Databind

Release 0.7.0

Adam Thompson-Sharpe

Sep 01, 2021

CONTENTS

1 Getting Started **3**
- 1.1 Library . . . 3
- 1.2 Installation . . . 3
- 1.3 Creating a Project . . . 3
- 1.4 Writing Code . . . 4
- 1.5 Building . . . 4
- 1.6 Additional Files . . . 4
- 1.7 See Examples . . . 4

2 Syntax **5**

3 Databind CLI **7**
- 3.1 What Can Be Compiled . . . 7
- 3.2 Using the CLI . . . 7

4 Databind Configuration **9**
- 4.1 Configuration File . . . 9
- 4.2 Example Config . . . 9
- 4.3 CLI Arguments . . . 9

5 Macros **11**
- 5.1 Macros that use Databind code . . . 11
- 5.2 Macros that call other macros . . . 11
- 5.3 Macros that define functions . . . 12
- 5.4 Files for macros . . . 14

6 Global Vars **15**
- 6.1 Types . . . 15
- 6.2 Using Global Vars . . . 15
- 6.3 When to use . . . 16

7 Folder Structure **17**

8 Examples **19**
- 8.1 Function Examples . . . 19
- 8.2 If/Else Examples . . . 21
- 8.3 Objective Examples . . . 22
- 8.4 Variable Examples . . . 23
- 8.5 While Examples . . . 25

Contents:

CHAPTER

ONE

GETTING STARTED

Get started with Databind.

1.1 Library

If you're looking to use the Databind library in your own Rust project, then look at the docs hosted on docs.rs.

1.2 Installation

Databind is build and installed from source using cargo. With cargo installed, run `cargo install databind` to get the latest version. If Rust is in your path, then you should be able to access the CLI by running `databind` in any command line.

Built binaries are also available on the GitHub releases page.

1.3 Creating a Project

To create a new project, use the `databind create` command.

```
USAGE:
databind create [OPTIONS] <NAME>

FLAGS:
-h, --help       Prints help information
-V, --version    Prints version information

OPTIONS:
--description <DESCRIPTION>    The pack description [default: A databind pack]
--path <PATH>                  The path to create the pack in

ARGS:
    <NAME>    The name of the project
```

Example use:

`databind create my_project` to create a new project in a folder called `my_project`.

`databind create --description "My first project" my_project` to create a new project with the description `My first project`.

`databind create --path . my_project` to create a new project in the current directory. Only works if empty.

1.4 Writing Code

Below is the default `main.databind` file. `.databind` files **can only be used** to contain function definitions.

```
func main
    tag load
    tellraw @a "Hello, World!"
end
```

First, a function named main is defined. The name can be changed, it doesn't have to be main. Then, it is tagged with `load`. This tag is normal to datapacks and means that a function will run when the datapack is initially loaded. After that, an ordinary `tellraw`, and then `end` to close the function definition.

When compiled, this will create a file called `main.mcfunction` that contains the following:

```
tellraw @a "Hello, World!"
```

A `load.json` file will also be generated in `minecraft/tags/functions` to give the function a load tag.

1.5 Building

To build your project, run `databind` in the root directory of your project. Alternatively, you can run `databind <PATH>` where `<PATH>` is the path to your project.

1.6 Additional Files

You are able to create as many `.databind` files and as many namespaces as you'd like. You are also able to mix normal `.mcfunction` files with `.databind` files, meaning you don't have to have a project that only uses Databind. This is helpful if you want to convert a normal datapack to a Databind project. Databind files cannot contain anything other than function definitions, so something such as this alone in a `.databind` file:

```
say Hello, World!
```

Would not generate any output.

1.7 See Examples

If you want to see some examples of language features, go to the *Examples*. Otherwise, you may continue to the next page.

CHAPTER

TWO

SYNTAX

Syntax	Notes
`var varName := <int>`	Define a new variable
`obj <objective_name> <objective>`	Define a new scoreboard objective
`sobj <target> <objective> <assignment operator> <int>`	Set the value of an objective for a given target (eg. `@a` or `PlayerName`)
`var varName <assignment operator> <int>`	Update the value of an existing variable
`tvar varName`	Used to test variables in `if` commands (eg. `execute if tvar varName matches 1`)
`func name`	Define a function. Generates a new mcfunction file
`!def macro($arg1, $arg2)`	Define a macro. See the *macros page* for more information
`?macro("arg1", "arg2")`	Calls a macro. See the *macros page* for more information
`!end`	Ends a macro definition. See the *macros page* for more information
`call <function>`	Call a function. Can infer namespace based on directory (see function calling example)
`runif <condition>`	Starts an if statement
`else`	Runs if an if statement's condition was not true
`while <condition>`	Create a while loop. Condition should be something passable to `execute if`
`end`	Close a function, while loop, or if statement
`sbop`	Shorthand for `scoreboard players operation`
`gvar varName`	Can be used with a scoreboard operation as such: `sbop gvar var1 += gvar var2`
`delvar varName` OR `delobj objName`	Delete a variable or objective. Can be used interchangeably
`%`	Used to escape keywords (eg. `say %call a function` -> `say call a function`)
Assignment Operators	
`+=`	Add to a variable
`-=`	Subtract from a variable
`=`	Set the value of a variable

CHAPTER

THREE

DATABIND CLI

3.1 What Can Be Compiled

Databind compiles Databind projects (see *Creating a Project*). Databind will look for included files (`**/*.databind` by default) and leave other files alone.

Note that the namespace inference used for `func` assumes a proper file structure (`<datapack>/data/<namespace>/functions` for functions), but it **does not check if this is the case.** A `minecraft/tags/functions/` folder may be generated in an unexpected place if an invalid folder is passed.

3.2 Using the CLI

```
USAGE:
    databind [FLAGS] [OPTIONS] <PROJECT>
    databind [FLAGS] [OPTIONS] <SUBCOMMAND>

FLAGS:
    -h, --help             Prints help information
        --ignore-config    Ignore the config file. Used for testing
    -V, --version          Prints version information

OPTIONS:
    -c, --config <FILE>      Configuration for the compiler
    -o, --out <DIRECTORY>    The output file or directory [default: out]

ARGS:
    <PROJECT>    The Databind project to compile

SUBCOMMANDS:
    create    Create a new project
    help      Prints this message or the help of the given subcommand(s)
```

3.2.1 From an Installation

When installed, you can access the CLI by running `databind` in any command line. Running `databind --help` will output the text above.

3.2.2 With `cargo run`

After building Databind yourself, you can use `cargo run` to run it. Everything works almost the exact same. You just need to add two dashes (`--`) after `run` (eg. `cargo run -- --help`).

CHAPTER

FOUR

DATABIND CONFIGURATION

4.1 Configuration File

Databind can be configured via the `databind.toml` generated in the project's root. A config file can also be passed with the `-c` or `--config` option.

This table represents the default values of the options if no config changes are made.

Option	Notes
`inclusions = ["**/*.databind"]`	Specify what files to compile using globs
`exclusions = []`	Specify what files not to copy over/compile using globs
`output = "out"`	The output file or folder

4.2 Example Config

Below is a configuration file with all of the above settings.

```
inclusions = ["**/*.databind"]
exclusions = []
output = "out"
```

4.3 CLI Arguments

Most options that can be set in the `databind.toml` file can also be set using CLI arguments.

Example use:

```
databind -c config.toml -o ./target ./datapack
```

CHAPTER

FIVE

MACROS

Macros in Databind are advanced functions that allow you to take arguments, unlike traditional mcfunctions. All arguments must be surrounded by double quotes (`"`). Here is a macro that says "Hello" to a name you pass:

```
!def say_hello($name)
    say Hello, $name!
!end
```

And here is how it would be called:

```
?say_hello("World")
```

The macro call above would become the following when compiled:

```
say Hello, World!
```

As you can see, the `$name` in the body of the macro was replaced with the `"World"` string that was passed to it.

5.1 Macros that use Databind code

Macros are able to use Databind code just like any other place in a `.databind` file. Here is a macro that creates a variable with a name that is passed to it, then announces a message to all players:

```
!def create_var($name)
    var $name := 5
    tellraw @a "A variable named $name was created."
!end
```

5.2 Macros that call other macros

Macros are also able to call other macros and pass arguments to them.

```
!def macro_1($name)
   say Hello, $name!
!end

!def macro_2($name)
    # There is a % before 'call' here because 'call' is a Databind keyword
    # See the syntax table for info on escaping keywords
    say I am about to %call macro_1
```

(continues on next page)

(continued from previous page)

```
    ?macro_1("$name")
!end
```

Keep in mind that macro arguments must be surrounded by double quotes, which is why `macro_2`'s call of `macro_1` is `"$name"` instead of just `$name`.

5.3 Macros that define functions

Since macros can use any Databind code, this also means that they're able to define functions. This makes it possible to create macros that set up a series of functions to avoid copy + pasting code.

```
!def create_toggle_function($funcname)
    # This appends '_load' to the end of the function name
    func $funcname_load
    tag load
        var $funcname_state := 0
        var $funcname_toggled := 0
    end

    # This appends '_on' to the end of the function name
    func $funcname_on
        say $funcname has been enabled
        var $funcname_state = 1
    end

    # This appends '_off' to the end of the function name
    func $funcname_off
        say $funcname has been disabled
        var $funcname_state = 0
    end

    # This appends '_toggle' to the end of the function name
    func $funcname_toggle
        say Toggling $funcname
        execute if tvar $funcname_state matches 1 run var $funcname_toggled = 1
        execute if tvar $funcname_state matches 1 unless tvar $funcname_toggled␣
↪matches 0 run call $funcname_off
        execute if tvar $funcname_state matches 0 unless tvar $funcname_toggled␣
↪matches 1 run call $funcname_on
        var $funcname_toggled = 0
    end
!end
```

This entire macro creates four functions per call:

1. A function that loads when the datapack is loaded (`$funcname_load`)
2. A function that enables something (`$funcname_on`)
3. A function that disables something (`$funcname_off`)
4. A toggle function (calls `$funcname_on` when disabled and `$funcname_off` when enabled)

These functions can all be created by running the following line:

```
?create_toggle_function("my_function")
```

Of course, creating functions that only say "Enabled" or "Disabled" isn't useful in most situations. What would be useful is to be able to pass commands to run when the function is enabled, disabled, or toggled.

This is entirely possible using macros due to the fact that the arguments passed can be multiline.

If we change the macro above to look like this:

```
!def create_toggle_function($funcname, $on_cmds, $off_cmds)
    # This appends '_load' to the end of the function name
    func $funcname_load
    tag load
        var $funcname_state := 0
        var $funcname_toggled := 0
    end

    # This appends '_on' to the end of the function name
    func $funcname_on
        var $funcname_state = 1
        $on_cmds
    end

    # This appends '_off' to the end of the function name
    func $funcname_off
        var $funcname_state = 0
        $off_cmds
    end

    # This appends '_toggle' to the end of the function name
    func $funcname_toggle
        execute if tvar $funcname_state matches 1 run var $funcname_toggled = 1
        execute if tvar $funcname_state matches 1 unless tvar $funcname_toggled␣
↪matches 0 run call $funcname_off
        execute if tvar $funcname_state matches 0 unless tvar $funcname_toggled␣
↪matches 1 run call $funcname_on
        var $funcname_toggled = 0
    end
!end
```

We're now able to pass commands to run when the function is enabled and disabled. If we wanted a command that summoned an armor stand when enabled and killed it when disabled, we could call the macro like this:

```
# This formatting is not required, it's just to make the code
# easier to read
?create_toggle_function(
    "astand",

    "summon armor_stand ~ ~ ~
     say Created armor stand",

    "kill @e[type=armor_stand]
     say Killed armor stand",
)
```

When compiled to a datapack, if we wanted to run our toggle function in-game, we could run the following:

```
/function namespace:astand_toggle
```

5.4 Files for macros

Any file whose name starts with an `!` symbol is able to define macros that work anywhere in the project. These files, if they only contain macros, should generally be placed right in the `src/` directory as opposed to in a namespace's `functions/` directory, however you can place them wherever you'd like.

It's important to note that the reason the `!` was chosen is that the compiler goes through the `src/` directory in alphabetical order. This means that if you, for example, have two namespaces, `abc` and `xyz`, macros defined in `xyz` will not be available in `abc`. A good idea is to begin the names of any folders containing macro definitions with an `!`, similar to the files. That way, they are always compiled first.

Macros that contain calls to other macros can be defined in any order. If you have the following two macros:

```
!def macro_1()
    say Macro 1
!end
```

```
!def macro_2()
    say Macro 2
    ?macro_1()
!end
```

You don't have to define `macro_1` before `macro_2`; it's only important that they're both defined before `macro_2` is called. A project using macros might have a file structure similar to this:

```
project_root
│   databind.toml
└───src
    │   pack.mcmeta
    ├───!macros
    │       !my_macro.databind
    └───data
        └───namespace
            └───functions
                    main.databind
```

CHAPTER

SIX

GLOBAL VARS

You can define global variables with a file called `vars.toml` in the project root. Keys and values aren't put in a section of the `.toml`, they're just in the file. For example:

```
name="World"
```

This defines a global variable `name` that can be used in your code.

6.1 Types

The TOML format supports datatypes other than just strings, such as booleans and integers. Types that aren't strings are converted to strings. Booleans that are `true` are turned into `1`, and `false` ones are turned into `0`. Floats like `1.0` are truncated, but floats with non-zero decimals are left alone.

6.2 Using Global Vars

To use a global variable in your code, use an `&` symbol followed by the variable name. Like this:

```
say Hello, &name!
```

Which, with the `vars.toml` defined above, becomes:

```
say Hello, World!
```

Instances of `&varname` are directly replaced, meaning that escaping them with a `%` symbol doesn't work. This means that the following code:

```
say Hello, %&name!
```

won't stop the replacement of `&name`.

6.3 When to use

Global variables are useful to let users more easily configure aspects of your datapack. This does mean that the project must be recompiled whenever the configuration is changed, and that users must have Databind downloaded to use the project. If you are only configuring number values, eg. an amount of time to wait for something, then it might be easier for people using your datapack to have a `config.mcfunction` file somewhere in the project that sets scoreboard values.

CHAPTER

SEVEN

FOLDER STRUCTURE

How the folder structure of Databind works.

In a project started with `databind create`, the file structure might look something like this:

```
project_root
│   databind.toml
│   LICENSE
│   README.md
└──src
    │   pack.mcmeta
    │   pack.png
    └──data
        └──namespace
            └──functions
                    main.databind
```

All of the Databind-related files (other than the configuration file) are contained in the *src/* directory. Other files such as the project's license and the README are just in the root. These files are not generated by default, but they've been added in the example to show where they might be placed.

It's possible to create a project without using `databind create`, but it's not ideal and bugs caused by it generally won't be fixed.

CHAPTER

EIGHT

EXAMPLES

Various examples on how to use Databind and its features.

Contents:

8.1 Function Examples

Examples using functions.

Contents:

8.1.1 Calling

Different ways to call a function.

`function` command

Built into mcfunctions. Requires a namespace.

`example/src/data/example/functions/main.databind`

```
func example_func
say Hello, World!
end

func main
function example:example_func
end
```

`call` (infer namespace)

Add namespaces to functions while compiling. Allows more freedom with directory names.

`example/src/data/example/functions/main.databind`

```
func example_func
say Hello, World!
end

func main
```

(continues on next page)

(continued from previous page)

```
call example_func
end
```

Compiled, `call example_func` becomes `function example:example_func`.

`call` (explicit namespace)

`example/src/data/example/functions/main.databind`

```
func example_func
    say Hello, World!
end

func main
    call example:example_func
end
```

Effectively the same as the `function` command.

8.1.2 Simple Function

Example

A function that increments a counter and logs when it's run.

`example/src/data/example/functions/main.databind`

```
func load
tag load
    var counter := 0
end

func example
    tellraw @a "Example_function run"
    var counter += 1
end
```

Compiled

`example/out/data/example/functions/load.mcfunction`

```
scoreboard objectives add counter dummy
scoreboard players set --databind counter 0
```

`example/out/data/example/functions/example.mcfunction`

```
tellraw @a "Example_function run"
scoreboard players add --databind counter 1
```

8.2 If/Else Examples

Examples using if/else statements.

If statements use several files, so compiled output is not shown in the examples.

Contents:

8.2.1 Single If Statement

A lone if statement.

Example

`example/src/data/example/functions/main.databind`

```
func main
tag load
    var test := 1
    runif tvar test matches 1
        say Test is equal to 1
    end
end
```

8.2.2 If/Else

An if statement with an `else` block.

Example

`example/src/data/example/functions/main.databind`

```
func main
tag load
    var test := 1
    runif tvar test matches 1
        say Test is equal to 1
    else
        say Test is not equal to 1
    end
end
```

8.2.3 Nested If Statements

Multiple if statements inside of each other.

`example/src/data/example/functions/main.databind`

```
func main
tag load
    var i := 0
    var j := 0
    runif tvar i matches 0
        runif tvar j matches 0
            say i is 0 and j is 0
        else
            say i is 0 and j is not
        end
    end
end
```

8.3 Objective Examples

Examples using objectives.

Contents:

8.3.1 Create Objective

Create a scoreboard objective.

Example

```
# Create an objective points and set everyone's score to 100
obj points dummy
sobj @a points = 100
```

Compiled

```
scoreboard objectives add points dummy
scoreboard players set @a points 100
```

8.3.2 Deletion

Example

Define an objective and delete it.

```
obj objective dummy
delobj objective
# or
delvar objective
```

Compiled

```
scoreboard objectives add objective dummy
scoreboard objectives remove objective
```

8.3.3 Scoreboard Operations

Example

Define two objectives and use a scoreboard operation to multiply the first.

```
obj obj1
obj obj2
sobj @a obj1 = 5
sobj @a obj2 = 2
sbop @a obj1 *= @a obj2
```

Compiled

```
scoreboard objectives add obj1 dummy
scoreboard objectives add obj2 dummy
scoreboard players set @a obj1 5
scoreboard players set @a obj2 2
scoreboard players operation @a obj1 *= @a obj2
```

8.4 Variable Examples

Examples using variables.

Contents:

8.4.1 Create, Modify & Test

Example

```
# Create a variable called example and set it to 2
var example := 2
# Add 1 to example
var example += 1
# Subtract 2 from example
var example -= 2
# Set example to 1
var example = 1
# Say something if example is 1
execute if tvar example matches 1 run say Variable example is equal to 1!
```

Compiled

```
scoreboard objectives add example dummy
scoreboard players set --databind example 2
scoreboard players add --databind example 1
scoreboard players remove --databind example 2
scoreboard players set --databind example 1
execute if score --databind example matches 1 run say Variable example is equal to 1!
```

8.4.2 Deletion

Example

Define a variable and delete it.

```
var variable := 1
delvar variable
# or
delobj variable
```

Compiled

```
scoreboard objectives add variable dummy
scoreboard players set --databind variable 5
scoreboard objectives remove variable
```

8.4.3 Scoreboard Operations

Example

Define two variables and use a scoreboard operation to multiply the first.

```
var variable1 := 5
var variable2 := 2
sbop gvar variable1 *= gvar variable2
```

Compiled

```
scoreboard objectives add variable1 dummy
scoreboard players set --databind variable1 5
scoreboard objectives add variable2 dummy
scoreboard players set --databind variable2 2
scoreboard players operation --databind variable1 *= --databind variable2
```

8.5 While Examples

Examples using while loops.

Contents:

8.5.1 For Loop

A for loop-like while loop.

Example

`example/src/data/example/functions/main.databind`

```
func load
tag load
    var i := 10
    while tvar i matches 1..
        tellraw @a "Variable i is above 0"
        var i -= 1
    end
    tellraw @a "Variable i is at 0"
end
```

Compiled

When while loops are compiled, functions with random characters at the end are created. In compiled examples, these characters will be `abcd`.

`example/out/data/example/functions/load.mcfunction`

```
scoreboard objectives add i dummy
scoreboard players set --databind i 10
function example:while_abcd
tellraw @a "Variable i is at 0"
```

`example/out/data/example/functions/while_abcd.mcfunction`

```
execute if score --databind i matches 1.. run function example:condition_abcd
```

`example/out/data/example/functions/condition_abcd.mcfunction`

```
tellraw @a "Variable i is above 0"
scoreboard objectives remove --databind i 1
function example:loop_abcd
```

8.5.2 Loop Until False

Use an integer as a boolean to loop until false.

Example

`example/src/data/example/functions/main.databind`

```
func load
tag load
    var bool := 1
    while tvar bool matches 1
        tellraw @a "Bool is true"
    end
end
```

Compiled

When while loops are compiled, functions with random characters at the end are created. In compiled examples, these characters will be `abcd`.

`example/out/data/example/functions/load.mcfunction`

```
scoreboard objectives add bool dummy
scoreboard players set --databind bool 1
function example:while_abcd
```

`example/out/data/example/functions/while_abcd.mcfunction`

```
execute if score --databind bool matches 1 run function example:condition_abcd
```

`example/out/data/example/functions/condition_abcd.mcfunction`

```
tellraw @a "Bool is true"
function example:while_abcd
```