Readers & Writers

Importance and usage

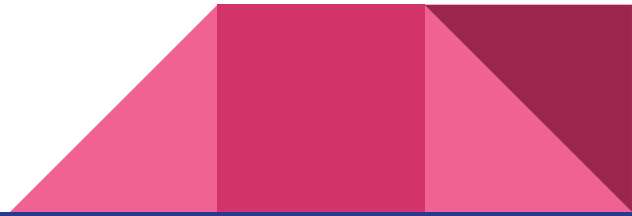
Reader and Writer Interface

func Read(p []byte) (n int, err error)

func Write(p []byte) (n int, err error)

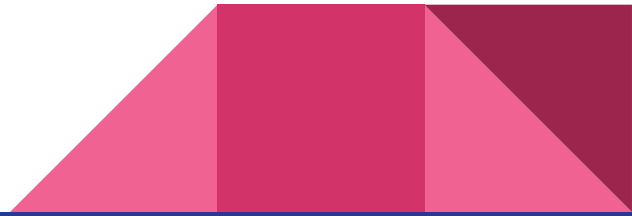
Importance

- Readers and Writers are the only way to get data in and out of a command line program.
- Even in graphical programs, Readers and Writers are used for filesystem and networking.



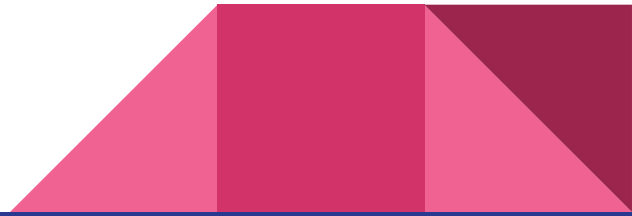
Direct Usage

- There is more to Readers and Writers than just calling their function.
- It is rare that you will call Read or Write directly.
- Additionally, even if you do call Read or Write directly, you will usually have to convert your data to or from a slice of bytes.



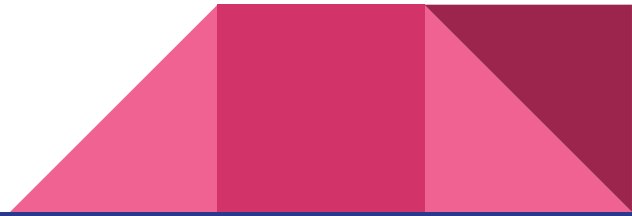
Three Parts

- Most interactions with Readers and Writers takes place in two or three parts: Converter, Filter, and Endpoint.
- The Converter changes your data to or from a slice of bytes.
- The Filter changes a slice of bytes into a different slice of bytes.
- The Endpoint is the actual Reader or Writer.



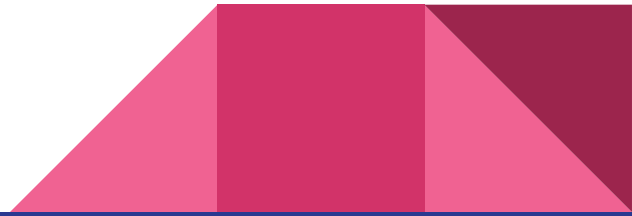
Endpoint

- The endpoint is the actual Reader or Writer itself.
- There are many possible Endpoints.
- The most common Endpoints are:
  - os.Stdin, os.Stdout
  - Files
  - Network connections
- In-memory buffers are also common in tests in go.



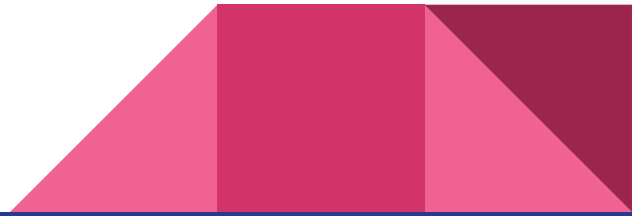
Endpoint

- Most endpoints follow the ReadCloser or WriteCloser interface.
- This just means that they have a Close() method in addition to the Read or Write method.
- Most of the time, you will use defer on the Close method after creating the endpoint, so the endpoint does not remain active, taking up resources.



Converter

- This part of the program changes data to or from a slice of bytes.
- It is rare that data in a program is already in a slice of bytes. Most data is in structs, or ints, or strings.
- The converter will change the data into a slice of bytes and then change the same slice of bytes back into the original data.
- This process is known as Serialization, or occasionally as Marshalling and Unmarshalling.

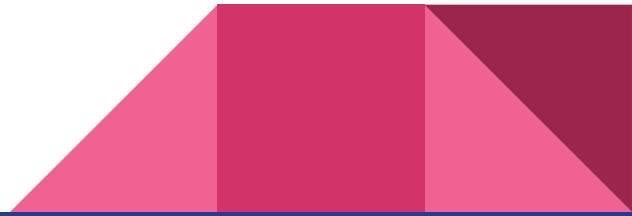


Converter

- The converter usually has its own set of functions to call, but will usually take in a Reader or a Writer when it is being created.
- The more restrictions on the data a converter imposes, the smaller the resulting slice of bytes will usually be.
- The JSON library and the CSV library are examples of converters.

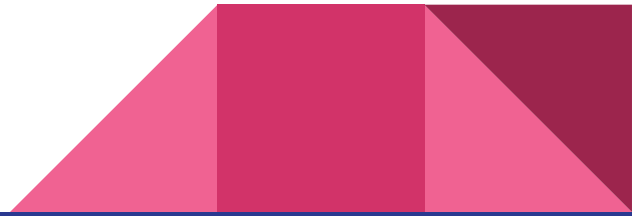
Converter

- Surprisingly, the fmt package is full of converters as well, the Print and Scan functions are designed to convert strings, with other data types, into slices of bytes.
- The Print and Scan functions have Fprint and Fscan variants that take in a Reader or a Writer as their first argument.
- The basic Print and Scan functions go to os.Stdin and os.Stdout, which are linked to the command line.



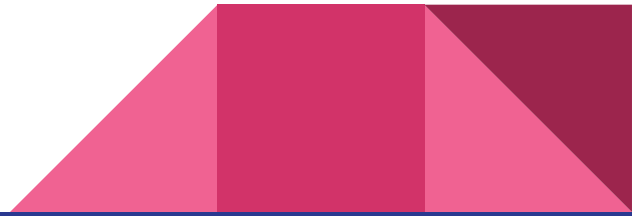
Converter

- Usually, you will have some additional code on top of a builtin library that performs extra work as well.
- This code may put variables into different positions, or perform additional calculations to fill in holes in the data.
- You can also write your own converters.
- A converter that only works on your own data, will be the smallest possible.



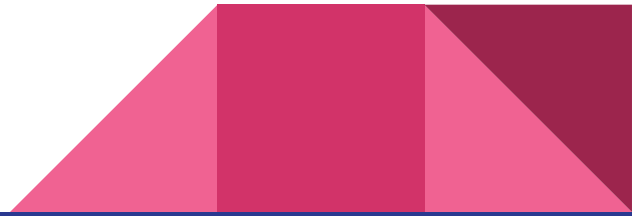
Filter

- The filter changes a slice of bytes into a different slice of bytes.
- Filters are not really commonly used.
- Most filters follow the Reader or Writer interface, and take in another Reader or Writer when they are being created.
- Compression and Encryption are the two most common filters used in programs.
- Filters are also used to make data access easier or safer in some way.



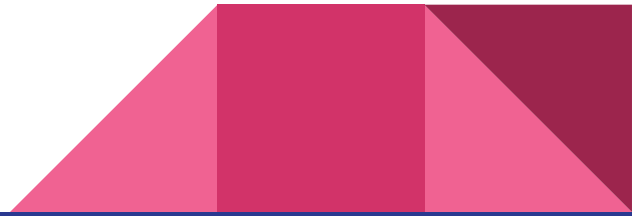
Filter

- For example: the ioutil.ReadAll function gets all of the bytes from a Reader all at once. This can be dangerous.
- If the Reader is a network connection, what if the other end is malicious and sends several gigabytes of data.
- The io.LimiterReader filter wraps around a Reader, and limits the maximum data from that Reader to a programmer set limit.
- This way, you can set a maximum, and use ReadAll without worrying about how big the actual data is.



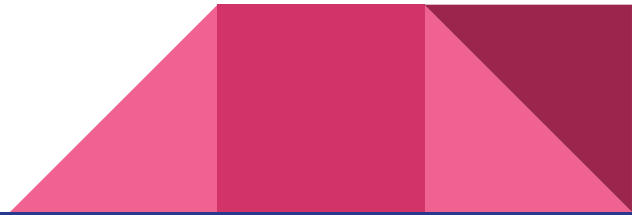
Web Server Pattern

- Get data from request reader
- Get data from database reader
- Process data
- Sometimes send data to the database writer
- If request is a normal request, send data through a template converter to the response writer
- If request is an AJAX request, send data through a JSON converter to the response writer



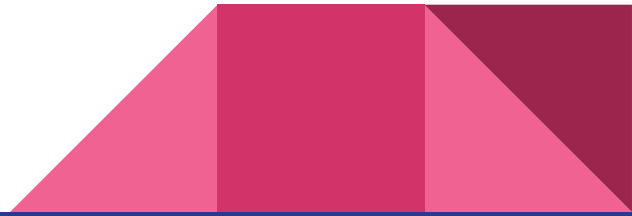
JSON

- The *encoding/json* package converts data to and from JSON data.
- It is probably the second most used converter in web programming.
- It is not very space efficient.
- It is very useful since Javascript has built-in functions to convert to and from JSON data as well.
- It has two modes: immediate mode, and wrapper mode.



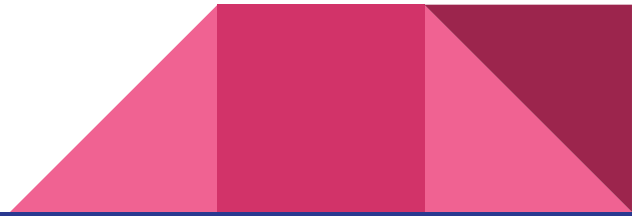
JSON

- Immediate mode converts data to and from a slice of bytes immediately.
- *bts, err := json.Marshal(&data)*
- *err := json.Unmarshal(bts, &data)*
- Wrapper mode wraps around a Reader or a Writer to send data directly along the route.
- *err := json.NewEncoder(wtr).Encode(&data)*
- *err := json.NewDecoder(rdr).Decode(&data)*



Template

- The *html/template* package adds data into an html file to be shown to the user.
- It is probably the most common converter.
- It only works for a writer, it is not helpful to have a reader form.
- Most webpages can use the template package to customize the page for the user, based on who they are, or where they are.



Template

- Usually declared in the package level:
- *var tpl *template.Template*
- Afterwards in main or an init function, usually parses html files:
- *tpl = template.Must(template.ParseGlob(“templates/*.gohtml”))*
- In http requests, it is usually actually used:
- *err := tpl.ExecuteTemplate(res, “index”, &data)*