docs] which in this case is implemented by `Incoming`, that is the return type of `TcpListener::incoming`.

TCP protocol

You don't need to know the (rather complicated) details of the TCP protocol, but as a software developer you should be familiar with IP addresses and TCP ports.

Testing with the Redis CLI

Other than using the CodeCrafters tests you can interact with your server using the Redis CLI. Once you installed it in your system you can first run your server with

```
$ cargo run
```

or, if you are using the CodeCrafters setup, running

```
$ spawn_redis_server.sh
```

and then open a new terminal and run `redis-cli`

```
$ redis-cli
127.0.0.1:6379>
```

The server should react printing the message `accepted new connection`. You cannot issue commands at the moment, as the server ignores data coming through the stream.

CodeCrafters

Stage 1: Bind to a port

The application doesn't do much at the moment, and running `cargo run` won't show anything interesting. However, this passes Stage 1 of the CodeCrafters challenge.

Source code

https://github.com/lgiordani/sider/tree/ed1/step1.1

Step 2 - Respond to PING

The server has to respond to incoming requests, so the next step is to start listening on the TCP connection and parse the incoming data. Redis uses a binary protocol called Redis Serialization Protocol (RESP), so to understand the request we will eventually have to implement a parser for this protocol.

As a first step, however, we can just discard the incoming data and send back the same response regardless of the request. The simplest Redis command is `PING`, which receives the response `+PONG\r\n`. This is the string `PONG` encoded as a RESP simple string, as we will see later.

`src/main.rs`

```rust
use std::io::{Read, Write};
use std::net::{TcpListener, TcpStream};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").unwrap();

    for stream in listener.incoming() {
        match stream {
            Ok(mut stream) => {
                handle_connection(&mut stream);
            }
            Err(e) => {
                println!("error: {}", e);
            }
        }
    }
}

fn handle_connection(stream: &mut TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let response = "+PONG\r\n";
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

In this version we isolate the management of the incoming connections. Each new connection, represented by a `TcpStream` [docs], is passed to the function `handle_connection`. There, we `read` [docs] from the stream into a binary buffer, then we prepare the response and `write` [docs] it to the stream. At the moment, the code still uses `unwrap` to manage the error cases, but in the future it will be worth dealing properly with errors.

`Iterator` and `Item`

How do we know the type of `stream`? Have a look at `TcpListener::incoming` [docs] and you will see that the return type is the struct `Incoming` [docs]. This in turn implements `Iterator` [docs] and specifies `type Item = Result<TcpStream, Error>`, which leads us to `TcpStream` [docs].

Testing and TDD

Testing is arguably more complicated in strongly typed compiled languages like Rust than it is in languages like Python. In particular, while it is possible to create mocks, it is in general hard to replace functions at run time. So, a strict TDD approach is, in my opinion, impossible. This forces us to apply inversion of control more often, but it's important to understand that we cannot achieve the same level of inspection that we have in languages with a higher level of abstraction.

For example, in the current case the classic strategy in a high-level OOP language would be:

- create a mock of a `TcpStream` that has only the method `read`
- run the function `handle_connection`
- check that the mock method has been called with the right value

But in Rust this is more complicated. We have to pass the function a value of type `TcpStream` and while we can redefine `TcpStream` just for tests with some clever use of conditional checks like `#[cfg(...)]`, this would clutter the production code and cannot be done for a single test only.

So, we will add tests to the code following the TDD approach, but not all cases will be covered. For now, it's perfectly fine to print messages and data on the standard output to understand what the code is doing. Here, you can add a `println!` to see what the server receives from the client

src/main.rs

```rust
fn handle_connection(stream: &mut TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    println!("Received: {:?}", buffer);

    let response = "+PONG\r\n";
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

The code of the project doesn't contain any prints to avoid cluttering the tests output, but feel free to add them to better understand what happens behind the scenes.

Memory and safety

A prominent part of the Rust language deals with safe memory usage, so it's paramount to understand how to use references and mutability, in particular when it comes to passing values to functions. I highly recommend Patterns Are Not Expressions by Árpád Goretity and There are no mutable parameters in Rust by Michael Snoyman.

Ownership, which stems directly from safe memory usage, is everywhere in Rust, and can arguably be a pain to deal with, sometimes. Understanding the rationale of ownership is paramount and should be done upfront, otherwise you'll spend a lot of time being frustrated by the compiler's complains. I recommend to read at least chapter 4 of the Rust book.

CodeCrafters

Stage 2: Respond to PING

The application passes the Stage 2 of the CodeCrafters challenge. We can't use the Redis CLI to interact with it yet, though. The server terminates the connection immediately after sending the response, which is something the CLI tool doesn't like.

Source code

`https://github.com/lgiordani/sider/tree/ed1/step1.2`

Step 3 - Respond to multiple PING

The problem at this point is that the function `handle_connection` terminates once it processes one response, while it should keep it open, receiving bytes from the client and sending responses.

This is fixed quickly enough by moving the code of the function into a loop. As we do that, however, it is worth taking the time to properly manage the output of `TcpStream::read` [docs]. The function returns a `Result<usize>` that contains the amount of bytes received. This is a blocking function, so if the amount of bytes is 0 the connection has been closed, and if the result is `Err` something went wrong with the stream.

src/main.rs

```rust
fn handle_connection(stream: &mut TcpStream) {
    let mut buffer = [0; 512];

    loop {
        match stream.read(&mut buffer) {
            Ok(size) if size != 0 => {
                let response = "+PONG\r\n";
                stream.write(response.as_bytes()).unwrap();
                stream.flush().unwrap();
            }
            Ok(_) => {
                println!("Connection closed");
                break;
            }
            Err(e) => {
                println!("Error: {}", e);
                break;
            }
        }
    }
}
```

It is important to understand the blocking nature of `TcpStream::read`. When the code executes the function the whole program sits waiting for something to happen on the TCP connection, without doing anything until data appears or an error happens.

This is clearly unacceptable for a server that has to interact with multiple clients, so in the next step we will see a solution to this problem.

Loops and pattern matching

`loop` [docs] is an interesting construct. In other languages infinite loops with breaks are often frowned upon, while in Rust their are not only accepted but considered an idiomatic solution.

Pattern matching [docs] is a powerful concept that is predominant in functional programming languages. It's important to learn how to deconstruct types and how to use guards. Rust is very strict about covering all possible patterns, which greatly helps to avoid bugs.

Testing with the Redis CLI

Now that the server is actually listening you can open a new terminal, run `redis-cli`, and have a brief interaction with your server sending a `PING` command.

```
$ redis-cli
127.0.0.1:6379> ping
PONG
```

At the moment your server will respond with `PONG` whatever the request is, but we will soon implement other commands

```
127.0.0.1:6379> echo 42
PONG
127.0.0.1:6379>
```

CodeCrafters

Stage 3: Respond to multiple PINGs

The code we have at the moment passes Stage 3 of the CodeCrafters challenge. The application is not truly processing the incoming request yet, but the backbone is in place.

Source code

https://github.com/lgiordani/sider/tree/ed1/step1.3

Step 4 - Handle concurrent clients

Now the game starts to become a bit more serious, both in terms of requirements and in terms of the technical implementation. A server needs to interact with multiple clients, so we have to introduce concurrency.

There are mainly three ways to implement concurrency: multiprocessing, multithreading, and asynchronous programming. They have different pros and cons and depending on the language you might find one of them easier to implement than others.

In this case we will use asynchronous programming. The application we are implementing is mostly running I/O bound tasks and it's worth exploring how Rust and its strict memory model work with async/await.

Asynchronous programming

There is an huge amount of resources available on asynchronous programming both in general and specifically to Rust. I highly recommend reading first one or more generic introductions about multiprocessing and multithreading at the operating system level.

As for Rust, threads are explained in chapter 16 of the Rust book [docs] while async has its own book [docs]. Among many others, Jon Gjengset is doing an incredible work of digging deep into Rust topics on his YouTube channel. The video Crust of Rust: async/await, in particular, can answer many questions.

To run the asynchronous loop we will use Tokio. Let's add the crate to the project

```
$ cargo add tokio --features full
```

which will add a dependency to `Cargo.toml`

`Cargo.toml`

```
[dependencies]
tokio = { version = "1.38.0", features = ["full"] }
```

We can then import the relevant components, replacing the previous imports

src/main.rs

```
use tokio::{
    io::{AsyncReadExt, AsyncWriteExt},
    net::{TcpListener, TcpStream},
};
```

The changes to the code are minimal, reflecting the fact that the syntax of asynchronous programs tries to follow the synchronous paradigm, hiding the complexity of concurrency.

There are four main changes in the function `handle_connection`:

- the word `async` in front of its definition
- the function takes ownership of the stream
- the need to `await` the result of `stream.read(&mut buffer)`
- `write` and `flush` are merged into `write_all`, which needs an `await` as well

We can also add some error management around `stream.write_all` that wasn't there previously. This is completely unrelated to `async`, as we could still use `unwrap` and let the server panic, but that doesn't sound like a good idea now that the code starts to evolve.

src/main.rs

```
async fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    loop {
        match stream.read(&mut buffer).await {
            Ok(size) if size != 0 => {
                let response = "+PONG\r\n";

                if let Err(e) = stream.write_all(response.as_bytes()).await {
                    eprintln!("Error writing to socket: {}", e);
                }
            }
            Ok(_) => {
                println!("Connection closed");
                return;
            }
            Err(e) => {
                eprintln!("Error: {}", e);
                return;
            }
        }
    }
}
```

The main changes are in the function `main`. Here, we need to instantiate the asynchronous loop and create an independent handler for each connection

src/main.rs

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    loop {
        match listener.accept().await {
            Ok((stream, _)) => {
                tokio::spawn(handle_connection(stream));
            }
            Err(e) => {
                println!("Error: {}", e);
                continue;
            }
        }
    }
}
```

For the function `main` to be `async` we need to decorate it with `#[tokio::main]` which, behind the scenes, transforms it into a normal entry point that creates the main loop and runs the rest of the code.

src/main.rs

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    ...
```

`TcpListener::bind` returns `std::io::Result<()>`, and adding the same return type to `main` makes sense as that is the only point where the function should be able to crash. This allows us to use the operator `?` to unwrap the result of `bind` or return the error.

Error checking and propagation

Error handling [docs] is an important topic in every programming language, but it requires even more planning in a strongly typed one.

Rust has a very elegant way to manage error conditions that can arise in complicated chains of functions or in loops: the operator `?` [docs]. To use it, you need to ensure compatibility between the error and the return type of the current function, so you might want to look into automatic type conversion with `From` [docs]

The new connection monitoring function `TcpListener::accept` [docs] is not an iterator any more. It's an asynchronous function, so the pattern is to wrap it in a `loop` and to `await` it. Each time a new connection is created the function returns a tuple `(TcpStream, SocketAddr)`, of which we keep only the stream.

src/main.rs

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    loop {
        match listener.accept().await {
            ...
        }
    }
}
```

The core of the architecture is clearly `tokio::spawn`, that creates a new task. This means that the provided function will be run in isolation in the asynchronous loop, and that the system will pause it every time an `await` is invoked in its code.

src/main.rs

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {

    ...

        match listener.accept().await {
            Ok((stream, _)) => {
                tokio::spawn(handle_connection(stream));
            }

    ...
}

async fn handle_connection(mut stream: TcpStream) {
    ...
```

A common pattern for `spawn` is to run it in combination with `async move` [docs], passing some closure defined on the spot. This is done to make sure that ownership of the closure context is transferred to the task. In this case, there is no need here to use `async move` as the function `handle_connection` already takes ownership of `stream`.

CodeCrafters

Stage 4: Handle concurrent clients

This version of the code passes Stage 4 of the CodeCrafters challenge. We can actually connect to the server using multiple Redis CLI instances in different terminals.

Since in the next chapter we will go through some refactoring, I recommend not to mark the stage as completed and wait before you move to the next step. This way, you can keep checking that what we are doing still passes the tests.

Source code

`https://github.com/lgiordani/sider/tree/ed1/step1.4`

Chapter 2

The RESP Protocol

This conversation can serve no purpose any more.
2001: A Space Odyssey (1968)

Now that we have a basic generic server we need to focus on an important feature that is more closely related to Redis: the RESP protocol.

Redis clients and servers communicate with a custom binary protocol called RESP (REdis Serialization Protocol). On the redis website you can find the full specification, but we won't implement the whole beast.

To complete the basic stages of the challenge we need to implement the following elements:

- Simple string
- Bulk string
- Array

This might sound trivial, but when it comes to binary protocols there are a lot of low-level functions to implement. So, while it won't be complicated, it will take the whole chapter to reach a working implementation.

Once these elements and the underlying machinery are in place it will be a breeze to implement other types (like Integer) for later stages.

The RESP binary format

RESP is roughly made of a binary representation of certain data types and of a format for requests and responses (both successful and unsuccessful).

The official documentation explains the difference between RESP types, even though some corner cases might have been clarified with more examples. For the time being, you should familiarise yourself with the way simple strings, bulk strings, and arrays are implemented.

Please also note that Redis commands shall be sent by the client as RESP arrays of bulk strings and that responses will be sent according to the requested command. For example, the command `GET` [docs] will reply with either a bulk string with the value of the key or with a null.

The steps we will follow in this chapter are:

1. **Define a custom result type** to have a nice way to manage successful results and errors.
2. **Extract binary values from RESP** will give us the raw binary data.
3. **Convert binary values to string** will give us the corresponding Rust string..
4. **Parse the RESP type** to be able to identify the RESP data type.
5. **Parse a RESP simple string** to put together what has been done so far.
6. **Use simple string for PING** to see what we have done in action.
7. **Parse generic RESP** to process a generic binary buffer containing RESP data.
8. **Parse a RESP bulk string**.
9. **Parse a RESP array**.
10. **Process PING the right way** to use generic RESP functions to process `PING`.
11. **Process ECHO** - to use generic RESP functions to process `ECHO`.

Unfortunately, the development of this part of the system has little to show in terms of Redis features. Having RESP in place will allow us to implement `ECHO`, `GET` and `SET`, but we need to climb this hill before we see those cities.

The positive side is that we will be able to use and learn TDD in Rust, as the functions that we will develop can be tested very easily.

Step 1 - Define a custom result type

It's always worth to have a proper description of errors, and this is even more true when it comes to parsing, which is notoriously error-prone. Let's start implementing an enum for errors and a type for results

src/resp_result.rs

```rust
#[derive(Debug)]
pub enum RESPError {}

pub type RESPResult<T> = Result<T, RESPError>;
```

We will add enum values to `RESPError` later when we need to encapsulate specific errors in the parsing stage.

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.1

Step 2 - Extract binary values from RESP

The binary values 13 and 10 (ASCII for `\r\n`) are the main separator between RESP parts, so it makes sense to be able to read binary data until those characters are met.

The first function we will implement is `binary_extract_line` that extracts bytes from a buffer until a `\r\n` is reached. Here, `index` represents the position where we start parsing the buffer, and it will be updated by the function.

src/resp.rs

```rust
use crate::resp_result::{RESPError, RESPResult};

// Extracts bytes from the buffer until a `\r` is reached
fn binary_extract_line(buffer: &[u8], index: &mut usize) -> RESPResult<Vec<u8>> {
    let mut output = Vec::new();

    // We try to read after the end of the buffer
    if *index >= buffer.len() {
        return Err(RESPError::OutOfBounds(*index));
    }

    // If there is not enough space for \r\n
    // the buffer is definitely invalid
    if buffer.len() - *index - 1 < 2 {
        *index = buffer.len();
        return Err(RESPError::OutOfBounds(*index));
    }

    let mut previous_elem: u8 = buffer[*index].clone();
    let mut separator_found: bool = false;
    let mut final_index: usize = *index;

    // Scan the whole buffer looking for \r\n
    for &elem in buffer[*index..].iter() {
        final_index += 1;

        if elem == b'\n' && previous_elem == b'\r' {
            separator_found = true;
            break;
        }
        previous_elem = elem.clone();
    }

    // If the previous element is not \n
    // we are out of bounds
    if !separator_found {
        *index = final_index;
        return Err(RESPError::OutOfBounds(*index));
    }

    // Copy the bytes from the buffer to the output vector
    output.extend_from_slice(&buffer[*index..final_index - 2]);
    // Make sure the index is updated with the latest position
    *index = final_index;

    Ok(output)
}
```

As you can see, this is a pretty low-level interaction with the binary buffer. There are tests that cover several corner cases

src/resp.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_binary_extract_line_empty_buffer() {
        let buffer = "".as_bytes();
        let mut index: usize = 0;

        match binary_extract_line(buffer, &mut index) {
            Err(RESPError::OutOfBounds(index)) => {
                assert_eq!(index, 0);
            }
            _ => panic!(),
        }
    }

    #[test]
    fn test_binary_extract_line_single_character() {
        let buffer = "O".as_bytes();
        let mut index: usize = 0;

        match binary_extract_line(buffer, &mut index) {
            Err(RESPError::OutOfBounds(index)) => {
                assert_eq!(index, 1);
            }
            _ => panic!(),
        }
    }

    #[test]
    fn test_binary_extract_line_index_too_advanced() {
        let buffer = "OK".as_bytes();
        let mut index: usize = 1;

        match binary_extract_line(buffer, &mut index) {
            Err(RESPError::OutOfBounds(index)) => {
                assert_eq!(index, 2);
            }
            _ => panic!(),
        }
    }

    #[test]
    fn test_binary_extract_line_no_separator() {
        let buffer = "OK".as_bytes();
        let mut index: usize = 0;

        match binary_extract_line(buffer, &mut index) {
            Err(RESPError::OutOfBounds(index)) => {
                assert_eq!(index, 2);
            }
            _ => panic!(),
        }
    }

    #[test]
    fn test_binary_extract_line_half_separator() {
        let buffer = "OK\r".as_bytes();
        let mut index: usize = 0;

        match binary_extract_line(buffer, &mut index) {
            Err(RESPError::OutOfBounds(index)) => {
                assert_eq!(index, 3);
            }
            _ => panic!(),
        }
    }

    #[test]
    fn test_binary_extract_line_incorrect_separator() {
        let buffer = "OK\n".as_bytes();
        let mut index: usize = 0;

        match binary_extract_line(buffer, &mut index) {
            Err(RESPError::OutOfBounds(index)) => {
                assert_eq!(index, 3);
            }
            _ => panic!(),
        }
    }

    #[test]
    fn test_binary_extract_line() {
        let buffer = "OK\r\n".as_bytes();
        let mut index: usize = 0;

        let output = binary_extract_line(buffer, &mut index).unwrap();

        assert_eq!(output, "OK".as_bytes());
        assert_eq!(index, 4);
    }
}
```

The code relies on a specific error variant, `RESPError::OutOfBounds`, that has to be added to the enum. The error can be made printable implementing `fmt::Display` for the type src/resp_result.rs

```rust
use std::fmt;

#[derive(Debug)]
pub enum RESPError {
    OutOfBounds(usize),
}

impl fmt::Display for RESPError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            RESPError::OutOfBounds(index) => write!(f, "Out of bounds at index {}", index),
        }
    }
}
```

Traits

Rust is not an object-oriented programming language, but some constructs can definitely be compared to equivalent OOP ones. Rust structs and enums, for example, cannot inherit from a parent structure, but we can write an implementation that adds methods to the data type, in a way that reminds of classes.

One of the most powerful concepts in Rust is that of a trait [docs]. For those coming from object-oriented languages, traits are halfway between interfaces and mixins.

As interfaces, they can be used in function prototypes to represent types, with the rich trait bound syntax `impl SomeTrait` (or its longer form `<T: SomeTrait>` after the function name). As mixins, they can behave like "classes without data", providing a default method implementation that can be attached to an existing type with an empty `impl` block or with `#[derive]`.

A lot of language features like automatic type conversion depend on traits, so it's highly recommended to become familiar with the concept and its syntax.

Last, the new files have to be added as mods to `main.rs`

src/main.rs

```
use tokio::{
    io::{AsyncReadExt, AsyncWriteExt},
    net::{TcpListener, TcpStream},
};

mod resp;
mod resp_result;
```

At this point we can run tests with `cargo test`

```
$ cargo test

...

running 7 tests
test resp::tests::test_binary_extract_line_empty_buffer ... ok
test resp::tests::test_binary_extract_line ... ok
test resp::tests::test_binary_extract_line_half_separator ... ok
test resp::tests::test_binary_extract_line_index_too_advanced ... ok
test resp::tests::test_binary_extract_line_incorrect_separator ... ok
test resp::tests::test_binary_extract_line_no_separator ... ok
test resp::tests::test_binary_extract_line_single_character ... ok

test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
↪  0.00s
```

Binary values and strings

Strings are a surprisingly complicated topic in computer science. While binary values are relatively simple once we agree on the definition of byte, strings can be implemented in several different ways. In particular, programming languages differ in the way they represent strings in memory, and in how they deal with encoding.

When it comes to Rust, it's important to understand the two types of strings `String` [docs] and `&str` [docs]. As for the encoding, all modern languages embraced Unicode and UTF-8, so once again these are concepts one should be at least superficially familiar with.

You can learn more about Rust Strings and UTF-8 in the the Rust book.

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.2

Step 3 - Convert binary values to string

In certain cases we know for sure that the RESP content is meant to represent UTF-8 data, which means that it will be useful to have a function that extracts binary data and converts it into a Rust `String`.

```
src/resp.rs
```

```rust
// Extracts bytes from the buffer until a `\r` is reached and converts them into a string
pub fn binary_extract_line_as_string(buffer: &[u8], index: &mut usize) -> RESPResult<String
↪   > {
    let line = binary_extract_line(buffer, index)?;

    Ok(String::from_utf8(line)?)
}
```

And the associated test

```
src/resp.rs
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_binary_extract_line_as_string() {
        let buffer = "OK\r\n".as_bytes();
        let mut index: usize = 0;

        let output = binary_extract_line_as_string(buffer, &mut index).unwrap();

        assert_eq!(output, String::from("OK"));
        assert_eq!(index, 4);
    }

    ...
}
```

The function `String::from_utf8()` [docs] returns `Result<String, FromUtf8Error>`, so we need to manage that error. The code `String::from_utf8(line)?` is at the moment incompatible with the return type `RESPResult<String>` that contains a `RESPError`.

We can implement the trait `From` that implicitly converts `FromUtf8Error` into `RESPError` src/resp_result.rs

```
use std::fmt;
use std::string::FromUtf8Error;

#[derive(Debug)]
pub enum RESPError {
    FromUtf8,
    OutOfBounds(usize),
}

impl fmt::Display for RESPError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            RESPError::FromUtf8 => write!(f, "Cannot convert from UTF-8"),
            RESPError::OutOfBounds(index) => write!(f, "Out of bounds at index {}", index),
        }
    }
}

impl From<FromUtf8Error> for RESPError {
    fn from(_err: FromUtf8Error) -> Self {
        Self::FromUtf8
    }
}

pub type RESPResult<T> = Result<T, RESPError>;
```

Automatic type conversion

Whenever data is passed to a function, Rust checks that the data type corresponds to the function signature. If it doesn't, Rust tries to perform a conversion between the two data types using the method `from`, which is provided by a specific trait `From<T>`. As we see in the code above, this is extremely useful when it comes to deal with discrepancies between return types.

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.3

Step 4 - Parse the RESP type

RESP types are always prefixed with a specific character, so we need something that removes the prefix before we can parse the rest. We could just skip the first byte, but it's better to double check that the buffer contains exactly what we expect.

src/resp.rs

```rust
// Checks that the first character of a RESP buffer is the given one and removes it.
pub fn resp_remove_type(value: char, buffer: &[u8], index: &mut usize) -> RESPResult<()> {
    if buffer[*index] != value as u8 {
        return Err(RESPError::WrongType);
    }

    *index += 1;

    Ok(())
}
```

The tests for this function are

src/resp.rs

```rust
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_binary_remove_type() {
        let buffer = "+OK\r\n".as_bytes();
        let mut index: usize = 0;

        resp_remove_type('+', buffer, &mut index).unwrap();

        assert_eq!(index, 1);
    }

    #[test]
    fn test_binary_remove_type_error() {
        let buffer = "*OK\r\n".as_bytes();
        let mut index: usize = 0;

        let error = resp_remove_type('+', buffer, &mut index).unwrap_err();

        assert_eq!(index, 0);
        assert_eq!(error, RESPError::WrongType);
    }
    ...
}
```

The code relies on the error `RESPError::WrongType`. We also have to add `PartialEq` to `RESPError` to be able to compare values during the tests.

src/resp_result.rs

```rust
#[derive(Debug, PartialEq)]
pub enum RESPError {
    FromUtf8,
    OutOfBounds(usize),
    WrongType,
}

impl fmt::Display for RESPError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            RESPError::FromUtf8 => write!(f, "Cannot convert from UTF-8"),
            RESPError::OutOfBounds(index) => write!(f, "Out of bounds at index {}", index),
            RESPError::WrongType => write!(f, "Wrong prefix for RESP type"),
        }
    }
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.4

Step 5 - Parse a RESP simple string

With this last function in place, we can define a RESP simple string as a variant of an enum and parse a binary buffer into it.

First let's define the enum and the variant

src/resp.rs

```rust
use crate::resp_result::{RESPError, RESPResult};

#[derive(Debug, PartialEq)]
pub enum RESP {
    SimpleString(String),
}
```

Then let's create a specific function that parses a binary buffer into that variant

src/resp.rs

```rust
// Parse a simple string in the form `+VALUE\r\n`
fn parse_simple_string(buffer: &[u8], index: &mut usize) -> RESPResult<RESP> {
    resp_remove_type('+', buffer, index)?;

    let line: String = binary_extract_line_as_string(buffer, index)?;

    Ok(RESP::SimpleString(line))
}
```

The associated test is

src/resp.rs

```rust
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_parse_simple_string() {
        let buffer = "+OK\r\n".as_bytes();
        let mut index: usize = 0;
        let output = parse_simple_string(buffer, &mut index).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("OK")));
        assert_eq!(index, 5);
    }

    ...

}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.5

Step 6 - Use simple string for PING

Let's take a quick break from the implementation of RESP types and use `RESP::SimpleString` to send the response to `PING`. It's a small change, but it will give us a sense of the link between what we are doing with RESP and the overall system.

src/main.rs

```
use crate::resp::RESP;

...

async fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    loop {
        match stream.read(&mut buffer).await {
            Ok(size) if size != 0 => {
                let response = RESP::SimpleString(String::from("PONG"));

                if let Err(e) = stream.write_all(response.to_string().as_bytes()).await {
                    eprintln!("Error writing to socket: {}", e);
                }
            }

    ...

}
```

However, the type `RESP` cannot be converted to a String with `to_string()` because it doesn't implement the right trait yet. Let's do it

src/resp.rs

```
use crate::resp_result::{RESPError, RESPResult};
use std::fmt;

#[derive(Debug, PartialEq)]
pub enum RESP {
    SimpleString(String),
}

impl fmt::Display for RESP {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let data = match self {
            Self::SimpleString(data) => format!("+{}\r\n", data),
        };
        write!(f, "{}", data)
    }
}
```

Which trait for string conversion?

To provide the code that converts a type into a string we implement two different traits:

- `Into<String>`, when implemented, is called automatically by Rust to convert a type into a `String`, and is considered a general-purpose conversion trait.

- `fmt:Display`, automatically implements the trait `ToString` as well. This trait is usually considered more suitable for strings that have to be printed on screen.

Ultimately, they both provide the same service but with a slightly different flavour.

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.6

Step 7 - Parse generic RESP

Now that the basic functions have been created we can move to a higher level. We eventually need to parse a generic RESP buffer, so we should implement a function that accepts a binary slice and returns the right variant of `RESP`

src/resp.rs

```
fn parser_router(
    buffer: &[u8],
    index: &mut usize,
) -> Option<fn(&[u8], &mut usize) -> RESPResult<RESP>> {
    match buffer[*index] {
        b'+' => Some(parse_simple_string),
        _ => None,
    }
}

pub fn bytes_to_resp(buffer: &[u8], index: &mut usize) -> RESPResult<RESP> {
    match parser_router(buffer, index) {
        Some(parse_func) => {
            let result: RESP = parse_func(buffer, index)?;
            Ok(result)
        }
        None => Err(RESPError::Unknown),
    }
}
```

The function `parser_router` returns one of the parsing functions according to the initial character in the buffer. The function `bytes_to_resp` uses the returned function to parse the buffer and return the result. This is done mainly to simplify the code by separating responsibilities and having shorter, more essential functions.

When the content of the RESP buffer cannot be parsed we return a specific error that has to be added to the `RESPError` enum.

src/resp_result.rs

```
#[derive(Debug)]
pub enum RESPError {
    FromUtf8,
    OutOfBounds(usize),
    Unknown,
    WrongType,
}
```

```
impl fmt::Display for RESPError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            RESPError::FromUtf8 => write!(f, "Cannot convert from UTF-8"),
            RESPError::OutOfBounds(index) => write!(f, "Out of bounds at index {}", index),
            RESPError::Unknown => write!(f, "Unknown format for RESP string"),
            RESPError::WrongType => write!(f, "Wrong parsing route for RESP type"),
        }
    }
}
```

The tests for this code are

src/resp.rs

```
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_bytes_to_resp_simple_string() {
        let buffer = "+OK\r\n".as_bytes();
        let mut index: usize = 0;

        let output = bytes_to_resp(buffer, &mut index).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("OK")));
        assert_eq!(index, 5);
    }

    #[test]
    fn test_bytes_to_resp_unknown() {
        let buffer = "?OK\r\n".as_bytes();
        let mut index: usize = 0;

        let error = bytes_to_resp(buffer, &mut index).unwrap_err();

        assert_eq!(error, RESPError::Unknown);
        assert_eq!(index, 0);
    }

    ...

}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.7

Step 8 - Parse a RESP bulk string

The second RESP type we need is a bulk string. We can start adding the variant `RESP::BulkString`, together with `RESP::Null` that will represent the empty string.

src/resp.rs

```
#[derive(Debug, PartialEq)]
pub enum RESP {
    BulkString(String),
    Null,
    SimpleString(String),
}

impl fmt::Display for RESP {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let data = match self {
            Self::BulkString(data) => format!("${}\r\n{}\r\n", data.len(), data),
            Self::Null => String::from("$-1\r\n"),
            Self::SimpleString(data) => format!("+{}\r\n", data),
        };

        write!(f, "{}", data)
    }
}
```

Bulk Strings provide the length of the contained data as a prefix, so it makes sense to use that to speed up the extraction of bytes from the buffer. The function `binary_extract_bytes` is very similar to `binary_extract_line`, but relies on a provided `length`

src/resp.rs

```
// Extracts a given amount of bytes from the buffer
fn binary_extract_bytes(buffer: &[u8], index: &mut usize, length: usize) -> RESPResult<Vec<
↪  u8>> {
    let mut output = Vec::new();

    // Check if we are allowed to read length bytes
    if *index + length > buffer.len() {
        return Err(RESPError::OutOfBounds(*index + buffer.len()));
    }

    // Copy the bytes into the output vector
    output.extend_from_slice(&buffer[*index..*index + length]);

    // Update the index
    *index += length;
    Ok(output)
}
```

The tests for this function are

src/resp.rs

```rust
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_binary_extract_bytes() {
        let buffer = "SOMEBYTES".as_bytes();
        let mut index: usize = 0;

        let output = binary_extract_bytes(buffer, &mut index, 6).unwrap();

        assert_eq!(output, "SOMEBY".as_bytes().to_vec());
        assert_eq!(index, 6);
    }

    #[test]
    fn test_binary_extract_bytes_out_of_bounds() {
        let buffer = "SOMEBYTES".as_bytes();
        let mut index: usize = 0;

        let error = binary_extract_bytes(buffer, &mut index, 10).unwrap_err();

        assert_eq!(error, RESPError::OutOfBounds(9));
        assert_eq!(index, 0);
    }

    ...

}
```

To get the length of the data contained in a RESP bulk string we need to parse the part of the buffer that contains it. As the length ends with `\r\n` we can reuse `binary_extract_line_as_string` for this purpose and convert the string into a numeric value src/resp.rs

```rust
// Extracts a single line from a RESP buffer and interprets it as length.
// The type used for the number is RESPLength.
pub fn resp_extract_length(buffer: &[u8], index: &mut usize) -> RESPResult<RESPLength> {
    let line: String = binary_extract_line_as_string(buffer, index)?;
    let length: RESPLength = line.parse()?;

    Ok(length)
}
```

This relies on the type `RESPLength` that we can define in `src/resp_result.rs` and import into `src/resp.rs`

src/resp_result.rs

```rust
type RESPLength = i32;
```

src/resp.rs

```rust
use crate::resp_result::{RESPError, RESPLength, RESPResult};
```

We cannot use `usize` here as the length of a string might be `-1` (null string). The conversion into `RESPLength` requires the trait `From<ParseIntError>` [docs] to be implemented for `RESPError`, since we are returning the error directly through the operator `?`. We also need an error variant for incorrect length values (negative values that are not `-1`).

src/resp_result.rs

```rust
use std::fmt;
use std::num;
use std::string::FromUtf8Error;

#[derive(Debug)]
pub enum RESPError {
    FromUtf8,
    IncorrectLength(RESPLength),
    OutOfBounds(usize),
    ParseInt,
    Unknown,
    WrongType,
}

impl fmt::Display for RESPError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            RESPError::FromUtf8 => write!(f, "Cannot convert from UTF-8"),
            RESPError::IncorrectLength(length) => write!(f, "Incorrect legth {}", length),
            RESPError::OutOfBounds(index) => write!(f, "Out of bounds at index {}", index),
            RESPError::ParseInt => write!(f, "Cannot parse string into integer"),
            RESPError::Unknown => write!(f, "Unknown format for RESP string"),
            RESPError::WrongType => write!(f, "Wrong parsing route for RESP type"),
        }
    }
}

...

impl From<num::ParseIntError> for RESPError {
    fn from(_err: num::ParseIntError) -> Self {
        Self::ParseInt
    }
}
```

With those functions in place, parsing a bulk string can be done with

src/resp.rs

```
fn parse_bulk_string(buffer: &[u8], index: &mut usize) -> RESPResult<RESP> {
    resp_remove_type('$', buffer, index)?;

    let length = resp_extract_length(buffer, index)?;

    if length == -1 {
        return Ok(RESP::Null);
    }

    if length < -1 {
        return Err(RESPError::IncorrectLength(length));
    }

    let bytes = binary_extract_bytes(buffer, index, length as usize)?;

    let data: String = String::from_utf8(bytes)?;

    // Increment the index to skip the \r\n
    *index += 2;

    Ok(RESP::BulkString(data))
}
```

with the relative tests

src/resp.rs

```rust
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_parse_bulk_string() {
        let buffer = "$2\r\nOK\r\n".as_bytes();
        let mut index: usize = 0;

        let output = parse_bulk_string(buffer, &mut index).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("OK")));
        assert_eq!(index, 8);
    }

    #[test]
    fn test_parse_bulk_string_empty() {
        let buffer = "$-1\r\n".as_bytes();
        let mut index: usize = 0;

        let output = parse_bulk_string(buffer, &mut index).unwrap();

        assert_eq!(output, RESP::Null);
        assert_eq!(index, 5);
    }

    #[test]
    fn test_parse_bulk_string_wrong_type() {
        let buffer = "?2\r\nOK\r\n".as_bytes();
        let mut index: usize = 0;

        let error = parse_bulk_string(buffer, &mut index).unwrap_err();

        assert_eq!(error, RESPError::WrongType);
        assert_eq!(index, 0);
    }

    #[test]
    fn test_parse_bulk_string_unparsable_length() {
        let buffer = "$wrong\r\nOK\r\n".as_bytes();
        let mut index: usize = 0;

        let error = parse_bulk_string(buffer, &mut index).unwrap_err();

        assert_eq!(error, RESPError::ParseInt);
        assert_eq!(index, 8);
    }

    #[test]
    fn test_parse_bulk_string_negative_length() {
        let buffer = "$-7\r\nOK\r\n".as_bytes();
        let mut index: usize = 0;

        let error = parse_bulk_string(buffer, &mut index).unwrap_err();

        assert_eq!(error, RESPError::IncorrectLength(-7));
        assert_eq!(index, 5);
    }

    #[test]
    fn test_parse_bulk_string_data_too_short() {
        let buffer = "$7\r\nOK\r\n".as_bytes();
        let mut index: usize = 0;

        let error = parse_bulk_string(buffer, &mut index).unwrap_err();

        assert_eq!(error, RESPError::OutOfBounds(8));
        assert_eq!(index, 4);
    }
}
```

As always, when it comes to low-level protocols, there are many more tests that we might add, but these should be enough for now. At this point `parse_bulk_string` can be added to `parser_router`

`src/resp.rs`

```
fn parser_router(
    buffer: &[u8],
    index: &mut usize,
) -> Option<fn(&[u8], &mut usize) -> RESPResult<RESP>> {
    match buffer[*index] {
        b'+' => Some(parse_simple_string),
        b'$' => Some(parse_bulk_string),
        _ => None,
    }
}
```

with a test to check that `bytes_to_resp` can parse the new type

`src/resp.rs`

```
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_bytes_to_resp_bulk_string() {
        let buffer = "$2\r\nOK\r\n".as_bytes();
        let mut index: usize = 0;

        let output = bytes_to_resp(buffer, &mut index).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("OK")));
        assert_eq!(index, 8);
    }
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.8

Step 9 - Parse a RESP array

The final piece of code that we need before we can put everything together is a function to parse a RESP array. Arrays are a tiny bit more complicated than bulk strings because they are recursive, being containers of other RESP types (including other arrays).

As before, we first add the relevant variant to the `RESP` enum

src/resp.rs

```
#[derive(Debug, PartialEq)]
pub enum RESP {
    Array(Vec<RESP>),
    BulkString(String),
    Null,
    SimpleString(String),
}

impl fmt::Display for RESP {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let data = match self {
            Self::Array(data) => {
                let mut output = String::from("*");
                output.push_str(format!("{}\r\n", data.len()).as_str());

                for elem in data.iter() {
                    output.push_str(elem.to_string().as_str());
                }

                output
            }
            Self::BulkString(data) => format!("${}\r\n{}\r\n", data.len(), data),
            Self::Null => String::from("$-1\r\n"),
            Self::SimpleString(data) => format!("+{}\r\n", data),
        };

        write!(f, "{}", data)
    }
}
```

Then, as we did for bulk strings, we need to create a function `parse_array`. As an array contains other types the function will use `parser_router` to process them src/resp.rs

```
fn parse_array(buffer: &[u8], index: &mut usize) -> RESPResult<RESP> {
    resp_remove_type('*', buffer, index)?;

    let length = resp_extract_length(buffer, index)?;

    if length < 0 {
        return Err(RESPError::IncorrectLength(length));
    }

    let mut data = Vec::new();

    for _ in 0..length {
        match parser_router(buffer, index) {
            Some(parse_func) => {
                let array_element: RESP = parse_func(buffer, index)?;
                data.push(array_element);
            }
            None => return Err(RESPError::Unknown),
        }
    }

    Ok(RESP::Array(data))
}
```

The tests are

src/resp.rs

```
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_parse_array() {
        let buffer = "*2\r\n+OK\r\n$5\r\nVALUE\r\n".as_bytes();
        let mut index: usize = 0;

        let output = parse_array(buffer, &mut index).unwrap();

        assert_eq!(
            output,
            RESP::Array(vec![
                RESP::SimpleString(String::from("OK")),
                RESP::BulkString(String::from("VALUE"))
            ])
        );
        assert_eq!(index, 20);
    }
    #[test]
    fn test_parse_array_invalid_length() {
        let buffer = "*-1\r\n+OK\r\n$5\r\nVALUE\r\n".as_bytes();
        let mut index: usize = 0;

        let error = parse_array(buffer, &mut index).unwrap_err();

        assert_eq!(error, RESPError::IncorrectLength(-1));
        assert_eq!(index, 5);
    }
}
```

Finally, let's add `parse_array` itself to `parser_router`

src/resp.rs

```
fn parser_router(
    buffer: &[u8],
    index: &mut usize,
) -> Option<fn(&[u8], &mut usize) -> RESPResult<RESP>> {
    match buffer[*index] {
        b'+' => Some(parse_simple_string),
        b'$' => Some(parse_bulk_string),
        b'*' => Some(parse_array),
        _ => None,
    }
}
```

With a test to check the behaviour of `bytes_to_resp`

src/resp.rs

```
#[cfg(test)]
mod tests {

    ...

    #[test]
    fn test_bytes_to_resp_array() {
        let buffer = "*2\r\n+OK\r\n$5\r\nVALUE\r\n".as_bytes();
        let mut index: usize = 0;

        let output = bytes_to_resp(buffer, &mut index).unwrap();

        assert_eq!(
            output,
            RESP::Array(vec![
                RESP::SimpleString(String::from("OK")),
                RESP::BulkString(String::from("VALUE"))
            ])
        );
        assert_eq!(index, 20);
    }
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.9

Step 10 - Process PING the right way

We can at this point refactor the part of the server that processes an incoming request, parses it, and executes the command. Redis commands are sent as arrays of bulk strings, so it's a good idea to have a generic processor of requests to simplify the addition of new commands in the future.

Let's start creating a result type for our server in a new file, `src/server.rs`

src/server.rs

```
use std::fmt;

#[derive(Debug, PartialEq)]
pub enum ServerError {
    CommandError,
}

impl fmt::Display for ServerError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            ServerError::CommandError => write!(f, "Error while processing!"),
        }
    }
}

pub type ServerResult<T> = Result<T, ServerError>;
```

For now, a single variant will be sufficient. Later, when different error cases emerge, we will add more.

We need to add the module to `main.rs` to make it visible.

src/main.rs

```
mod resp;
mod resp_result;
mod server;
```

We can then create a function to process an incoming RESP request. This function needs to parse the incoming RESP data, figure out which command it contains, and call the right function to process it.

src/server.rs

```
use crate::RESP;

...

pub fn process_request(request: RESP) -> ServerResult<RESP> {
    let elements = match request {
        RESP::Array(v) => v,
        _ => {
            return Err(ServerError::CommandError);
        }
    };

    let mut command = Vec::new();
    for elem in elements.iter() {
        match elem {
            RESP::BulkString(v) => command.push(v),
            _ => {
                return Err(ServerError::CommandError);
            }
        }
    }

    match command[0].to_lowercase().as_str() {
        "ping" => Ok(RESP::SimpleString(String::from("PONG"))),
        _ => {
            return Err(ServerError::CommandError);
        }
    }
}
```

We accept only a `RESP::Array`, and from it we can extract the inner `Vec<RESP>` and verify that each element is a `RESP::BulkString`. At that point the vector `command` contains each element of the command that was sent (e.g. `"PING"` or `"ECHO 42"`, where each element is a Rust `String`). We can then route the processing according to the first element of the vector.

The relative tests are

src/server.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_process_request_ping() {
        let request = RESP::Array(vec![RESP::BulkString(String::from("PING"))]);

        let output = process_request(request).unwrap();
        assert_eq!(output, RESP::SimpleString(String::from("PONG")));
    }

    #[test]
    fn test_process_request_not_array() {
        let request = RESP::BulkString(String::from("PING"));

        let error = process_request(request).unwrap_err();

        assert_eq!(error, ServerError::CommandError);
    }

    #[test]
    fn test_process_request_not_bulkstrings() {
        let request = RESP::Array(vec![RESP::SimpleString(String::from("PING"))]);

        let error = process_request(request).unwrap_err();

        assert_eq!(error, ServerError::CommandError);
    }
}
```

The last step is to call `process_request` from `handle_connection`

src/main.rs

```
use crate::resp::{bytes_to_resp, RESP};
use crate::server::process_request;

...

async fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    loop {
        match stream.read(&mut buffer).await {
            Ok(size) if size != 0 => {
                let mut index: usize = 0;

                let request = match bytes_to_resp(&buffer[..size].to_vec(), &mut index) {
                    Ok(v) => v,
                    Err(e) => {
                        eprintln!("Error: {}", e);
                        return;
                    }
                };

                let response = match process_request(request) {
                    Ok(v) => v,
                    Err(e) => {
                        eprintln!("Error parsing command: {}", e);
                        return;
                    }
                };

                if let Err(e) = stream.write_all(response.to_string().as_bytes()).await {
                    eprintln!("Error writing to socket: {}", e);
                }
            }
            Ok(_) => {
                println!("Connection closed");
                break;
            }
            Err(e) => {
                println!("Error: {}", e);
                break;
            }
        }
    }
}
```

CodeCrafters

Stage 4: Handle concurrent clients

Everything we did in this chapter was in preparation for future changes, but the last refactoring altered code that was actually used by the server. However, as we haven't changed the behaviour, this version of the code still passes Stage 4 of the CodeCrafters challenge.

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.10

Step 11 - Process ECHO

All the work we have done in this chapter was meant to simplify our job when it comes to adding new commands. We will see that in action now.

Let's add the command `ECHO`, that will return its argument to the client as a bulk string. As the logic behind `ECHO` is trivial, the only thing we need to do is to add a new arm to the `match` construct in `process_request`

src/server.rs

```rust
pub fn process_request(request: RESP) -> ServerResult<RESP> {
    let elements = match request {
        RESP::Array(v) => v,
        _ => {
            return Err(ServerError::CommandError);
        }
    };

    let mut command = Vec::new();
    for elem in elements.iter() {
        match elem {
            RESP::BulkString(v) => command.push(v),
            _ => {
                return Err(ServerError::CommandError);
            }
        }
    }

    match command[0].to_lowercase().as_str() {
        "ping" => Ok(RESP::SimpleString(String::from("PONG"))),
        "echo" => Ok(RESP::BulkString(command[1].clone())),
        _ => {
            return Err(ServerError::CommandError);
        }
    }
}
```

with this test

src/server.rs

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_process_request_ping() {
        let request = RESP::Array(vec![RESP::BulkString(String::from("PING"))]);

        let output = process_request(request).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("PONG")));
    }

    #[test]
    fn test_process_request_echo() {
        let request = RESP::Array(vec![
            RESP::BulkString(String::from("ECHO")),
            RESP::BulkString(String::from("42")),
        ]);

        let output = process_request(request).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("42")));
    }
```

CodeCrafters

Stage 5: Implement the ECHO command

At this point your code should pass Stage 5 of the CodeCrafters challenge.

Source code

https://github.com/lgiordani/sider/tree/ed1/step2.11

Chapter 3

GET and SET

We will never find the command that Nedry used. He's covered his tracks far too well.
Jurassic Park (1993)

The goal of this chapter is to implement the two commands `GET` and `SET`, arguably the core of a key/value store like Redis. We will go through the basic implementation of the storage in memory, and we won't implement persistence yet (writing data on the file system). However, in the next chapter we will implement key expiry, both passive and active.

The features we will add in this chapter are going to increase the complexity of the system, and it will become increasingly clear that we need to change our approach. In the next chapter we will therefore refactor the whole system into a set of independent asynchronous tasks. This change is paramount to make it possible to tackle replication and transactions in an elegant way.

Step 1 - Create the storage manager

In order to properly manage storage[1], it's important to have a data type that allows us to manipulate data in a simple way. We will then introduce a struct `Storage` that captures the stored data and the low-level commands that the system exposes.

Let's start as usual creating a new file with a nicely defined result type

`src/storage_result.rs`

```rust
use std::fmt;

#[derive(Debug)]
pub enum StorageError {
    IncorrectRequest,
    CommandNotAvailable(String),
}

impl fmt::Display for StorageError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            StorageError::IncorrectRequest => {
                write!(f, "The client sent an incorrect request!")
            }
            StorageError::CommandNotAvailable(c) => {
                write!(f, "The requested command {} is not available!", c)
            }
        }
    }
}

pub type StorageResult<T> = Result<T, StorageError>;
```

We can replace `ServerError::CommandError` with `StorageError` variants. This will keep things coherent from the point of view of result types. We can keep `ServerError`, as it will be useful in the future to manage other types of errors in the server.

The type `Storage` is at the moment just a wrapper around a `std::collections::HashMap` that contains instances of the enum `StorageValue`. This allows us to decouple the storage from the nature of the data itself.

[1]The term *storage* here means the part of the system that manages the actual data. For now, it will be implemented in memory, so there is no persistence.

src/storage.rs

```rust
use std::collections::HashMap;

#[derive(Debug, PartialEq)]
pub enum StorageValue {
    String(String),
}

pub struct Storage {
    store: HashMap<String, StorageValue>,
}

impl Storage {
    pub fn new() -> Self {
        let store: HashMap<String, StorageValue> = HashMap::new();

        Self { store: store }
    }
}
```

As usual, we need to add the new modules to `main.rs`

src/main.rs

```rust
mod resp;
mod resp_result;
mod server;
mod storage;
mod storage_result;
```

We will move the two commands `PING` and `ECHO` into the storage, and define `GET` and `SET` there as well. In later chapters we will move the code of those commands again, but for the time being the storage looks like a good place to host them.

src/storage.rs

```rust
use crate::resp::RESP;
use crate::storage_result::{StorageError, StorageResult};
use std::collections::HashMap;

...

impl Storage {
     ...

     pub fn process_command(&mut self, command: &Vec<String>) -> StorageResult<RESP> {
        match command[0].to_lowercase().as_str() {
            "ping" => self.command_ping(&command),
            "echo" => self.command_echo(&command),
            _ => Err(StorageError::CommandNotAvailable(command[0].clone())),
        }
    }

    fn command_ping(&self, _command: &Vec<String>) -> StorageResult<RESP> {
        Ok(RESP::SimpleString("PONG".to_string()))
    }

    fn command_echo(&self, command: &Vec<String>) -> StorageResult<RESP> {
        Ok(RESP::BulkString(command[1].clone()))
    }

    ...

}
```

There are three tests for this module. Two of them, `test_command_ping` and `test_command_echo` have been moved here from `src/server.rs`

`src/storage.rs`

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_create_new() {
        let storage: Storage = Storage::new();

        assert_eq!(storage.store.len(), 0);
    }

    #[test]
    fn test_command_ping() {
        let command = vec![String::from("ping")];
        let storage: Storage = Storage::new();

        let output = storage.command_ping(&command).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("PONG")));
    }

    #[test]
    fn test_command_ping_uppercase() {
        let command = vec![String::from("PING")];
        let storage: Storage = Storage::new();
        let output = storage.command_ping(&command).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("PONG")));
    }

    #[test]
    fn test_command_echo() {
        let command = vec![String::from("echo"), String::from("42")];
        let storage: Storage = Storage::new();

        let output = storage.command_echo(&command).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("42")));
    }
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step3.1

Step 2 - Use the storage manager

We need now to actually instantiate and use the storage manager. This introduces a new problem, that of passing values to asynchronous tasks. It is a problem that has many ramifications, and in this chapter we will see only some of them.

In Rust, a variable that is passed to a function is *owned* by that function. This, practically speaking, means that we cannot instantiate a variable of type `Storage` and then pass it to each connection that we create. When we pass it to the first connection the ownership is gone.

For example

src/main.rs [NOT WORKING]

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    let mut storage = Storage::new();

    loop {
        match listener.accept().await {
            Ok((stream, _)) => {
                tokio::spawn(handle_connection(stream, storage));
            }
            Err(e) => {
                println!("Error: {}", e);
                continue;
            }
        }
    }
}

async fn handle_connection(mut stream: TcpStream, mut storage: Storage) {
    ...
```

This code is an attempt to create the storage and then pass it to the connection handlers. The compiler however will not accept it

```
19 |     let mut storage = Storage::new();
   |         ----------- move occurs because `storage` has type `storage::Storage`,
   |                     which does not implement the `Copy` trait
20 |
21 |     loop {
   |     ---- inside of this loop
...
24 |                 tokio::spawn(handle_connection(stream, storage));
   |                                                        ^^^^^^^ value moved here,
   |                                                                in previous iteration
   |                                                                of loop
```

This behaviour comes directly from the language rules: there cannot be more than one owner.

We might consider passing a reference, which works as long as we don't need to mutate the referenced value, because there can be only one mutable reference alive at a time. For example

src/main.rs [NOT WORKING]

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    let mut storage = Storage::new();

    loop {
        match listener.accept().await {
            Ok((stream, _)) => {
                tokio::spawn(handle_connection(stream, &mut storage));
            }
            Err(e) => {
                println!("Error: {}", e);
                continue;
            }
        }
    }
}

async fn handle_connection(mut stream: TcpStream, storage: &mut Storage) {
    ...
```

The compiler is indeed still not happy with this solution

```
error[E0499]: cannot borrow `storage` as mutable more than once at a time
  --> src/main.rs:24:56
   |
24 |                 tokio::spawn(handle_connection(stream, &mut storage));
   |                              --------------------------^^^^^^^^^^^^-
   |                              |                          |
   |                              |                          `storage` was mutably borrowed
   |                              |                          here in the previous iteration
   |                              |                          of the loop
   |                              argument requires that `storage` is borrowed
   |                              for `'static`
```

Having references that allow multiple concurrent *reads* is perfectly acceptable, but there should be only one reference that allows to *write* (mutate).

We actually have two problems to solve here:

1. grant multiple tasks access to the same resource, allowing them to both read and write.
2. make sure that the resource is always valid, forbidding one of the tasks to drop it.

The first problem can be solved using a classic locking mechanism. A perfect structure for such a job is `std::sync::Mutex` [docs], as it allows multiple tasks to work on the same resources creating and releasing exclusive access locks.

A way to address the second problem is to use a reference counter. Such a structure keeps count of how many references exist for a resource, and drops it only when the number of references is zero. An implementation of this concept can be found in `std::sync::Arc` [docs], which increments reference counts atomically. This last requirements is important because asynchronous tasks are eventually run using threads, so it is important for operations to be thread-safe. Incrementing a value like a reference count in a non-atomic way is the classic example of an unsafe concurrent operation.

Reference counting and race conditions

Reference counting is one of the most important techniques used to manage memory allocation, and you might want to learn a bit about its pros and cons and the relationship between it and garbage collection algorithms.

If you plan to write concurrent code, you need to be familiar with the concept of race condition, which is one of many issues that poorly written multithreaded code can run into.

Practically speaking, this means that we can use `Arc<Mutex<T>>` to share access to a single value of type `T`. Variables of type `Arc` can be cloned to increment the reference count and whoever has a reference can try to lock the resource using `Mutex::lock` [docs], eventually receiving exclusive access.

`src/main.rs`

```rust
use crate::resp::{bytes_to_resp, RESP};
use crate::server::process_request;
use crate::storage::Storage;
use std::sync::{Arc, Mutex};
use tokio::{
    io::{AsyncReadExt, AsyncWriteExt},
    net::{TcpListener, TcpStream},
};

...

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    let storage = Arc::new(Mutex::new(Storage::new()));

    loop {
        match listener.accept().await {
            Ok((stream, _)) => {
                tokio::spawn(handle_connection(stream, storage.clone()));
            }
            Err(e) => {
                println!("Error: {}", e);
                continue;
            }
        }
    }
}

async fn handle_connection(mut stream: TcpStream, storage: Arc<Mutex<Storage>>) {
    let mut buffer = [0; 512];

    loop {
        match stream.read(&mut buffer).await {
            Ok(size) if size != 0 => {
                let mut index: usize = 0;

                let request = match bytes_to_resp(&buffer[..size].to_vec(), &mut index) {
                    Ok(v) => v,
                    Err(e) => {
                        eprintln!("Error: {}", e);
                        return;
                    }
                };

                let response = match process_request(request, storage.clone()) {
                    Ok(v) => v,
                    Err(e) => {
                        eprintln!("Error parsing command: {}", e);
                        return;
                    }
                };

                if let Err(e) = stream.write_all(response.to_string().as_bytes()).await {
                    eprintln!("Error writing to socket: {}", e);
                }
            }
            Ok(_) => {
                println!("Connection closed");
                return;
            }
            Err(e) => {
                eprintln!("Error: {}", e);
                return;
            }
        }
    }
}
```

As you can see, we are just passing an additional parameter to `handle_connection` and to `process_request`. The `Arc` is cloned in the main loop, and this increments the reference count that keeps the resource alive until every task is completed.

In the function `process_request` we need to move from `ServerError::CommandError` to the newly defined `StorageError::IncorrectRequest`. At the end of the function we also lock and use the storage.

src/server.rs

```
use crate::storage::Storage;
use crate::storage_result::{StorageError, StorageResult};
use crate::RESP;
use std::fmt;
use std::sync::{Arc, Mutex};

...

pub fn process_request(request: RESP, storage: Arc<Mutex<Storage>>) -> StorageResult<RESP>
↪ {
    let elements = match request {
        RESP::Array(v) => v,
        _ => {
            return Err(StorageError::IncorrectRequest);
        }
    };

    let mut command = Vec::new();
    for elem in elements.iter() {
        match elem {
            RESP::BulkString(v) => command.push(v.clone()),
            _ => {
                return Err(StorageError::IncorrectRequest);
            }
        }
    }

    let mut guard = storage.lock().unwrap();
    let response = guard.process_command(&command);
    response
}
```

Please note that the vector `command` now needs a clone of the bulk string because we are eventually transferring its ownership to `storage.process_command`.

We also need to change the tests accordingly

src/server.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_process_request_ping() {
        let request = RESP::Array(vec![RESP::BulkString(String::from("PING"))]);
        let storage = Arc::new(Mutex::new(Storage::new()));

        let output = process_request(request, storage).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("PONG")));
    }

    #[test]
    fn test_process_request_echo() {
        let request = RESP::Array(vec![
            RESP::BulkString(String::from("ECHO")),
            RESP::BulkString(String::from("42")),
        ]);
        let storage = Arc::new(Mutex::new(Storage::new()));

        let output = process_request(request, storage).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("42")));
    }

    #[test]
    fn test_process_request_not_array() {
        let request = RESP::BulkString(String::from("PING"));
        let storage = Arc::new(Mutex::new(Storage::new()));
        let error = process_request(request, storage).unwrap_err();

        assert_eq!(error, StorageError::IncorrectRequest);
    }

    #[test]
    fn test_process_request_not_bulkstrings() {
        let request = RESP::Array(vec![RESP::SimpleString(String::from("PING"))]);
        let storage = Arc::new(Mutex::new(Storage::new()));

        let error = process_request(request, storage).unwrap_err();

        assert_eq!(error, StorageError::IncorrectRequest);
    }
}
```

and to run those we need to add `PartialEq` to `StorageError`

`src/storage_result.rs`

```
#[derive(Debug, PartialEq)]
pub enum StorageError {
    IncorrectRequest,
    CommandNotAvailable(String),
}
```

CodeCrafters

Stage 5: Implement the ECHO command

Once again the code we wrote was in preparation for a future change, so the previous behaviour should be unchanged. This version of the code still passes Stage 5 of the CodeCrafters challenge.

Source code

https://github.com/lgiordani/sider/tree/ed1/step3.2

Step 3 - Implement GET and SET

The basic implementation of `GET` and `SET` is extremely simple. After all we are basically building a wrapper around a dictionary, so we can directly add two methods `get` and `set` to the struct `Storage`

`src/storage.rs`

```
impl Storage {

    ...

    fn set(&mut self, key: String, value: String) -> StorageResult<String> {
        self.store.insert(key, StorageValue::String(value));

        Ok(String::from("OK"))
    }

    fn get(&self, key: String) -> StorageResult<Option<String>> {
        match self.store.get(&key) {
            Some(StorageValue::String(v)) => return Ok(Some(v.clone())),
            None => return Ok(None),
        }
    }
}
```

Here, we are simply using the underlying methods `HashMap::insert` [docs] and `HashMap::get` [docs]. The tests for the functions we wrote are

`src/storage.rs`

```
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_set_value() {
        let mut storage: Storage = Storage::new();
        let avalue = StorageValue::String(String::from("avalue"));

        let output = storage
            .set(String::from("akey"), String::from("avalue"))
            .unwrap();

        assert_eq!(output, String::from("OK"));
        assert_eq!(storage.store.len(), 1);
        match storage.store.get(&String::from("akey")) {
            Some(value) => assert_eq!(value, &avalue),
            None => panic!(),
        }
    }

    #[test]
    fn test_get_value() {
        let mut storage: Storage = Storage::new();
        storage.store.insert(
            String::from("akey"),
            StorageValue::String(String::from("avalue")),
        );

        let result = storage.get(String::from("akey")).unwrap();

        assert_eq!(storage.store.len(), 1);
        assert_eq!(result, Some(String::from("avalue")));
    }

    #[test]
    fn test_get_value_key_does_not_exist() {
        let storage: Storage = Storage::new();

        let result = storage.get(String::from("akey")).unwrap();

        assert_eq!(storage.store.len(), 0);
        assert_eq!(result, None);
    }

    ...

}
```

We need to wrap those methods in order to expose them as commands. Wrappers are important mostly for error management. For example, it's in the wrappers that we need to consider syntax errors in the commands passed by users.

src/storage.rs

```
impl Storage {

    ...

    fn command_set(&mut self, command: &Vec<String>) -> StorageResult<RESP> {
        if command.len() != 3 {
            return Err(StorageError::CommandSyntaxError(command.join(" ")));
        }

        let _ = self.set(command[1].clone(), command[2].clone());
        Ok(RESP::SimpleString(String::from("OK")))
    }

    fn command_get(&mut self, command: &Vec<String>) -> StorageResult<RESP> {
        if command.len() != 2 {
            return Err(StorageError::CommandSyntaxError(command.join(" ")));
        }

        let output = self.get(command[1].clone());

        match output {
            Ok(Some(v)) => Ok(RESP::BulkString(v)),
            Ok(None) => Ok(RESP::Null),
            Err(_) => Err(StorageError::CommandInternalError(command.join(" "))),
        }
    }

    ...

}
```

The tests for the wrappers are

src/storage.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_process_command_set() {
        let mut storage: Storage = Storage::new();
        let command = vec![
            String::from("set"),
            String::from("key"),
            String::from("value"),
        ];

        let output = storage.process_command(&command).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("OK")));
        assert_eq!(storage.store.len(), 1);
    }

    #[test]
    fn test_process_command_get() {
        let mut storage: Storage = Storage::new();
        storage.store.insert(
            String::from("akey"),
            StorageValue::String(String::from("avalue")),
        );
        let command = vec![String::from("get"), String::from("akey")];

        let output = storage.process_command(&command).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("avalue")));
        assert_eq!(storage.store.len(), 1);
    }

    ...

}
```

The code relies on two new errors

src/storage_result.rs

```rust
#[derive(Debug)]
pub enum StorageError {
    IncorrectRequest,
    CommandNotAvailable(String),
    CommandSyntaxError(String),
    CommandInternalError(String),
}

impl fmt::Display for StorageError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            StorageError::IncorrectRequest => {
                write!(f, "The client sent an incorrect request!")
            }
            StorageError::CommandNotAvailable(c) => {
                write!(f, "The requested command {} is not available!", c)
            }
            StorageError::CommandSyntaxError(string) => {
                write!(f, "Syntax error while processing {}!", string)
            }
            StorageError::CommandInternalError(string) => {
                write!(f, "Internal error while processing {}!", string)
            }
        }
    }
}
```

and the tests use `process_command`, so we need to add the two new commands to that function

src/storage.rs

```rust
impl Storage {

    ...

    pub fn process_command(&mut self, command: &Vec<String>) -> StorageResult<RESP> {
        match command[0].to_lowercase().as_str() {
            "ping" => self.command_ping(&command),
            "echo" => self.command_echo(&command),
            "get" => self.command_get(&command),
            "set" => self.command_set(&command),
            _ => Err(StorageError::CommandNotAvailable(command[0].clone())),
        }
    }

    ...

}
```

CodeCrafters

Stage 6: Implement the SET & GET commands

It's time to update the testing suite and to check that this version of the code passes Stage 6 of the CodeCrafters challenge.

Source code

https://github.com/lgiordani/sider/tree/ed1/step3.3

Chapter 4

Key Expiry

No code, no riddle, no fancy little countdown.
Die Hard: With a Vengeance (1995)

Things starts to become interesting, and a little bit convoluted, with expiry.

Redis supports two types of key expiry, passive and active. With passive expiry, a key is is checked upon retrieval. A `GET` operation might thus find a key but discover that it's expired and remove it before returning it. With active expiry, the server periodically scans keys and removes the expired ones.

From the end user's point of view the two mechanisms produce the same result: keys created with an explicit expiry time will eventually be removed. At the end of the chapter we will be able to send commands like `SET answer 42 EX 5` and see the key `answer` disappear after 5 seconds.

The reason why things start to become interesting is that in this chapter we will add a component to the system that goes beyond the simple request/response model we implemented so far. The active expiry mechanism requires an independent process that is a reaction to an internal timeout rather than to a client request.

We will come up with a working solution, but we will also discuss the limits of the current architecture. In the next chapter we will go through a major refactoring that will introduce a much more powerful structure based on actors.

For now, we will implement the following requirements:

- Add creation time and expiry to stored data.

- Keep a list of keys whose expiry time has been set.
- Check the expiry time of retrieved keys.
- Periodically scan the list of expiring keys and remove the ones whose time is up.

Step 1 - Creation time and expiry

We can start adding the creation time and expiry to the data contained inside the storage. To do this we need to define a struct that represents stored data

src/storage.rs

```rust
use std::time::{Duration, SystemTime};

...

#[derive(Debug)]
pub struct StorageData {
    pub value: StorageValue,
    pub creation_time: SystemTime,
    pub expiry: Option<Duration>,
}
```

Here, we are using the two types `SystemTime` [docs] (creation time, absolute) and `Duration` [docs] (expiry, relative).

We can then create a method `add_expiry` and a function to create `StorageData` from a `String`

src/storage.rs

```rust
impl StorageData {
    pub fn add_expiry(&mut self, expiry: Duration) {
        self.expiry = Some(expiry);
    }
}

impl From<String> for StorageData {
    fn from(s: String) -> StorageData {
        StorageData {
            value: StorageValue::String(s),
            creation_time: SystemTime::now(),
            expiry: None,
        }
    }
}
```

To make sure that we can check the equality between two pieces of data we need to implement `PartialEq` [docs]

```
src/storage.rs

impl PartialEq for StorageData {
    fn eq(&self, other: &Self) -> bool {
        self.value == other.value && self.expiry == other.expiry
    }
}
```

Last, we need to use the new data structure in the storage

```
src/storage.rs

pub struct Storage {
    store: HashMap<String, StorageData>,
}
```

Which requires a straightforward set of changes in the rest of the code

```
src/storage.rs

impl Storage {
    pub fn new() -> Self {
        let store: HashMap<String, StorageData> = HashMap::new();

        Self { store: store }
    }

    ...

    fn set(&mut self, key: String, value: String) -> StorageResult<String> {
        self.store.insert(key, StorageData::from(value));

        Ok(String::from("OK"))
    }

    fn get(&self, key: String) -> StorageResult<Option<String>> {
        match self.store.get(&key) {
            Some(StorageData {
                value: StorageValue::String(v),
                creation_time: _,
                expiry: _,
            }) => return Ok(Some(v.clone())),
            None => return Ok(None),
        }
    }
    ...
}

#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_set_value() {
        let mut storage: Storage = Storage::new();
        let avalue = StorageData::from(String::from("avalue"));

        let output = storage
            .set(String::from("akey"), String::from("avalue"))
            .unwrap();

        assert_eq!(output, String::from("OK"));
        assert_eq!(storage.store.len(), 1);
        match storage.store.get(&String::from("akey")) {
            Some(value) => assert_eq!(value, &avalue),
            None => panic!(),
        }
    }

    #[test]
    fn test_get_value() {
        let mut storage: Storage = Storage::new();
        storage.store.insert(
            String::from("akey"),
            StorageData::from(String::from("avalue")),
        );

        let result = storage.get(String::from("akey")).unwrap();

        assert_eq!(storage.store.len(), 1);
        assert_eq!(result, Some(String::from("avalue")));
    }

    #[test]
    fn test_process_command_get() {
        let mut storage: Storage = Storage::new();
        storage.store.insert(
            String::from("akey"),
            StorageData::from(String::from("avalue")),
        );
        let command = vec![String::from("get"), String::from("akey")];

        let output = storage.process_command(&command).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("avalue")));
        assert_eq!(storage.store.len(), 1);
    }

    ...
```

```
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step4.1

Step 2 - Storage support for expiry

At this point we need to add support for expiry into the `Storage` struct. We need to create a function `expire_keys` that is triggered periodically and whose task is to decide if expiring keys are still valid or not.

A simple way to speed up the execution of such an operation is to keep a separate account of keys with an expiry, so that we don't need to scan all the keys stored in the system every time we trigger `expire_keys`. This structure has to be a `HashMap` as it needs to contain both the key and the expiry time.

`src/storage.rs`

```rust
pub struct Storage {
    store: HashMap<String, StorageData>,
    expiry: HashMap<String, SystemTime>,
    active_expiry: bool,
}
```

The idea behind the flag `active_expiry` is to allow the process to be halted. Keep in mind that this is not a perfect implementation of what happens in Redis, but a parallel implementation of similar concepts, so this might or might not be what a Redis cluster actually allows you to do.

The new fields have to be initialised when a value is created

`src/storage.rs`

```rust
impl Storage {

    ...

    pub fn new() -> Self {
        let store: HashMap<String, StorageData> = HashMap::new();

        Self {
            store: store,
            expiry: HashMap::<String, SystemTime>::new(),
            active_expiry: true,
        }
    }

    ...

}
```

and the creation test changes accordingly

src/storage.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_create_new() {
        let storage: Storage = Storage::new();

        assert_eq!(storage.store.len(), 0);
        assert_eq!(storage.expiry.len(), 0);
        assert_eq!(storage.expiry, HashMap::<String, SystemTime>::new());
        assert!(storage.active_expiry);
    }

    ...

}
```

Finally we can create two methods `set_active_expiry` and `expire_keys`, where the latter is the function that we want to run periodically to clean up expired keys.

src/storage.rs

```
impl Storage {

    ...

    pub fn set_active_expiry(&mut self, value: bool) {
        self.active_expiry = value;
    }

    pub fn expire_keys(&mut self) {
        if !self.active_expiry {
            return;
        }

        let now = SystemTime::now();

        let expired_keys: Vec<String> = self
            .expiry
            .iter()
            .filter_map(|(key, &value)| if value < now { Some(key.clone()) } else { None })
            .collect();
        for k in expired_keys {
            self.store.remove(&k);
            self.expiry.remove(&k);
        }
    }
}
```

The tests for these two functions are

src/storage.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_expire_keys() {
        let mut storage: Storage = Storage::new();

        storage
            .set(String::from("akey"), String::from("avalue"))
            .unwrap();

        storage.expiry.insert(
            String::from("akey"),
            SystemTime::now() - Duration::from_secs(5),
        );

        storage.expire_keys();
        assert_eq!(storage.store.len(), 0);
    }

    #[test]
    fn test_expire_keys_deactivated() {
        let mut storage: Storage = Storage::new();
        storage.set_active_expiry(false);

        storage
            .set(String::from("akey"), String::from("avalue"))
            .unwrap();

        storage.expiry.insert(
            String::from("akey"),
            SystemTime::now() - Duration::from_secs(5),
        );

        storage.expire_keys();
        assert_eq!(storage.store.len(), 1);
    }
    ...
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step4.2

Step 3 - Run a function periodically

So far the structure of the code is pretty linear, despite the fact that we are using asynchronous code. Every time a new client connection is established, the server spawns a new connection handler task that receives a reference to the storage.

When the task needs to interact with the storage it can create a lock and access the resource in an exclusive fashion. Creating a lock on the whole storage regardless of the nature of the operation is clearly suboptimal, but we won't dig into performance optimisation in this project.

Now we need to periodically run the method `expire_keys` of the storage. The best way to do it is to set up a timer that will run the function every given amount of time.

To do this we first create an asynchronous function that locks the storage and runs the method.

src/main.rs

```
async fn expire_keys(storage: Arc<Mutex<Storage>>) {
    let mut guard = storage.lock().unwrap();

    guard.expire_keys();
}
```

Now we can add an asynchronous timer to the main loop and have this function called every 10 milliseconds. In a production system this value would be fetched from a configuration file, but in this case we will just hard code it.

The current loop in `main` relies on the fact that there is only one asynchronous source of events, that is `listener.accept()`

src/main.rs

```
#[tokio::main]
async fn main() -> std::io::Result<()> {

    ...

    loop {
        match listener.accept().await {

            ...

        }
    }
    ...
}
```

but this is not the case any more, as we have both the listener and the timer to await. This is the perfect job for `select!` [docs], so we'll first introduce it in the current version of `main`

src/main.rs

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    let storage = Arc::new(Mutex::new(Storage::new()));

    loop {
        tokio::select! {
            connection = listener.accept() => {
                match connection {
                    Ok((stream, _)) => {
                        tokio::spawn(handle_connection(stream, storage.clone()));
                    }
                    Err(e) => {
                        println!("Error: {}", e);
                        continue;
                    }
                }
            }
        }
    }
}
```

and at this point we can easily add the new task

src/main.rs

```rust
use std::time::Duration;

...

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    let storage = Arc::new(Mutex::new(Storage::new()));
    let mut interval_timer = tokio::time::interval(Duration::from_millis(10));

    loop {
        tokio::select! {
            connection = listener.accept() => {
                match connection {
                    Ok((stream, _)) => {
                        tokio::spawn(handle_connection(stream, storage.clone()));
                    }
                    Err(e) => {
                        println!("Error: {}", e);
                        continue;
                    }
                }
            }

            _ = interval_timer.tick() => {
                tokio::spawn(expire_keys(storage.clone()));
            }
        }
    }
}
```

Blocking functions

Please note that `expire_keys` is an asynchronous function but its true nature is that of a blocking one. The method `expire_keys` *might* in theory run for a very long time, thus making the whole asynchronous assumption false. In this project we will not address this problem, but keep in mind that a production system should.

Should you want to try, a simple improvement is to make the function remove only a certain amount of keys to keep execution time short.

Let's pause for a moment to consider what we have done here, as it will be crucial in the next chapter. Since the function `expire_keys` is triggered by an internal timer, the assumption that the server waits for incoming connections only is not valid any more. There might be several events that we need to process in `select!`, and each one of them needs to receive a protected copy of the shared resources it affects (clone of `Arc<Mutex<T>>`).

Source code

`https://github.com/lgiordani/sider/tree/ed1/step4.3`

Step 4 - SET parameters

To be able to add an expiry time to keys we need to change the way the command `SET` works. As there are many parameters supported by this command it's clear that we need a separated function to parse them. This is the first symptom that commands should be more than just methods of the struct `Storage`, but for now we won't refactor anything on that side.

Command line parsing is a complicated and messy process, and generally it's better to rely on specific libraries. In this case, however, it might make sense to implement something custom since the `SET` command is not a traditional shell command with long and short options preceded by hyphens.

We will add support for the options `EX`, `PX`, `NX`, `XX`, and `GET`. There are some requirements:

- `NX` and `XX` are mutually exclusive
- `EX` and `PX` are mutually exclusive
- `EX` and `PX` have to be followed by an integer (respectively seconds and milliseconds)
- options can be listed in any order

We can start creating a new file called `src/set.rs`, and defining some enums and a struct that represent the possible settings

`src/set.rs`

```
#[derive(Debug, PartialEq)]
pub enum KeyExistence {
    NX,
    XX,
}

#[derive(Debug, PartialEq)]
pub enum KeyExpiry {
    EX(u64),
    PX(u64),
}

#[derive(Debug, PartialEq)]
pub struct SetArgs {
    pub expiry: Option<KeyExpiry>,
    pub existence: Option<KeyExistence>,
    pub get: bool,
}

impl SetArgs {
    pub fn new() -> Self {
        SetArgs {
            expiry: None,
            existence: None,
            get: false,
        }
    }
}
```

Enums are a good way to represent mutually exclusive options like `NX` and `XX` (similar to a boolean `OR`), while a struct is great to group values (similar to a boolean `AND`).

We also need to add the new file as a module

src/main.rs

```rust
mod resp;
mod resp_result;
mod server;
mod set;
mod storage;
mod storage_result;
```

Then we can create a function `parse_set_arguments` that receives a `Vec<String>` and creates a `SetArgs`. It makes sense to reuse `StorageResult` as a result type, since the parsing happens in the storage.

src/set.rs

```rust
use crate::storage_result::{StorageError, StorageResult};

...

pub fn parse_set_arguments(arguments: &Vec<String>) -> StorageResult<SetArgs> {
    let mut args = SetArgs::new();

    ...

    Ok(args)
}
```

Inside that function we need to loop on the input vector and check if strings correspond to commands. We cannot strictly iterate on them because `EX` and `PX` are followed by an argument, so `loop` is the best solution.

Basic structure and the argument `NX`

This is the implementation that matches `NX`

src/set.rs

```rust
pub fn parse_set_arguments(arguments: &Vec<String>) -> StorageResult<SetArgs> {
    let mut args = SetArgs::new();

    let mut idx: usize = 0;

    loop {
        if idx >= arguments.len() {
            break;
        }

        match arguments[idx].to_lowercase().as_str() {
            "nx" => {
                if args.existence == Some(KeyExistence::XX) {
                    return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
                }

                args.existence = Some(KeyExistence::NX);

                idx += 1;
            }
            _ => {
                return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
            }
        }
    }

    Ok(args)
}
```

with the following tests

src/set.rs

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_parse_nx() {
        let commands: Vec<String> = vec![String::from("NX")];

        let args = parse_set_arguments(&commands).unwrap();

        assert_eq!(args.existence, Some(KeyExistence::NX));
    }

    #[test]
    fn test_parse_nx_lowercase() {
        let commands: Vec<String> = vec![String::from("nx")];

        let args = parse_set_arguments(&commands).unwrap();

        assert_eq!(args.existence, Some(KeyExistence::NX));
    }
}
```

Let's have a look at this initial implementation, as the rest of the options will follow the same logic.

Two arguments are followed by a number so we need to be free to look at the next element, which means that we need to `loop` and to keep an eye on the current index.

src/set.rs

```
    let mut idx: usize = 0;

    loop {
        if idx >= arguments.len() {
            break;
        }

        ...

    }
```

In each loop we match the current element of the input vector against the name of a command and perform the relative actions. If the command is not among the supported ones we stop with an error.

src/set.rs

```
        match arguments[idx].to_lowercase().as_str() {
            "nx" => {
                ...
            }
            _ => {
                return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
            }
        }
```

The only argument we support at the moment is `NX`. As `NX` and `XX` are mutually exclusive, we need to check if `args.existence` is already set to `SetArgs::XX` and in that case we need to stop with an error. Otherwise, we can set `args.existence` to `SetArgs::NX` and increment the index.

src/set.rs

```
            "nx" => {
                if args.existence == Some(KeyExistence::XX) {
                    return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
                }

                args.existence = Some(KeyExistence::NX);

                idx += 1;
            }
```

The argument `xx`

The argument `XX` is clearly a clone of the above with minor changes to mirror the behaviour

src/set.rs

```
pub fn parse_set_arguments(arguments: &Vec<String>) -> StorageResult<Vec<SetArgs>> {
    let mut args: Vec<SetArgs> = vec![];

    let mut idx: usize = 0;

    loop {
        if idx >= arguments.len() {
            break;
        }

        match arguments[idx].to_lowercase().as_str() {

        ...

            "xx" => {
                if args.existence == Some(KeyExistence::NX) {
                    return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
                }

                args.existence = Some(KeyExistence::XX);
                idx += 1;
            }

        ...

        }
    }

    Ok(args)
}
```

The tests for this argument are

src/set.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_parse_xx() {
        let commands: Vec<String> = vec![String::from("XX")];

        let args = parse_set_arguments(&commands).unwrap();

        assert_eq!(args.existence, Some(KeyExistence::XX));
    }

    #[test]
    fn test_parse_xx_and_nx() {
        let commands: Vec<String> = vec![String::from("XX"), String::from("NX")];

        assert!(matches!(
            parse_set_arguments(&commands),
            Err(StorageError::CommandSyntaxError(_))
        ));
    }

    #[test]
    fn test_parse_nx_and_xx() {
        let commands: Vec<String> = vec![String::from("NX"), String::from("XX")];

        assert!(matches!(
            parse_set_arguments(&commands),
            Err(StorageError::CommandSyntaxError(_))
        ));
    }

    ...
}
```

The argument GET

The command GET has no specific requirements, as it is not clashing with anything

src/set.rs

```
        match arguments[idx].to_lowercase().as_str() {

        ...

            "get" => {
                args.get = true;
                idx += 1;
            }

        ...

        }
```

The tests for this argument are

src/set.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_parse_get() {
        let commands: Vec<String> = vec![String::from("GET")];

        let args = parse_set_arguments(&commands).unwrap();

        assert!(args.get);
    }

    #[test]
    fn test_parse_nx_and_get() {
        let commands: Vec<String> = vec![String::from("NX"), String::from("GET")];

        let args = parse_set_arguments(&commands).unwrap();
        assert_eq!(
            args,
            SetArgs {
                existence: Some(KeyExistence::NX),
                expiry: None,
                get: true
            }
        );
    }

    #[test]
    fn test_parse_xx_and_get() {
        let commands: Vec<String> = vec![String::from("XX"), String::from("GET")];

        let args = parse_set_arguments(&commands).unwrap();

        assert_eq!(
            args,
            SetArgs {
                existence: Some(KeyExistence::XX),
                expiry: None,
                get: true
            }
        );
    }

    ...

}
```

The arguments `EX` and `PX`

The two final commands `EX` and `PX` have the same structure as `NX` and `XX`, being mutually exclusive, but with the additional complexity of requiring a numeric value. We need to check that there is a following value and that it is a number. The implementation of `EX` is

src/set.rs

```
        match arguments[idx].to_lowercase().as_str() {

        ...

            "ex" => {
                if let Some(KeyExpiry::PX(_)) = args.expiry {
                    return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
                }

                if idx + 1 == arguments.len() {
                    return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
                }
                let value: u64 = arguments[idx + 1]
                    .parse()
                    .map_err(|_| StorageError::CommandSyntaxError(arguments.join(" ")))?;

                args.expiry = Some(KeyExpiry::EX(value));

                idx += 2;
            }

        ...

        }
```

and that of `PX` mirrors it

src/set.rs

```
        match arguments[idx].to_lowercase().as_str() {

        ...

            "px" => {
                if let Some(KeyExpiry::EX(_)) = args.expiry {
                    return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
                }

                if idx + 1 == arguments.len() {
                    return Err(StorageError::CommandSyntaxError(arguments.join(" ")));
                }

                let value: u64 = arguments[idx + 1]
                    .parse()
                    .map_err(|_| StorageError::CommandSyntaxError(arguments.join(" ")))?;

                args.expiry = Some(KeyExpiry::PX(value));

                idx += 2;
            }

        ...

        }
```

The tests for these arguments are

src/set.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_parse_ex() {
        let commands: Vec<String> = vec![String::from("EX"), String::from("100")];

        let args = parse_set_arguments(&commands).unwrap();

        assert_eq!(args.expiry, Some(KeyExpiry::EX(100)));
    }

    #[test]
    fn test_parse_ex_wrong_value() {
        let commands: Vec<String> = vec![String::from("EX"), String::from("value")];

        assert!(matches!(
            parse_set_arguments(&commands),
            Err(StorageError::CommandSyntaxError(_))
        ));
    }

    #[test]
    fn test_parse_ex_end_of_vector() {
        let commands: Vec<String> = vec![String::from("EX")];

        assert!(matches!(
            parse_set_arguments(&commands),
            Err(StorageError::CommandSyntaxError(_))
        ));
    }

    #[test]
    fn test_parse_px() {
        let commands: Vec<String> = vec![String::from("PX"), String::from("100")];

        let args = parse_set_arguments(&commands).unwrap();

        assert_eq!(args.expiry, Some(KeyExpiry::PX(100)));
    }

    #[test]
    fn test_parse_px_wrong_value() {
        let commands: Vec<String> = vec![String::from("PX"), String::from("value")];

        assert!(matches!(
            parse_set_arguments(&commands),
            Err(StorageError::CommandSyntaxError(_))
        ));
    }

    #[test]
    fn test_parse_px_end_of_vector() {
        let commands: Vec<String> = vec![String::from("PX")];

        assert!(matches!(
            parse_set_arguments(&commands),
            Err(StorageError::CommandSyntaxError(_))
        ));
    }

    #[test]
    fn test_parse_ex_and_px() {
        let commands: Vec<String> = vec![
            String::from("EX"),
            String::from("100"),
            String::from("PX"),
            String::from("100"),
        ];

        assert!(matches!(
            parse_set_arguments(&commands),
            Err(StorageError::CommandSyntaxError(_))
        ));
    }

    ...

}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step4.4

Step 5 - SET with expiry

It's time to update `Storage` to include the work we have done on `SET`. We can start importing the relevant components

src/storage.rs

```
use crate::set::{parse_set_arguments, SetArgs, KeyExpiry};
```

The internal command (implemented by `Storage::set`) should receive a argument of type `SetArgs` and act accordingly

src/storage.rs

```
impl Storage {

    ...

    fn set(&mut self, key: String, value: String, args: SetArgs) -> StorageResult<String> {
        let mut data = StorageData::from(value);

        if let Some(value) = args.expiry {
            let expiry = match value {
                KeyExpiry::EX(v) => Duration::from_secs(v),
                KeyExpiry::PX(v) => Duration::from_millis(v),
            };

            data.add_expiry(expiry);
            self.expiry
                .insert(key.clone(), SystemTime::now().add(expiry));
        }

        self.store.insert(key.clone(), data);

        Ok(String::from("OK"))
    }

    ...

}
```

Here, `EX` or `PX` are transformed into a `std::time::Duration` and added to the data that is about to be stored. The expiring key is also added to `self.expiry` for faster retrieval by `self.expire_keys`. To be able to run `SystemTime::now().add(expiry)` we also need to import a trait

src/storage.rs

```
use crate::resp::RESP;
use crate::set::{parse_set_arguments, KeyExpiry, SetArgs};
use crate::storage_result::{StorageError, StorageResult};
use std::collections::HashMap;
use std::ops::Add;
use std::time::{Duration, SystemTime};

...
```

As we want to implement also passive expiration, we need to change the method `get` as well.

src/storage.rs

```
impl Storage {

    ...

    fn get(&mut self, key: String) -> StorageResult<Option<String>> {
        if let Some(&expiry) = self.expiry.get(&key) {
            if SystemTime::now() >= expiry {
                self.expiry.remove(&key);
                self.store.remove(&key);
                return Ok(None);
            }
        }

        match self.store.get(&key) {
            Some(StorageData {
                value: StorageValue::String(v),
                creation_time: _,
                expiry: _,
            }) => return Ok(Some(v.clone())),
            None => return Ok(None),
        }
    }

    ...

}
```

The parameter `self` becomes mutable, as we are removing the key in case it's expired. Then we perform a simple check of the expiry time against `SystemTime::now()` to decide if the key is still valid. The last change to the methods is in `Storage::command_set`, where we need to parse the arguments and give them to `Storage::set`.

src/storage.rs

```rust
impl Storage {

    ...

    fn command_set(&mut self, command: &Vec<String>) -> StorageResult<RESP> {
        if command.len() < 3 {
            return Err(StorageError::CommandSyntaxError(command.join(" ")));
        }

        let key = command[1].clone();
        let value = command[2].clone();
        let args = parse_set_arguments(&command[3..].to_vec())?;

        let _ = self.set(key, value, args);

        Ok(RESP::SimpleString(String::from("OK")))
    }

    ...

}
```

Existing tests require some changes to match the new function prototypes

src/storage.rs

```rust
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_set_value() {
        let mut storage: Storage = Storage::new();
        let avalue = StorageData::from(String::from("avalue"));

        let output = storage
            .set(String::from("akey"), String::from("avalue"), SetArgs::new())
            .unwrap();

        assert_eq!(output, String::from("OK"));
        assert_eq!(storage.store.len(), 1);
        match storage.store.get(&String::from("akey")) {
            Some(value) => assert_eq!(value, &avalue),
            None => panic!(),
        }
    }
    ...

    #[test]
    fn test_get_value_key_does_not_exist() {
        let mut storage: Storage = Storage::new();

        let result = storage.get(String::from("akey")).unwrap();

        assert_eq!(storage.store.len(), 0);
        assert_eq!(result, None);
    }

    ...

        #[test]
    fn test_expire_keys() {
        let mut storage: Storage = Storage::new();

        storage
            .set(String::from("akey"), String::from("avalue"), SetArgs::new())
            .unwrap();

        storage.expiry.insert(
            String::from("akey"),
            SystemTime::now() - Duration::from_secs(5),
        );

        storage.expire_keys();
        assert_eq!(storage.store.len(), 0);
    }

    #[test]
    fn test_expire_keys_deactivated() {
        let mut storage: Storage = Storage::new();
        storage.set_active_expiry(false);

        storage
            .set(String::from("akey"), String::from("avalue"), SetArgs::new())
            .unwrap();

        storage.expiry.insert(
            String::from("akey"),
            SystemTime::now() - Duration::from_secs(5),
        );

        storage.expire_keys();
        assert_eq!(storage.store.len(), 1);
    }

    ...

}
```

We can also add a new test to check that `PX` works

src/storage.rs

```rust
#[cfg(test)]
mod tests {
    use super::*;

    ...

    #[test]
    fn test_set_value_with_px() {
        let mut storage: Storage = Storage::new();
        let mut avalue = StorageData::from(String::from("avalue"));
        avalue.add_expiry(Duration::from_millis(100));

        let output = storage
            .set(
                String::from("akey"),
                String::from("avalue"),
                SetArgs {
                    expiry: Some(KeyExpiry::PX(100)),
                    existence: None,
                    get: false,
                },
            )
            .unwrap();

        assert_eq!(output, String::from("OK"));
        assert_eq!(storage.store.len(), 1);
        match storage.store.get(&String::from("akey")) {
            Some(value) => assert_eq!(value, &avalue),
            None => panic!(),
        }

        storage.expiry.get(&String::from("akey")).unwrap();
    }

    ...
}
```

CodeCrafters

Stage 7: Expiry

This version of the code passes Stage 7 of the CodeCrafters challenge.

Source code

https://github.com/lgiordani/sider/tree/ed1/step4.5

Chapter 5

Concurrency with actors

You work like a slave for that idiot actor who won't give you a penny.

Amadeus (1984)

The code we developed so far is a basic (but working!) implementation of a remote key/value store like Redis. We developed an internal core that manages data and a request processor that can route commands, and with such devices we can implement many other services.

However, the component that manages the key expiry mechanism showed us that not everything in the system works according to the traditional request/response logic. For example, we can have components that react to timers and, more generally, parts of the system that respond to different events than the mere incoming client request.

The current architecture of the system is not ideal for such a task. For starters, the core of the system is the loop that runs `select!` in the function `main`, and adding more components would quickly lead to a version of the function that contains too many arms. While this is not a problem in terms of performance, it makes the system increasingly difficult to maintain and understand.

A second and more significant problem is connected with the Rust ownership model. Remember that in Rust, every time we call a function passing a variable, the function takes ownership of it.

Because of that, in a concurrent scenario we need to wrap shared resources with an `Arc<Mutex<T>>` and clone the reference before passing the value to a function, which will try to lock the underlying resource before accessing it.

Which resources are we discussing? In general, we need to pass relevant parts of the *state of the system* to every function we run in an asynchronous task. So far, the only shared resource we are managing is the storage, but in the future the complexity will increase. For example, to implement replication we will have to pass the list of connected replicas, and to implement transactions we will need to know if a client is currently in a transaction.

Overall, the `Arc<Mutex<T>>` pattern works very well for small systems, where the state passed to each function is compact. For more complex cases, we need a different approach. This is, specifically, the *actor model*.

Actors

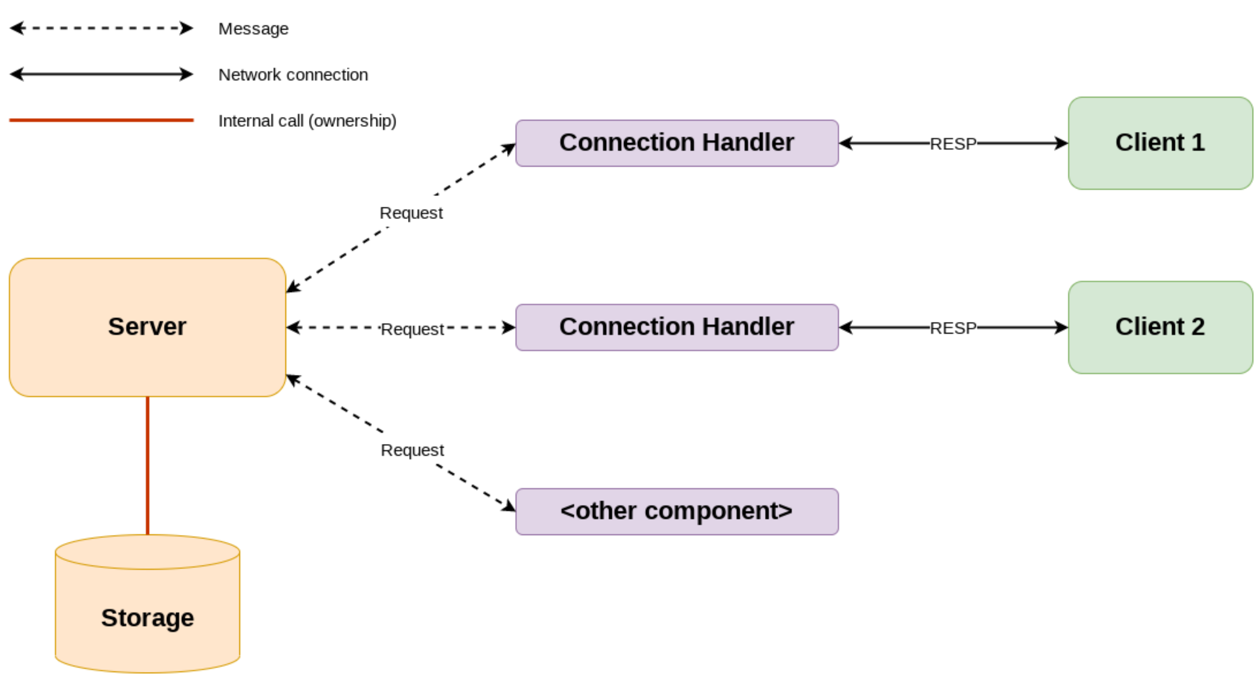
The idea of actors is extremely simple: they are independent components of an application that communicate with each other through messages. Introducing a message-based communication layer is one of the best strategies to promote decoupling between parts of a system, which is exactly what we need here.

Ultimately, actors and messaging systems inside a single application are yet another implementation of the concept of service and API, which you can find at different scales in any computer system (e.g. cloud computing, microservices, REST APIs).

When the system is run by actors, there is no need to use the `Arc<Mutex<T>>` model. A specific actor will be the unique owner of a certain resource, and other tasks can access the latter through messages. As you can see, this seems to perfectly fit the Rust ownership model.

In this chapter, we will refactor the existing system into a concurrent server based on actors, and we will learn to use some components of the Tokio runtime that simplify our job. In the next chapter we will use the new structure to implement replication.

The overall architecture we are going to implement is shown in the following picture.



The storage is owned by the server, and the interaction between the two consists of internal method calls. There is no need to lock the storage as the server is a single entity and won't therefore access the storage concurrently.

The server receives messages from other components, wrapped in a convenient structure `Request`. These messages will travel between Rust actors through Tokio channels, and are represented by specific Rust types.

At the moment, the only components that will send such messages are the Connection Handlers (one per client), but the picture shows that in the future there might be other components that implement different functionalities.

Connection Handlers receive RESP commands in binary form through a network connection, and their task is to convert them into `Request` messages. It's a good idea to decouple the interface we show to the client (RESP commands) from the interface the server uses internally.

The flexibility of the systems will be evident once we tackle replication and transactions, adding new components that will interact with the server to extend its functionalities.

Communication channels

To implement the new architecture we need two main components: a way to run actors and a way to exchange messages between them.

In this scenario, actors are just asynchronous tasks, so the way to run them is the asynchronous runtime, that in our case is Tokio. In particular, `tokio::spawn` [docs] will be the way to create actors. Rust is not an object-oriented programming language, so actors will always be functions.

As for messages, Tokio allows us to create *channels* that can be used to send and receive data to and from an actor (or any other part of a system). There are two main types of channels that can be created in Tokio. *One-shot channels* with `tokio::sync::oneshot` [docs] and *multi-producer/single-consumer* channels (MPSC) with `tokio::sync::mpsc`[docs]. We can consider channels like queues shared among asynchronous tasks (and thus among threads).

Tokio channels are always one way. A sender can push messages to the receiver, but the channel doesn't offer the latter a way to reply.

One-shot vs MPSC

One-shot channels are a way to send a single message between a producer and a consumer. In a scenario where an actor doesn't keep track of the components that connect to it, it's useful to use one-shot messages. Each message sent to the server will contain the one-shot channel that can be used to deliver responses. This mechanism is similar to that of pre-printed envelopes used in postal ballots, where you receive documents and the envelope that you have to use to send them back.

MPSC channels, on the other hand, are useful to establish a more permanent communication route. They allow us to create a single sender and multiple receivers, but they are perfect also for a scenario with a single receiver.

For reasons that will be clear in the next sections, in this book we will mostly use MPSC channels. We will however retain the idea of sending messages that contain the response channel.

MPSC channels

An MPSC channel is made of a `Sender` [docs] and a `Receiver` [docs].

Since MPSC channels have multiple producers the sender can be cloned [docs], while the receiver cannot.

In the rest of the book, we will stick to the following naming schema for channels:

- `COMPONENT_sender` is the cloneable `mpsc::Sender` used to send a message to `COMPONENT`
- `COMPONENT_receiver` is the `mpsc::Receiver` used by `COMPONENT` to receive messages.

Conversion steps

To convert the server into an actor we will go through the following steps:

1. Isolate types to represent server results and errors.
2. Create types to represents requests and messages sent to the server.
3. Refactor the connection handler into a `select!` loop.
4. Use the request type (without messages) in the connection handler.
5. Turn the server into an actor that listens for messages. At this stage the code will compile but not work properly.
6. Make the connection handler use messages.
7. Tidy up the code.

The steps are (hopefully) small enough to be understandable. The code will be in a working state that can be compiled and used until we turn the server into an actor (step 5). That and the following step have been split for clarity's sake, at the cost of having a stage where the system doesn't work properly.

Step 1 - Supporting types

Let's start, as usual, taking care of result types and errors. In a previous chapter, we put `ServerError` aside when we isolated the storage. Now, we can resurrect the code, moving both `ServerError` and `ServerResult` into `src/server_result.rs`

`src/server_result.rs`

```rust
use std::fmt;

#[derive(Debug, PartialEq)]
pub enum ServerError {
    CommandError,
}

impl fmt::Display for ServerError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            ServerError::CommandError => write!(f, "Error while processing!"),
        }
    }
}

pub type ServerResult<T> = Result<T, ServerError>;
```

We also need to add the file as a module

`src/main.rs`

```rust
mod resp;
mod resp_result;
mod server;
mod server_result;
mod set;
mod storage;
mod storage_result;
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.1

Step 2 - Requests and messages

The connection handler receives the client's requests in RESP binary format and needs to send them to the server. It makes sense to wrap this piece of information in a struct `Request` to capture all the data that we want to exchange. At the moment we want to send the RESP request and a sender for the response, but in the future we will need to add more fields like for example the binary representation of the RESP request.

First of all, the connection handler will send messages to the server, and we need a type to represent them.

`src/connection.rs`

```rust
use crate::request::Request;

#[derive(Debug)]
pub enum ConnectionMessage {
    Request(Request),
}
```

The reason for the abstraction is that in the future we might want to send other types of data, such as internal commands.

As explained before, the `Request` should contain the client's RESP request, but also the channel used to send a response. There is no such entity as "the channel", though, and the only way to access it is through a `Sender`. Remember the metaphor of the postal ballot envelope.

`src/request.rs`

```rust
use crate::{resp::RESP, server_result::ServerMessage};
use tokio::sync::mpsc;

#[derive(Debug)]
pub struct Request {
    pub value: RESP,
    pub sender: mpsc::Sender<ServerMessage>,
}
```

The sender will carry a response, that is a message that comes from the server, so let's define `ServerMessage`. Currently, any request sent to the server produces either a response in RESP binary format or an error.

src/server_result.rs

```rust
use crate::resp::RESP;

...

#[derive(Debug)]
pub enum ServerMessage {
    Data(RESP),
    Error(ServerError),
}
```

Finally, we need to define a structure that carries a message from the connection handler. At the moment, the only possible message to the server is a request.

It's in general a good idea to abstract messages with enums. They are a powerful way to express data types that can evolve in the future without affecting the function prototypes that have been already developed.

Finally, let's add the new files to the list of modules

src/main.rs

```rust
mod connection;
mod request;
mod resp;
mod resp_result;
mod server;
mod server_result;
mod set;
mod storage;
mod storage_result;
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.2

Step 3 - Refactor the connection handler

In this section we will refactor the connection handler to use the macro `select!` instead of awaiting directly `stream.read`. Being a refactor, we are not going to change the behaviour, but we are preparing the handler to host multiple asynchronous actions. This will be useful later when we receive messages sent by the server.

The refactoring is straightforward. Instead of

```
match stream.read(&mut buffer).await {
```

we will have

```
select! {
    result = stream.read(&mut buffer) => {
        match result {
```

but the rest of the function will be left untouched

src/main.rs

```
async fn handle_connection(mut stream: TcpStream, storage: Arc<Mutex<Storage>>) {
    let mut buffer = [0; 512];

    loop {
        select! {
            result = stream.read(&mut buffer) => {
                match result {
                    Ok(size) if size != 0 => {
                        let mut index: usize = 0;

                        let request = match bytes_to_resp(&buffer[..size].to_vec(), &mut
                        ↪  index) {
                            Ok(v) => v,
                            Err(e) => {
                                eprintln!("Error: {}", e);
                                return;
                            }
                        };

                        let response = match process_request(request, storage.clone()) {
                            Ok(v) => v,
                            Err(e) => {
                                eprintln!("Error parsing command: {}", e);
                                return;
                            }
                        };
                        if let Err(e) = stream.write_all(response.to_string().as_bytes()).
                        ↪   await {
                            eprintln!("Error writing to socket: {}", e);
                        }
                    }
                    Ok(_) => {
                        println!("Connection closed");
                        break;
                    }
                    Err(e) => {
                        println!("Error: {}", e);
                        break;
                    }
                }
            }
        }
    }
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.3

Step 4 - Send Requests to the server

This step involves more changes than the previous ones, but there is no additional complexity. At the moment the connection handler calls `process_request` passing the RESP request directly, but we will change it in order to use a `Request`.

This change prepares the system for a later step, where we won't call `process_request` directly any more, but will instead send a message to the server.

Let's start with the connection handler. Since we need to create a `Request`, we also need to create a channel that will send and receive messages of type `ServerMessage`

src/main.rs

```rust
use crate::request::Request;
use crate::resp::{bytes_to_resp, RESP};
use crate::server::process_request;
use crate::storage::Storage;
use server_result::ServerMessage;
use std::sync::{Arc, Mutex};
use std::time::Duration;
use tokio::{
    io::{AsyncReadExt, AsyncWriteExt},
    net::{TcpListener, TcpStream},
    select,
    sync::mpsc,
};

...

async fn handle_connection(mut stream: TcpStream, storage: Arc<Mutex<Storage>>) {
    let mut buffer = [0; 512];

    let (connection_sender, _) = mpsc::channel::<ServerMessage>(32);

    loop {
        select! {
            result = stream.read(&mut buffer) => {
                match result {
                    Ok(size) if size != 0 => {
                        let mut index: usize = 0;

                        let resp = match bytes_to_resp(&buffer[..size].to_vec(), &mut index
                        ↪ ) {
                            Ok(v) => v,
                            Err(e) => {
                                eprintln!("Error: {}", e);
                                return;
                            }
                        };

                        let request = Request {
                            value: resp,
                            sender: connection_sender.clone(),
                        };

                        let response = match process_request(request, storage.clone()) {
                            Ok(v) => v,
                            Err(e) => {
                                eprintln!("Error parsing command: {}", e);
                                return;
                            }
                        };

                        if let Err(e) = stream.write_all(response.to_string().as_bytes()).
                        ↪  await {
                            eprintln!("Error writing to socket: {}", e);
                        }
                    }
                    Ok(_) => {
                        println!("Connection closed");
                        break;
                    }
                    Err(e) => {
                        println!("Error: {}", e);
                        break;
                    }
                }
            }
        }
    }
}
```

Now we need to change `process_request` accordingly and adjust the tests. The function can be fixed very quickly

src/server.rs

```
use crate::request::Request;
use crate::storage::Storage;
use crate::storage_result::{StorageError, StorageResult};
use crate::RESP;
use std::sync::{Arc, Mutex};

pub fn process_request(request: Request, storage: Arc<Mutex<Storage>>) -> StorageResult<
↪  RESP> {
    let elements = match request.value {
        RESP::Array(v) => v,
        _ => {
            return Err(StorageError::IncorrectRequest);
        }
    };

...
```

Testing this function now requires a `Request`, which means that we also need to create the MPSC channels in the tests

src/server.rs

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_process_request_ping() {
        let (connection_sender, _) = mpsc::channel::<ServerMessage>(32);

        let request = Request {
            value: RESP::Array(vec![RESP::BulkString(String::from("PING"))]),
            sender: connection_sender,
        };

        let storage = Arc::new(Mutex::new(Storage::new()));

        let output = process_request(request, storage).unwrap();

        assert_eq!(output, RESP::SimpleString(String::from("PONG")));
    }

    #[test]
    fn test_process_request_echo() {
        let (connection_sender, _) = mpsc::channel::<ServerMessage>(32);

        let request = Request {
            value: RESP::Array(vec![
                RESP::BulkString(String::from("ECHO")),
                RESP::BulkString(String::from("42")),
            ]),
            sender: connection_sender,
        };

        let storage = Arc::new(Mutex::new(Storage::new()));

        let output = process_request(request, storage).unwrap();

        assert_eq!(output, RESP::BulkString(String::from("42")));
    }

    #[test]
    fn test_process_request_not_array() {
        let (connection_sender, _) = mpsc::channel::<ServerMessage>(32);

        let request = Request {
            value: RESP::BulkString(String::from("PING")),
            sender: connection_sender,
        };

        let storage = Arc::new(Mutex::new(Storage::new()));

        let error = process_request(request, storage).unwrap_err();
        assert_eq!(error, StorageError::IncorrectRequest);
    }

    #[test]
    fn test_process_request_not_bulkstrings() {
        let (connection_sender, _) = mpsc::channel::<ServerMessage>(32);

        let request = Request {
            value: RESP::Array(vec![RESP::SimpleString(String::from("PING"))]),
            sender: connection_sender,
        };

        let storage = Arc::new(Mutex::new(Storage::new()));

        let error = process_request(request, storage).unwrap_err();

        assert_eq!(error, StorageError::IncorrectRequest);
    }
}
```

We also need to import the two types where they were previously defined

src/server.rs

```
use crate::server_result::{ServerError, ServerResult};
use crate::storage::Storage;
use crate::storage_result::{StorageError, StorageResult};
use crate::RESP;
use std::sync::{Arc, Mutex};
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.4

Step 5 - Turn the server into an actor

At this point we can convert the server into a full-fledged actor. This clearly involves the connection handler as well, but in this step we will focus solely on the server. In the next step we will fix the connection handler and make sure both ends of the communication work properly. For this reason, at the end of this section the code won't work.

The server

Let's create a structure to capture the information the server needs to manage

src/server.rs

```rust
pub struct Server {
    pub storage: Option<Storage>,
}
```

As you can see, this is the core of the idea behind actors: the server *owns* the storage.

src/server.rs

```rust
impl Server {
    pub fn new() -> Self {
        Self { storage: None }
    }

    pub fn set_storage(mut self, storage: Storage) -> Self {
        self.storage = Some(storage);
        self
    }
}
```

In this implementation the server can be initialised without the storage, but this is just a stylistic choice that doesn't affect the way the actor works.

The next change introduces the actor itself, which is a function called `run_server`

src/server.rs

```rust
use crate::connection::ConnectionMessage;
use tokio::sync::mpsc;

...

pub async fn run_server(mut server: Server, mut crx: mpsc::Receiver<ConnectionMessage>) {
    loop {
        tokio::select! {
            Some(message) = crx.recv() => {
                match message {
                    ConnectionMessage::Request(request) => {
                        process_request(request, &mut server).await;
                    }
                }
            }
        }
    }
}
```

As you can see this is initialised with a `Server` and a channel that receives messages from the connection handler. The body is an infinite loop that fetches a message, extracts the `Request`, and runs `process_request` on it.

Speaking of which, the function `process_request` needs to be changed as well, since we are now passing directly a reference to `Server` instead of an `Arc<Mutex<Storage>>`.

src/server.rs

```rust
pub async fn process_request(request: Request, server: &mut Server) {
    let elements = match &request.value {
        RESP::Array(v) => v,
        _ => {
            panic!()
        }
    };

    let mut command = Vec::new();
    for elem in elements.iter() {
        match elem {
            RESP::BulkString(v) => command.push(v.clone()),
            _ => {
                panic!()
            }
        }
    }

    let storage = match server.storage.as_mut() {
        Some(storage) => storage,
        None => panic!(),
    };

    let response = storage.process_command(&command);
}
```

There are other important changes in the function. The first one is that the code to return an error containing `StorageError::IncorrectRequest` has been removed and replaced with `panic!()`. The reason is that we haven't set up the other half of the system that receives messages yet, so we don't have any other way to signal an error. `panic!()` is a good placeholder for now.

The second change is in the logic at the bottom of the function. We don't need to lock the storage any more as the server is the sole owner of that resource. The response is not returned by the function as it was previously. Once again, we need a system to send it back to the connection handler, and that will be implemented in the next step.

As we created a new type, it's a good idea to have tests, so we'll create `test_create_new` and `test_set_storage`. As `process_request` at the moment doesn't output the response in any meaningful way, we should also comment out the remaining tests

src/server.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_create_new() {
        let server: Server = Server::new();

        match server.storage {
            Some(_) => panic!(),
            None => (),
        };
    }

    #[test]
    fn test_set_storage() {
        let storage = Storage::new();

        let server: Server = Server::new().set_storage(storage);

        match server.storage {
            Some(_) => (),
            None => panic!(),
        };
    }
    // #[test]
    // fn test_process_request_ping() {
        ...
    // }

    // #[test]
    // fn test_process_request_echo() {
        ...
    // }

    // #[test]
    // fn test_process_request_not_array() {
        ...
    // }

    // #[test]
    // fn test_process_request_not_bulkstrings() {
        ...
    // }
}
```

The connection handler

Now that the server has been converted, we need to change the connection handler so that it uses a message channel to send requests.

The function `handle_connection` doesn't need to receive an `Arc<Mutex<Storage>>` any more, but will receive a sender for `ConnectionMessage` entities.

src/main.rs

```
async fn handle_connection(mut stream: TcpStream, server_sender: mpsc::Sender<
↪  ConnectionMessage>) {

    ...
```

The call to `process_request` is replaced by a call to `server_sender.send`, and since the latter doesn't directly return a response, the code that managed the error has been removed. Once again, this will be completed in the next step.

src/main.rs

```
async fn handle_connection(mut stream: TcpStream, server_sender: mpsc::Sender<
↪  ConnectionMessage>) {

    ...

                        let request = Request {
                            value: resp,
                            sender: connection_sender.clone(),
                        };

                        match server_sender.send(ConnectionMessage::Request(request)).await
                        ↪   {
                            Ok(()) => {},
                            Err(e) => {
                                eprintln!("Error sending request: {}", e);
                                return;
                            }
                        }
                    }
                    Ok(_) => {
                        println!("Connection closed");
                        break;
                    }
                    Err(e) => {
                        println!("Error: {}", e);
                        break;
                    }
                }
            }
        }
    }
}
```

As handle_connection changed its prototype, we need to change the code of main accordingly. As you can see, we also create the channel that will be used to send messages to the server.

src/main.rs

```
use connection::ConnectionMessage;

...

#[tokio::main]
async fn main() -> std::io::Result<()> {

    ...

    let (server_sender, _) = mpsc::channel::<ConnectionMessage>(32);
    loop {
        tokio::select! {
            connection = listener.accept() => {
                match connection {
                    Ok((stream, _)) => {
                        tokio::spawn(handle_connection(stream, server_sender.clone()));
                    }

                    Err(e) => {
                        println!("Error: {}", e);
                        continue;
                    }
                }
            }

            ...
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.5

Step 6 - Use messages in the connection handler

In the current state, the code cannot work properly because the function `process_request` doesn't have any way to send its response back to the connection handler. Previously, the function was called directly, so it was just a matter of awaiting its completion and reading the output value.

In this new configuration, however, the function is called through a message, so there is no immediate return value. The most natural way to send back the response is through a message from the server, which means that we need to modify the connection handler adding some code to receive messages and process their content. We also need to run the actor itself, which in this case is the function `run_server`.

Connection results

Let's start as usual defining a type to represent errors in the connection.

`src/connection.rs`

```rust
use crate::{request::Request, server_result::ServerError};
use std::fmt;

#[derive(Debug)]
pub enum ConnectionError {
    ServerError(ServerError),
}

impl fmt::Display for ConnectionError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            ConnectionError::ServerError(e) => {
                write!(f, "{}", format!("Server error: {}", e))
            }
        }
    }
}
```

At the moment there is only one possible error in the connection handler, which is an unsuccessful response from the server, represented here by the variant `ConnectionError::ServerError`.

Listening for server responses

The function `handle_connection` that represents the core of the connection handler has been already converted into a `select!` loop, so we just need to add a new arm that uses the `connection_receiver`

src/main.rs

```
async fn handle_connection(mut stream: TcpStream, server_sender: mpsc::Sender<
↪   ConnectionMessage>) {
    let mut buffer = [0; 512];

    let (connection_sender, mut connection_receiver) = mpsc::channel::<ServerMessage>(32);

    loop {
        select! {
            result = stream.read(&mut buffer) => {
                ...
            }

            Some(response) = connection_receiver.recv() => {
                let _ = match response {
                    ServerMessage::Data(v) => stream.write_all(v.to_string().as_bytes()).
                    ↪   await,
                    ServerMessage::Error(e) => {
                        eprintln!("Error: {}", ConnectionError::ServerError(e));
                        return;
                    }
                };
            }

        }
    }
}
```

Here, `connection_receiver.recv()` is awaited by `select!` and when a response is detected it is written to the stream if successful and to the standard error otherwise. Where does this response come from? As you remember, in `handle_connection` we clone `connection_sender` and we store it into the request, so that the server can use it

src/server.rs

```
use crate::server_result::ServerMessage;

...

pub async fn process_request(request: Request, server: &mut Server) {

    ...

    let response = storage.process_command(&command);

    match response {
        Ok(v) => {
            request.sender.send(ServerMessage::Data(v)).await.unwrap();
        }
        Err(e) => (),
    }
}
```

Please note that `process_request` is still heavily under construction, as we are not properly dealing with the error in the code above and still haven't replaced the `panic!()` calls.

Running the server

The last change of this step is to run the server as an actor, so that the connection handler has something to exchange messages with. To do this we have to change the function `main` to spawn the actor

`src/main.rs`

```
use crate::connection::ConnectionError;
use server::{run_server, Server};
use tokio::io::{AsyncReadExt, AsyncWriteExt}

...

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await?;

    let storage = Storage::new();
    let mut server = Server::new();
    server = server.set_storage(storage);

    let (server_sender, server_receiver) = mpsc::channel::<ConnectionMessage>(32);

    tokio::spawn(run_server(server, server_receiver));

    loop {

        ...
```

As you can see, the storage becomes part of the server and the server is run as an actor through `tokio::spawn`. The server receiver is now needed by `run_server`, so we store it in `server_receiver`. Please note that we also got rid of the second arm of the `select!` that awaited `interval_timer.tick()` and of the initialisation of `interval_timer`. As the storage is now owned by the server it is not possible to work on it directly, and in the next step we will move the management of key expiry somewhere else.

With the latest changes, the system comes back to life, and we can once again pass the end to end tests. Active expiry is not working any more, but passive expiry does, and that's enough to make the test pass.

The code is clearly in a terrible state. Imports should be tidied up, unit tests have been commented, and error management has been replaced by `panic!` calls. So, the plan for the next steps is to tidy it up.

CodeCrafters

Stage 7: Expiry

This version of the code passes Stage 7 of the CodeCrafters challenge, just like the version we had at the end of the previous chapter. This shows that our refactoring worked.

Source code

`https://github.com/lgiordani/sider/tree/ed1/step5.6`

Step 7 - Tidy up the code

When we go through a major refactoring, it's always important to keep the system in a working state as much as possible. The byproduct of this succession of intermediate states, where the system is transitioning from the old architecture to the new one, is a lot of temporary code and unused imports, and more generally an untidiness of the code base.

In this step we need to fill in the gaps and to make sure the code is well written.

Step 7.1 - Active expiry

The first move is to restore active expiry, that was removed in the previous step as `storage` was no more accessible from `main`. The strategy here is very simple.

First of all let's remove `expire_keys` from `main.rs` and recreate it as a method of the struct `Server`.

`src/server.rs`

```rust
impl Server {
    pub fn new() -> Self {
        Self { storage: None }
    }

    pub fn set_storage(mut self, storage: Storage) -> Self {
        self.storage = Some(storage);
        self
    }

    pub fn expire_keys(&mut self) {
        let storage = match self.storage.as_mut() {
            Some(storage) => storage,
            None => return,
        };

        storage.expire_keys();
    }
}
```

The fact that `expire_keys` is now a method instead of a simple function is just a matter of preference.

Now, since the server actor `run_server` is a `select!` loop we can once again create `interval_timer` and await `interval_timer.tick()`.

src/server.rs

```
use std::time::Duration;

...

pub async fn run_server(mut server: Server, mut crx: mpsc::Receiver<ConnectionMessage>) {
    let mut interval_timer = tokio::time::interval(Duration::from_millis(10));

    loop {
        tokio::select! {

            ...

            _ = interval_timer.tick() => {
                server.expire_keys();
            }
        }
    }
}
```

Once again, as the server owns the storage there is no need to lock the resource here.

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.7.1

Step 7.2 - Wrapping server values

So far we assumed that the server could return any type of result, but it makes sense to identify the actual types and to group them into an enum. The only one we need so far, at any rate, is RESP

src/server_result.rs

```
#[derive(Debug)]
pub enum ServerValue {
    RESP(RESP),
}

pub type ServerResult = Result<ServerValue, ServerError>;

#[derive(Debug)]
```

```
pub enum ServerMessage {
    Data(ServerValue),
    Error(ServerError),
}
```

This causes a couple of changes in other functions. In `handle_connection`

src/main.rs

```
use crate::server_result::ServerValue;

...

async fn handle_connection(mut stream: TcpStream, server_sender: mpsc::Sender<
↪  ConnectionMessage>) {
...

    loop {
        select! {
            ...

            Some(response) = connection_receiver.recv() => {
                let _ = match response {
                    ServerMessage::Data(ServerValue::RESP(v)) => stream.write_all(v.
                    ↪  to_string().as_bytes()).await,
                    ServerMessage::Error(e) => {
                        eprintln!("Error: {}", ConnectionError::ServerError(e));
                        return;
                    }
                };
            }

        }
    }
}
```

and in `process_request`

src/server.rs

```
use crate::server_result::{ServerMessage, ServerValue};

...

pub async fn process_request(request: Request, server: &mut Server) {
    ...

    match response {
        Ok(v) => {
            request
                .sender
                .send(ServerMessage::Data(ServerValue::RESP(v)))
                .await
                .unwrap();
        }
        Err(e) => (),
    }
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.7.2

Step 7.3 - A better Request

The struct `Request` is pretty important for the new architecture, as a lot of the functionalities of the system are connected with it. It makes sense to add a couple of helper methods to simplify its usage

src/request.rs

```
use crate::{
    resp::RESP,
    server_result::{ServerError, ServerMessage, ServerValue},
};

...

impl Request {
    pub async fn error(&self, e: ServerError) {
        self.sender.send(ServerMessage::Error(e)).await.unwrap();
    }

    pub async fn data(&self, d: ServerValue) {
        self.sender.send(ServerMessage::Data(d)).await.unwrap();
    }
}
```

and we can use them directly in `process_request`. First to simplify the last call to `request.sender.send`

src/server.rs

```rust
pub async fn process_request(request: Request, server: &mut Server) {
    ...

    match response {
        Ok(v) => {
            request.data(ServerValue::RESP(v)).await;
        }
        Err(e) => (),
    }
}
```

and then to replace the `panic!` calls. To do that we first need to define a couple of new variants of `ServerError`

src/server_result.rs

```rust
#[derive(Debug, PartialEq)]
pub enum ServerError {
    CommandError,
    IncorrectData,
    StorageNotInitialised,
}

impl fmt::Display for ServerError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            ServerError::CommandError => write!(f, "Error while processing!"),
            ServerError::IncorrectData => {
                write!(f, "Data received from stream is incorrect.")
            }
            ServerError::StorageNotInitialised => {
                write!(f, "Storage has not been initialised.")
            }
        }
    }
}
```

and then we can work on `process_request`

src/server.rs

```rust
pub async fn process_request(request: Request, server: &mut Server) {
    let elements = match &request.value {
        RESP::Array(v) => v,
        _ => {
            request.error(ServerError::IncorrectData).await;
            return;
        }
    };

    let mut command = Vec::new();
    for elem in elements.iter() {
        match elem {
            RESP::BulkString(v) => command.push(v.clone()),
            _ => {
                request.error(ServerError::IncorrectData).await;
                return;
            }
        }
    }

    let storage = match server.storage.as_mut() {
        Some(storage) => storage,
        None => {
            request.error(ServerError::StorageNotInitialised).await;
            return;
        }
    };

    let response = storage.process_command(&command);

    match response {
        Ok(v) => {
            request.data(ServerValue::RESP(v)).await;
        }
        Err(e) => (),
    }
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.7.3

Step 7.4 - Isolate the connection handler

Connection handlers are arguably the central part of the system, which after all is a server whose main task is to accept incoming requests and send appropriate responses. When we converted the server into an actor we did the same to connection handlers, which now react to "messages" from clients (requests) and to messages from the server. For each incoming connection, the main loop spawns a dedicated connection handler to monitor it.

The specific actor for the connection handler is the function `handle_connection`, so it is worth isolating it in a separate space. To make the code tidier, it's also useful to capture the main `select!` loop in a separate space.

Let's start moving `handle_connection` to `src/connection.rs`

`src/connection.rs`

```rust
use crate::request::Request;
use crate::resp::bytes_to_resp;
use crate::server_result::{ServerError, ServerMessage, ServerValue};
use tokio::{
    io::{AsyncReadExt, AsyncWriteExt},
    net::TcpStream,
    select,
    sync::mpsc,
};

...

async fn handle_connection(
    mut stream: TcpStream,
    server_sender: mpsc::Sender<ConnectionMessage>,
) {

    ...

}
```

The function is moved as it is. To isolate the listener loop, instead, we need to create a new function

`src/connection.rs`

```rust
use tokio::{
    io::{AsyncReadExt, AsyncWriteExt},
    net::{TcpListener, TcpStream},
    select,
    sync::mpsc,
};

...
```

```
pub async fn run_listener(host: String, port: u16, server_sender: mpsc::Sender<
↪  ConnectionMessage>) {
    let listener = TcpListener::bind(format!("{}:{}", host, port))
        .await
        .unwrap();

    loop {
        tokio::select! {
            connection = listener.accept() => {
                match connection {
                    Ok((stream, _)) => {
                        tokio::spawn(handle_connection(stream, server_sender.clone()));
                    }
                    Err(e) => {
                        eprintln!("Error: {}", e);
                        continue;
                    }
                }
            }
        }
    }
}
```

and this needs to be called in `main`

src/main.rs

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let storage = Storage::new();
    let mut server = Server::new();
    server = server.set_storage(storage);

    let (server_sender, server_receiver) = mpsc::channel::<ConnectionMessage>(32);

    tokio::spawn(run_server(server, server_receiver));

    run_listener("127.0.0.1".to_string(), 6379, server_sender).await;

    Ok(())
}
```

Source code

https://github.com/lgiordani/sider/tree/ed1/step5.7.4

Next steps

Now, the next step's a little tricky.
The Great Escape (1963)

The work done so far brought us from a simple server that can bind to a TCP port to a rich system that implements a basic remote dictionary with a clever architecture that can be easily extended.

The CodeCrafters challenge targets some of these extensions, allowing us to explore the following topics:

- Replication
- Transactions
- RDB persistence
- Streams

As I mentioned in the introduction, the book is not 100% complete, as I plan to tackle each one of those. The code for the first two is already working, but I still have to split it into steps and to write the relative explanation.

Final words (for now)

I hope you enjoyed the book so far!

As I said, I’m just a beginner with Rust, so I hope the book might help others to continue their journey with this amazing language.

Any type of feedback, correction, or suggestion is more than welcome.

Thanks for reading my book!