marvin Documentation

Release 0.3

Justus Adam

Aug 04, 2017

Contents

Marvin is a Haskell framework for creating chat bots, inspired by Hubot. Marvin aims to recreate the ease of use and straightforward API of Hubot, and combine it with the safety guarantees and purity of Haskell and the higher efficiency.

If you are new to marvin you may want to begin with the *Getting started* section or the *how to script with marvin* section to get a sense of how scripting works with marvin.

CHAPTER 1

Links

- Hackage
- GitHub Repository
- Bugtracker
- Documentation repository and bugtracker
- Slack channel (*signup instructions*)

CHAPTER 2

A quick snippet of code

```
module MyScript where

import Marvin.Prelude

script :: (IsAdapter a, SupportsFiles a) => ScriptInit a
script = defineScript "my-script" $ do
    hear "sudo (.+)" $ do
        match <- getMatch

        reply $(isL "All right, i'll do #{match !! 1}")

    respond "repeat" $ do
        message <- getMessage

        send $(isL "You wrote #{message}")

    respond "what is in file ([\\w\\._/-]+)\\??" $ do
        match <- getMatch
        let file = match !! 1

        contents <- liftIO $ readFile file

        send contents

    respond "upload file ([\\w\\._/-]+)" $ do
        [_, filepath] <- getMatch
        chan <- getChannel
        f <- sendFile filepath [chan]
        case res of
            Left err -> send $(isL "Failed to share file: #{err}")
            Right _  -> send "File successfully uploaded"

    enterIn "random" $ do
        user <- getUser
        send $(isL "Hello #{user^.username} welcome to the random channel!")
```

```
    fileSharedIn "announcements" $ do
        file <- getFile
        safeFileToDir file "shared-files"
```

CHAPTER 3

Testing and Talking

There's a slack channel where you can ask questions or play around with a test instance of marvin.

It's currently invite-only, so send me an email if you would like to join.

CHAPTER 4

Contents:

Getting started

Marvin projects basically comprise of a collection of individual scripts and a main file which ties them all together.

To start a new project marvin provides an initializer, called `marvin-init`. The initializer will set up a new project for you, including a sample script, the main file and the Haskell project configuration to make compiling smooth and easy.

Note: Always run `marvin-init` in an empty directory, as it will place certain files there and overwrite existing files with the same name.

If you install marvin through cabal (`cabal install marvin`) or stack (`stack install marvin`) it will install the initializer as well and add it to your path. To see the options of the initializer run `marvin-init --help` on your command line.

```
marvin-init ~ make a new marvin project

Usage: marvin-init BOTNAME [-a|--adapter ID]

Available options:
-h,--help                Show this help text
-a,--adapter ID          id of the adapter to use
```

Information on Adapters and their id's can be found in the *Adapters* section.

Installing marvin

You can get a release version of marvin on Hackage

However the recommended way to install this package is via stack. The marvin package is part if the stack lts as of `lts-8.5`. You can let stack do the resolving for you if you've added marvin in your `.cabal` file you can simply run

`stack solver --update-config` and it will choose the right versions for you.

After that `stack build` will pull and install marvin for you.

Important: Marvin uses the `text-icu` library for regexes. It therefore requires the `-dev` version of the `icu` C library.

Linux Simply install the `-dev` version of the icu library.

For instance `apt install libicu-dev` on Ubuntu.

OSX You also need the icu library. If you are using Homebrew you are looking for the `icu4c` package (`brew install icu4c`). Because OSX also provides some headers you will also need to link the headers manually. If you are using stack to build your projects the easiest way is to add the following lines to `$HOME/.stack/config.yaml`.

```
extra-include-dirs:
    - /usr/local/opt/icu4c/include
extra-lib-dirs:
    - /usr/local/opt/icu4c/lib
```

alternatively you can pass the paths via `--extra-include-dirs` and `extra-lib-dirs` to the `stack build` and `stack install` command.

Scripts

The functionality for your marvin installation is split into individual parts, called scripts. Each script is some Haskell structure created with `defineScript`. Scripts can be user defined or be included externally.

External scripts

You can include external scripts in the form of a library. To do this you must add the library name to the `.cabal` and `stack.yaml` file of your project.

You can find more information on external scripts and an example `external-scripts.json` file in the *external scripts section*

User defined scripts

You can also write some scripts yourself. Typically scripts are a Haskell source file which defines a `script` value.

As an example, a "hello world" script.

```haskell
-- file named "HelloWorld.hs" (must be the same as module name + ".hs")
module HelloWorld where

import Marvin.Prelude

-- This type signature is necessary to help the compiler
script :: IsAdapter a => ScriptInit a
script = defineScript
            "hello-world" -- script name (for logging and config)
            $ do  -- here follows the actual script definition
                ...
```

You can find more information on the actual script content in the *Writing marvin scripts* section.

The main file

This file (ususally called `Main.hs`) ties the scripts together and defines the *Adapters* which your marvin project uses.

Note: If you use the initializer `marvin-init` the main file will already be defined for you and registered in the `.cabal` file.

The file must be a Haskell source file i.e. end with `.hs` and be mentioned in the `main-is` section of your `.cabal` file. It will look someting like this:

```
-- import marvin runner
import Marvin.Run
-- imports chosen adapter
import Marvin.Adapter.Slack

-- import all scripts
import qualified HelloWorld
import qualified MyScript

-- list of all scripts to use
scripts :: [ScriptInit SlackRTMAdapter]
scripts = [ HelloWorld.script
          , MyScript.script
          ]

main :: IO ()
main = runMarvin scripts
```

You can write the main file yourself, but this can get tedious as you add more and more scripts. To make this easier Marvin includes a utility which allows you to let the main file be generated automatically, called *The marvin preprocessor (marvin-pp)*. `marvin-pp` creates the main file dynamically at compile time by scanning your project for scripts. You can add external scripts by adding an *external-scripts.json* file and `marvin-pp` will add those to your main file then.

To use `marvin-pp` simply add an empty main file, except for this line: `{-# OPTIONS_GHC -F -pgmF marvin-pp -optF --adapter -optF slack-rtm #-}` (this is what `marvin-init` does as well).

Important: The `marvin-pp` generator is a compile time preprocessor and thus its output is often cached by your build system. As a result you have to run `cabal clean` or `stack clean` after you added or removed a script to force the build system to regenerate the main file.

Writing marvin scripts

Each script in marvin is a Haskell module that defines a value `script` with the type `ScriptInit`. This value contains the code necessary to set up the script and will be run automatically by the marvin runner at startup. It returns the script.

It does not matter where in the module you define this value, only that it sits at the top level so that the main file can import it. You can define arbitrary other values in the top level of your script, such as mutable variables and you can import any Haskell library you like including other marvin scripts (for *Data sharing*).

Script boilerplate

Since each script is a Haskell module the module name and the file name must match. I.e. a script module MyScript must be in a file called `MyScript.hs`. Furthermore the module and file name may only contain word characters and the underscore _ and must begin with an upper case letter.

Note: A file which starts with an underscore _ or dot . is ignored by the automatic script discovery of the main file. This is a way to hide unfinished scripts from being included in the program.

When you have created your source file you should first import marvins prelude `Marvin.Prelude` (something like marvins standard library). It contains all the marvin related functions you will need.

Hint: You dont *have* to use `Marvin.Prelude`. The prelude is just a convenient collection of other modules, you can also import just the ones you need directly, but this is only recommended for people experienced with Haskell.

```haskell
-- File: MyScript.hs
module MyScript where -- Module definition (must match filename)

-- import the prelude
import Marvin.Prelude

-- import other modules and libraries you need

script :: IsAdapter a => ScriptInit a
script = defineScript "my-script" $ do
    -- here follows the actual scripting part
```

Lastly we define a value called `script` with the type signature `IsAdapter a => ScriptInit a`. This complicated looking type signature ensures our script will work with any adapter that satisfies the adapter type class (adapter interface). Here we call the function `defineScript` which takes an id string and an initializer block.

The id string is used fo two things

1. Scoping the config, i.e. the config for this script will be stored in the `scripts.<id-string>` key.
2. Logging. All logging messages from this script will be prefixed with `scripts.<id-string>`.

Usually the id string is some variation on the name of the script file and module.

The initializer block is where the actual scripting starts.

The initializer block

The initializer block is the code that is run when you start marvin.

First and foremost this block is used to add new reactions to your marvin script, which is most likely the main part of your scripts functionality.

But you can do a variety of other things here such as *define periodic tasks*, *read data* and *define mutable variables* for state tracking or data sharing.

The reaction Monad

```
data BotReacting a d r = ... deriving (Monad, MonadIO, MonadReader (BotActionState a␣
↪d)
                                      , MonadLogger, MonadLoggerIO)
```

The reaction monad offers basically four different capabilities.

1. **`MonadIO` allows the user to execute arbitrary `IO` actions by lifting them with `liftIO`.** This can be things such as performing HTTP requests, reading files etc.
2. **`MonadReader (BotActionState a d)` allows read access to the data carried by the monad.** In general you dont need to use this directly as functions such as *getUser* are much more convenient to use. However the readable data you get by using `ask` contains not only the payload which is of type `d` and different depending on each handler function, but also access to the adapter, the config and script id. And is therefore capable of
3. **Accessing the adapter.** This enables the handler to communicate. Functions such as *send* and *messageChannel* can be used to send messages to the chat application.
4. `MonadLogger(IO)` Allows you to write log messages using functions from the monad-logger package by importing `Control.Monad.Logging`.

Reaction functions

There are several functions for reacting to some event happening in you chat application. The type of reaction influences the kind of data available in the reaction handler. The data available in the handler can be seen listed in a tuple in the `BotReacting` monad. For instance `BotReacting a (User' a, Channel' a, Message, Match, TimeStamp) ()` will have access to a user, a channel, a message and so on. Functions for getting access to this data are listed in functions for handlers

The basic structure of a reaction is `<reaction-type> <matcher> <handler>`.

`<reaction-type>` Is one of the reaction functions, like hear or respond (more are to follow).

This also determines the type of data available in the handler.

`<matcher>` Is some selection criterium for which events you wish to handle, and also often influences the contents of the data available to the handler.

For instance for hear and respond this is a regex. The message will only be handled if the regex matches, and the result of the match, as well as the original message is available to the handler later.

`<handler>` Arbitrary code which runs whenever a matched event occurs.

Has access to message specific data (like a regex match of the message). Can communicate with the chat (send messages to people or channels).

Reacting to messages

There are two ways to react to a text message. A reaction defined with `hear` will trigger on any incoming message which matches its defined pattern (a regular expresion). By contrast reactions defined with `respond` will only trigger if the bot itself is being adressed. How one adresses the bot depends on the concrete adapter. However typically prefixing the message with the bots name or sending a direct message (if the adapter supports this) to the bot ususally trigger these reactions.

In the handler that is being attached you have access to the match groups of the regex with `getMatch`, the user who sent the message (`getUser`), the full text of the message (`getMessage`), the channel to which the message was posted (`getChannel`) and a timestamp for when the message arrived (`getTimeStamp`).

The type signature for both is the same.

```
hear, respond :: Regex -> BotReacting a (User' a, Channel' a, Match, Message,␣
↪TimeStamp) () -> ScriptDefinition a ()
hear regex handler = ...
respond regex handler = ...
```

A working example could be something like this:

```
defineScript "test" $ do

    hear "\\bmarvin\\b" $ do
        user <- getUser

        send $(isL "Yes #{user^.username}, that is my name")

    respond "^\\bsudo\\b(.+)" $ do
        match <- getMatch
        send #(isL "I will do #{match !! 1} immediately!"

    hear ".*" $ do
        channel <- getChannel
        unless (channel^.name == "#nsa") $ do
            message <- getMessage
            messageChannel "#nsa" $(isL "Psst, this message was just posted in #
↪{channel^.name}: #{message}")
```

Reacting to the topic

You can react to changes in the topic in two different ways. Using `topic` the handler will trigger whenever the topic in any channel changes. Using `topicIn` you can provide the name of a channel which you wish to watch for changes in the topic and the handler will only be run for changes to the topic in the specified channel.

In the handler you have access to the user which triggered the change (`getUser`), the channel in which the topic was changed (`getChannel`), the new topic (`getTopic`) and a timestamp for when this change occurred (`getTimeStamp`).

```
topic :: BotReacting a (User' a, Channel' a, Topic, TimeStamp) () -> ScriptDefinition␣
↪a ()
topic handler = ...

topicIn :: Text -> BotReacting a (User' a, Channel' a, Topic, TimeStamp) () ->␣
↪ScriptDefinition a ()
topicIn channelName handler = ...
```

Note: The `Topic` type is just for readability, it is just an alternate name for `Text`.

Reacting to changes in channel participants

Marvin can react both to people joining and leaving channels. `enter` triggers when a user enters **any** channel in which the bot is also participating. `enterIn` takes as an argument the name of a channel and ony triggers if a user joins **that** specific channel. `exit` triggers when a user leaves **any** channel in which the bot is also participating. `exitFrom` takes as an argument the name of a channel and ony triggers if a user leaves **that** specific channel.

All of these handlers have access to the channel which the user joined/left (`getChannel`), the user that joined/left (`getUser`) and a timestamp for when this occurred (`getTimeStamp`)

```
enter :: BotReacting a (User' a, Channel' a, TimeStamp) () -> ScriptDefinition a ()
enter handler = ...

enterIn :: Text -> BotReacting a (User' a, Channel' a, TimeStamp) () ->␣
↪ScriptDefinition a ()
enterIn channelName handler = ...

exit :: BotReacting a (User' a, Channel' a, TimeStamp) () -> ScriptDefinition a ()
exit handler = ...

exitFrom :: Text -> BotReacting a (User' a, Channel' a, TimeStamp) () ->␣
↪ScriptDefinition a ()
exitFrom channelName handler = ...
```

Reacting to files

The `fileShared` handler is invoked any time a file is shared in **any** channel the bot is participating in. By contrast the `fileSharedIn` handler takes as its first argument a channel name and only reacts to files being shared in that channel.

The handlers provide access to the user who shared the file (`getUser`), the channel in which the file was shared (`getChannel`), the `RemoteFile` object, containing information about the file being shared (`getRemoteFile`) and a timestamp for when the file was shared `getTimeStamp`).

```
fileShared :: BotReacting a (User' a, Channel' a, TimeStamp) () -> ScriptDefinition a␣
↪()
fileShared handler = ...

fileSharedFrom :: Text -> BotReacting a (User' a, Channel' a, TimeStamp) () ->␣
↪ScriptDefinition a ()
fileSharedFrom channelName handler = ...
```

Generic functions for handlers

The `send` function

```
send :: (IsAdapter a, Get m (Channel' a)) => Text -> BotReacting a m ()
send msg = ...
```

The `send` function is used to post messages to the same channel from which the event that triggered the handler came.

Explanation of the type signature:

`IsAdapter a` We require the saved `a` in `BotReacting` to be an adapter. This means this function actually interacts with the chat service (sends a message in this case).

`Get m (Channel' a)` The data in the monad must have an originating `Channel` in it somewhere to which the message will be posted. This is true for most handler functions, for instance hear, respond, enter all enter, exit and topic handlers.

The `reply` function

```
reply :: (IsAdapter a, Get m (User' a), Get m (Channel' a)) => Text -> BotReacting a␣
↪m ()
reply msg = ...
```

Reply is similar to *send*. It posts back to the same channel the original message came from, but it also references the author of the original message.

The `messageChannel` function

```
messageChannel :: (HasConfigAccess m, AccessAdapter m, IsAdapter (AdapterT m)) => L.
↪Text -> L.Text -> m ()
messageChannel channelName message = ...
```

Similar to *send* and *reply* this functions sends a message to the channel with the (human readable) `channelName`. If instead of a name you have a `Channel a` object, you can use *messageChannel'*.

The `messageChannel'` function

```
messageChannel' :: (HasConfigAccess m, AccessAdapter m, IsAdapter (AdapterT m),␣
↪MonadIO m) => Channel (AdapterT m) -> L.Text -> m ()
messageChannel' channel message = ...
```

Like *messgeChannel* but references the channel by channel object, rather than name.

The `getMatch` function

```
getMatch :: HasMatch m => BotReacting a m Match
```

Retrieves the result of a regex match inside a handler monad whos state supports it. Examples are the handlers for hear and respond.

Regex matches are a list of strings. The 0'th index is the full match, the following indexes are matched groups.

The `getMessage` function

```
getMessage :: Get m (Message a) => BotReacting a m (Message a)
```

Retrieves the respond structure for the message this handler is reacting to inside a handler monad whos state supports it. Examples are the handlers for hear and respond.

The `getTopic` function

```
getTopic :: HasTopic m => BotReacting a m Topic
```

This function is usable in handlers which react to changes of the topic of a channel. It returns the *new* topic.

Note: The `Topic` type is just for readability, it is just an alternate name for `Text`.

The `getChannel` function

```
getChannel :: Get m (Channel' a) => BotReacting a m (Channel a)
```

Usable in most handler functions, this function returns the channel in which some event occurred.

The `getUser` function

```
getUser :: Get m (User' a) => BotReacting a m User
```

Usable in all handler functions which involve an acting user (most). Returns the user who triggered an event.

Persistence

In memory

On disk

Periodic tasks

Data sharing

The abstract data model of marvin

The data model of marvin is that for many data types such as a `User` or a `RemoteFile` marvin leaves the concrete representation of the data structure to the used adapter. This is the reason these structures always contain a type variable for the adapter, like `User a`, `Channel a` or `RemoteFile a`. Adapters define these types as part of the implementation of the `IsAdapter` or `HasFiles` typeclasses.

The concrete representation of these types of course depends on the adapters and as such we do not know what the structure looks like. However to ensure some basic interactions marvins `isAdapter` and `HasFiles` typeclass place constraints on the data types int he form of lens class superclasses.

Generally a `Has<field> <stucture> <field-type>` class means that *structure* has a reachable *field* of *field-type*. For more thorough information on lenses see the *Lens quickstart* section, but for just some basics of lenses we can use the operators `.^` to access the field with the lens and `.~` to set the field at the lens.

```
let user1 = ... :: User SomeAdapter
    username = ... :: Lens' (User SomeAdapter) Text
```

```
x^.username -- returns the username

let y = x & username .~ "new_name"

y^.username - now returns "new_name"
```

Runtime configuration

Configuration for marvin is written in the configurator syntax.

The configuration is read-only, aka the program does not alter the config itself. However the config is also auto-reload, meaning that the live system can adapt to changes in the config which are made while the system is running.

Therefore it is recommended that scripts using config values do not cache those values if possible, but reread them instead.

Please note that the config refresh interval means that it takes up to a minute until changes to the config are live.

System config

Configuration pertaining to the system itself is stored under the "bot" key.

```
bot {
    name = "my-bot"
    logging = "INFO"
    adapter = "slack-rtm"
}
```

Script config

Configuration for scripts is automatically scoped. Each script has access to a configuration stored under `script.<script-id>` with the functions `getConfigVal` and `requireConfigVal`. And of course these scripts can have nested config groups.

```
bot {
    name = "my-bot"
}

script {
    script-1 {
        some-string = "foo"
        some-int = 1337
        bome-bool = true
    }
    script 2 {
        nested-group {
            val = false
        }
        name = "Trump"
        capable = false
    }
}
```

Adapter config

Configuration pertaining to a particular adapter is stored under `adapter.<adapter-name>`. The exact nature of the adapter config depends on the adapter itself.

```
bot {
    name = "my-bot"
    logging = "INFO"
}
adapter {
    slack-rtm {
        token = "eofk"
    }
}
```

Example

An example config with all currently available config options (excludes script config as those are user defined).

```
bot {
    # String, one of WARNING, ERROR, INFO, DEBUG, optional, defaults to WARNING
    # Logging level for the bot
    logging = "WARNING"

    # String, optional, default to "marvin", name for the bot
    # Also sometimes used to identify whether a given message should be interpreted
↪as a command
    name = "marvin"

    # String, one of the available adapter identifiers, optional, defaults to "slack-
↪rtm"
    # Adapter to use in the main file.
    # Only used by the preprocessor.
    adapter = "slack-rtm"
}

adapter {
    shell {
        # String, filepath, optional.
        # If present records the history in this file
        history-file = ""
    }
    slack-rtm {
        token = "" # String, required. Authentication token for slack api
    }
    slack-events {
        token = "" # String, required. This token is used to confirm recieved
↪messages come from slack

        # boolean, defaults to true. Whether to use TLS for encryption.
        # Note that slack requires a webhook receiver to be tls protected.
        # Therefore this must be activated unless the server is behind a proxy using
↪tls.
        use-tls = true
        certfile = "" # String (filepath), required if tls is used. As the server
↪needs to use ssl, a certificate is required.
        keyfile = "" # String (filepath), required if tls is used. As the server
↪needs to use ssl, a certificate is required.
```

```
        port = 7000 # Integer, optional. Defaults to 7000. Port on which the server␣
↪listens for requests.
    }
    telegram-poll {
        token = "" # String, required. Authentication token for telegram api
        polling-timeout = 120 # positive integer, optional. Timeout for long polling␣
↪requests
    }
    telegram-push {
        token = "" # String, required. Authentication token for telegram api

        url = "https://..." # String, required. Url of this server. (target for the␣
↪webhook)

        use-tls = false # boolean, defaults to true. Whether to use TLS for␣
↪encryption.
        certfile = "" # String (filepath), required if tls is used. As the server␣
↪needs to use ssl, a certificate is required.
        keyfile = "" # String (filepath), required if tls is used. As the server␣
↪needs to use ssl, a certificate is required.

        port = 7000 # Integer, optional. Defaults to 7000. Port on which the server␣
↪listens for requests.
    }
}
```

Files

Some adapters support up- and download of files. Not all adapters support this and the individual interactions are different between each of the adapters.

API

Marvin encodes its file interaction API in a separate typeclass `SupportsFiles`. Adapters can opt to implement this class to support file interactions. The concrete structure of local and remote files and metadata differs between each adapter. However the API enforces certain basic rules on the data.

Each adapter may implement additional custom functionality, but if you wish to be adapter agnostic, for instance when implementing a library of reactions, you can rely on the API interface.

Remote Files

Remote files must have

- An optional *name*, available through the `name` lens
- An optional *url*, from which the file can be downloaded, available through the `url` lens
- An optional *file type*, available through the `fileType` lens
- A *creation date*, available through the `creationDate` lens
- and a *size*, available through the `size` lens

All optional fields use lenses which return `Maybe` values.

Content of remote files can be downloaded using `readTextFile` or `readFileBytes`.

Local Files

Local files must have

- A *name*, available through the `name` lens
- *content*, which is either in-memory or on-disk, available through the `content` lens
- and an optional *file type*, available through the `fileType` lens

Local files can be created using the `newLocalFile` function and uploaded using `shareFile`.

Adapters which support the `HasFiles` class can emit the `FileSharedEvent`. This event can be handled with the `fileShared` and `fileSharedIn` triggers.

External scripts

Since marvin scripts are just Haskell values adding external scripts is as easy as importing a library.

Assuming you use cabal or stack for building your project you need to add the library to your `.cabal` file. Each library may define multiple scripts.

As with user scripts you need to wire the scripts into the main file.

If you manually create the main file, you add the script like a user script by importing the module and adding the scripts to the list of scripts.

If you use the automatic main file you can add external scripts by listing the modules to import in the `external-scripts.json` file. Currently the `external-scripts.json` only supports listing modules. This means each external script must be in its own module and be named `script`.

```
[
    "Marvin.Script.SomeScript",
    "Marvin.Script.AnotherScript",
    "SomeUserScript",
    "Some.Library.Script"
]
```

Note: The API around `external-scripts.json` is not stable and it will probably change in the future, although we might preserve backward compatibility.

You can join the discussion around its design on GitHub.

Logging in marvin

Marvin integrates with a logging library monad-logger.

This means all marvin monads are an instance of `MonadLogger`, meaning they already know how to log messages. You can use all the functions in `Control.Monad.Logger` to log messages in marvin and they will be automatically filtered and processed as the config specifies.

Basics of how to log messages

The monad-logger library exposes some nice functions for logging messages. For basic logging you should use functions such as `logWarnN` and `logErrorN`.

Marvin will automatically prepend some location information for you i.e. if you log a message in the `money` script it will show up in the log with `script.money`. This makes it easier to trace where a logging message came from.

```
{-# LANGUAGE OverloadedStrings #-}

script = defineScript "hello" $ do
    logDebugN "Starting definition of script"

    hear "hello .*" $ do
        logInfoN "Heard a hello"
        send "Hello to you too"
```

Since the logging functions all use strict `Text` as input it is recommended to use marvins strict text interpolator if your messages should contain external strings and data as the interpolator will take care of converting the data for you. The interpolator for strict text is called `isT` and exposed by default if you import `Marvin.Prelude`. For more information on how interpolation is used in general see *interpolation*.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE OverloadedStrings #-}
import Marvin.Interpolate.Text

script =
    ...

    hear "sudo .*" $ do
        match <- getMatch

        logInfoN $(isT "I'm asked to do #{match !! 1}")

        send "okay"
```

For more advanced logging monad-logger offers some template Haskell functions which also record the place in the source code where the message came from. The functions are called `logWarning` and `logError`. They require template Haskell to be enabled and must be invoked like so: `$logWarning "my str"`.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE OverloadedStrings #-}

script =
    ...
    $logDebug "my message"
```

And again it is recommende to use this in conjuction with the interpolator to easily include data in the message.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE OverloadedStrings #-}

script =
    ...

    hear "sudo .*" $ do
        match <- getMatch
```

```
        $logInfo $(isT "I'm asked to do #{match !! 1}")

        send "okay"
```

Filtering log messages

You can set a lower bound for the level of log messages which are reported.

The config key `bot.logging` controls which is the lowest level of log messages which are recorded. Available levels are (in ascending order, case insensitive) `debug`, `info`, `warning`, `error`. Currently the choice of level is final, meaning changes in the config will not take effect until the program is restarted. This is likely to change in the future.

Command line parameters can be used to overwrite the logging settings. Passing `-v` to marvin during startup sets the logging level to `info` regardles of the config parameters. Similarly passing `--debug` sets it to `debug`.

Choosing a logging target

Attention: Not implemented yet. Currently log messages will always be printed to stderr.

The marvin preprocessor (marvin-pp)

The basic usage of the marvin preprocessor is to generate the main file.

To use `marvin-pp` add a line like `{-# OPTIONS_GHC -F -pgmF marvin-pp -optF --adapter -optF slack-rtm #-}` to the top of your main file.

Note: If you use `marvin-pp` it will generate the entre file, any previous content of the file is **ignored completely**.

Explanation of the preprocessor invocation

`{-# OPTIONS_GHC -F -pgmF marvin-pp #-}` tells the Haskell compiler to use `marvin-pp` as a preprocessor.

You can pass additional arguments to the preprocessor by prepending `-optF ARGUMENT` to the option line.

Important: Each argument has to be prefixed with `-optF`, i.e. to pass `--adapter slack-rtm` to the preprocessor you have to add `-optF --adapter -optF slack-rtm` to the option line.

Arguments to `marvin-pp`

Output from `marvin-pp --help`:

Note: The order of options is irrelevant.

```
marvin-pp ~ the marvin preprocessor

Usage: marvin-pp [-a|--adapter ID] NAME PATH PATH [-s|--external-scripts PATH] [-c|--
↪config-location PATH]

Available options:
-h,--help                Show this help text
-a,--adapter ID          adapter to use
-s,--external-scripts PATH
                         config file of external scripts to
                         load (default: "external-scripts.json")
-c,--config-location PATH
                         config to use (default: "config.cfg")
```

Option	Usage
`-h,--help`	Only used for printing the help on the command line
`NAME PATH PATH`	Arguments. Passed by GHC. (irrelevant for user)
`-a,--adapter`	identifier for the adapter to use. If omitted will attempt to read from the config (`--config`) at the `bot.adapter` key.
`-s, --external-scripts`	Point to an alternative file containting :ref‘external scripts <external-scripts>‘
`-c, --config-location`	Point to an alternate config file. See the *Runtime configuration* section. (only used for looking up the adapter to use, see `--adapter`)

Adapters

Adapters are the backend of marvin. The exchangeable part that talks to the chat service itself.

Adapters are not yet exchangeable at runtime. The bot application is compiled against one adapter.

Adapter polymorphism

The capabilties required of an adapter is defined via the typeclass `IsAdapter` in `Marvin.Adapter`. Therefore you may define generic scripts which will work with any adapter using just the `IsAdapter` constraint in the script initializer type signature.

```
import Marvin

script :: IsAdapter a => ScriptInit a
script = defineScript "name" $ do ...
```

Or if you need capabilities specific to some adapter you can reference the adapter type directly.

```
import Marvin
import Marvin.Adapter.Slack.RTM

script :: ScriptInit (Slack RTM)
script = defineScript "name" $ do ...
```

Users can define their own adapters of course but are strongly encouraged to release generic adapters publicly or contribute them to marvin.

Shell

Quick info

Adapter id	`shell`
Module	`Marvin.Adapter.Shell`
Type	`ShellAdapter`

Supports **'files'**_

The simplest of all adapters, the shell adapter is used mostly for testing purposes.

The adapter id (for including it via the preprocessor) is "shell". To wire manually import `ShellAdapter` from `Marvin.Adapter.Shell`.

It is recommended to run a shell instance of marvin with stderr piped to a file so that it does not interfere with your interactions with marvin.

The shell adapter supports a persistent history by specifying `adapters.shell.history-file` in your config.

Configuration keys

Name	Type	Necessity	Description
history-file	String	optional	If set the history of entered commands will be persisted here

Slack

For both of the following adapters you'll have to create a new bot user for your slack team.

Also for both of the following adapters you'll have to invite your bot to any channel in which it should be active (in slack) using `/invite <botname>`.

Channel references (for instance for `enterIn` and `messageChannel`) for this adapter use the actual channel name *without* the `#`. For instance the channel `#random` is referenced only with the string `random`.

Supports **'files'**_

Real Time Messaging API

Quick info

Adapter id	`slack-rtm`
Module	`Marvin.Adapter.Slack.RTM`
Type	`SlackAdapter RTM`

The adapter for the slack real time messaging api is currently the best supported adapter.

It works by opening a websocket to the slack servers from which it recieves events in real time.

The adapter id is "slack-rtm". For manual wiring you'll need the `(SlackAdapter RTM)` data structure from `Marvin.Adapter.Slack.RTM`.

Configuration keys

Name	Type	Necessity	Description
token	String	required	Authentication token for the slack API

Events API

Quick info

Adapter id	`slack-events`
Module	`Marvin.Adapter.Slack.EventsAPI`
Type	`SlackAdapter EventsAPI`

This adapter creates a server, which listens for events from the slack Events API.

In addition to configuring marvin for this adapter you'll also have tell slack the url for this bots server when configuring the bot.

The adapter id is "slack-rtm". For manual wiring you'll need the `(SlackAdapter RTM)` data structure from `Marvin.Adapter.Slack.RTM`.

Configuration keys

Name	Type	Necessity	Description
token	String	required	Authentication token for the slack API
use-tls	Bool	optional	Whether to use TLS encryption, defaults to true
certfile	String	required if TLS is used	Path to the TLS certificate
keyfile	String	required if TLS is used	Path to the TLS key
port	Int	optional	Port on which to run the server

Important: This adapter is not very well tested yet, please report any issues you encounter here.

Telegram

Both of the following adapters require you to create and register a telegram bot

<<<<<<< Updated upstream .. admonition:: Caveats

> In telegram file size for remote files is optional. Since the marvin adapter class requires a `size` field this field will be negative if there was no size present on the file.
>
> Additionally since there is per default no `url` property the field will always be `Nothing` and setting it is a no-op.

Poll

Quick info

Adapter id	`telegram-poll`
Module	`Marvin.Adapter.Telegram.Poll`
Type	`TelegramAdapter Poll`

The telegram poll adapter sends long running http requests to the telegram servers to recieve events in near real time.

A unique trait of this adapter is the `polling-timeout` configuration key, which governs how long at maximum the polling requests may be kept open if no new event has arrived.

Configuration keys

Name	Type	Necessity	Description
token	String	required	Authentication token for the Telegram API
polling-timeout	Int	optional	Timeout for the polling requests (seconds) defaults to 120

Important: This adapter is not very well tested yet, please report any issues you encounter here.

Push

Quick info

Adapter id	`telegram-push`
Module	`Marvin.Adapter.Telegram.Push`
Type	`TelegramAdapter Push`

The telegram push adapter creates a server and registers a webhook with telegram to receive event updates.

Configuration keys

Name	Type	Necessity	Description
token	String	required	Authentication token for the slack API
url	String	required	URL on which this server runs
use-tls	Bool	optional	Whether to use TLS encryption, defaults to true
certfile	String	required if TLS is used	Path to the TLS certificate
keyfile	String	required if TLS is used	Path to the TLS key
port	Int	optional	Port on which to run the server

Important: This adapter is not very well tested yet, please report any issues you encounter here.

IRC

Quick info

Adapter id	`irc`
Module	`Marvin.Adapter.IRC`
Type	`IRCAdapter`

The irc adapter connects to your IRC server via the irc-conduit library.

Configuration keys

Name	Type	Necessity	Description
host	String	required	Url for the IRC server
port	Int	reqired	Port for the irc server

Command and message events

- All direct messages (`privmsg`) to the bot are interpreted as a command, and the contents of the message is passed to the handlers such as respond.
- Messages in a channel, which are directed at the bot are also interpreted as commands.
- All other messages are interpreted as message events.

Important: Caveats

Message encoding As I am not very familiar with the IRC API and its message encodings in the current adapter I assume that all messages are utf-8 encoded. Should this not be the case, please report it. Should you be someone who is familiar with the encodings of IRC messages, please contact me so that we can improve this adapter.

CTCP messages CTCP messages are not supported. All CTCP messages are currently ignored. However I welcome anyone who would like to try and implement some CTCP functionality.

Strings and String conversions (in Haskell)

Representation of Strings is a sore spot in Haskell, unfortunately.

The fundamental problem is that the 'default' String, the `String` type from the standard library, is a *linked list* of characters. Nicely enough it is unicode capeable and handles special characters nicely, however using linked lists as strings is very inefficient.

Therefore marvin uses a more efficient string type called `Text`. To be precise the `Text` type in `Data.Text.Lazy` from the text library.

Functions exposed by the marvin library generally ALL deal with this string type, to make it as easy as possible for the user.

However when you interact with other libraries you might encounter other string types, such as `ByteString` (often the result of HTTP requests or input for JSON decoding) and `String` from the standard library, often in the form of `FilePath` as name for files and directories.

Strict and lazy `Text` and `ByteString`

Furthermore both `Text` and `ByteString` have a **strict** and a **lazy** variant. It is not really necessary to know the difference between the lazy and strict variants of these strings, suffice to say they are not the same thing.

If you need to convert between the strict and lazy variants of these strings there is a `fromStrict` (converts strict to lazy) and `toStrict` (converts lazy to strict). This is the case for both `Text` and `ByteString`. The two conversion functions `toStrict` and `fromStrict` are always contained in the module holding the **lazy** version of the type. This means for `Text` it is `Data.Text.Lazy` and for `ByteString` it is `Data.ByteString.Lazy`.

If you want to know which String type you have, look at the module.

- strict `Text` comes from either `Data.Text` or `Data.Text.Internal`

- lazy `Text` comes from either `Data.Text.Lazy` or `Data.Text.Internal.Lazy`

`ByteString` uses the same naming scheme, just replace `Text` with `ByteString`.

Converting `String`

To convert a `String` value **to** the string type used in marvin use `pack` from the `Data.Text.Lazy` module. To convert a `String` value **from** the string type used in marvin use `unpack` from the `Data.Text.Lazy` module.

You can use the same functions for `FilePath` as it is only an alias for `String`.

Converting `ByteString`

`ByteString` values are a lot harder to convert than `String`, becuase they have no specified encoding.

To convert a `ByteString` value you, first make sure it is a lazy `ByteString`. If it is not convert it with `fromStrict`.

Now you'll have to decode the `ByteString`. The `text` library offers a number of decoding functions in the `Data.Text.Lazy.Encoding` module. The `encode*` functions are used to create `ByteString``s from text and the ``decode*` functions are used to turn `ByteString``s into ``Text`.

If you are not sure what encoding your source text is try `Utf8`, its the most common.

Lens quickstart

The `lens` library is a clever library which brings some useful parts of object oriented syntax to Haskell. Namely the ability to easily access and manipulate nested data structures and. Marvin depends on and uses, both internally and in its interfaces, a library called microlens. `microlens` is very similar to lens. It offers a smaller set of features (which suffice for marvin) but also has far fewer dependencies. It is however fully compatible with `lens`, meaning a `Lens` value from `microlens` can be used as a `Lens` value `lens` as both are simply type aliases. So if you want to use features from `lens` you don't need to also use `microlens` to be compatible with marvin.

The `Lens'` type can be used to manipulate a field in a data structure. For example a lens `foo :: Lens' Bar Int` pertains to a field of type `Int` in a data structure called `Bar`.

Getting

The operator `^.` is used to access the contents of a field. `x ^. foo` accesses the `foo` field in the `x` value.

These lenses are composable. If we have a lens `bar :: Lens' Baz Bar` and a value `y :: Baz` we can access the nested `foo` value with `y ^. bar . foo`.

Setting

The same lenses can also be used to modify the contents of the referenced field. `foo .~ value` creates a function which sets the `foo` field to `value`. Often this is combined with the revers application operator `&` to write code such as `x & foo .~ value` which sets `foo` in `x` to `value`. Using `&` we can also chain modifications like so `x & foo .~ value & anotherField .~ anotherValue`. This does not modify the original `x` but instead returns a new value of type `Bar` which is identical to `x` except for the contents of the `foo` field.

Another operator for modification is `%~` where `foo %~ f` modifies the content of the `foo` field with the function `f`.

Lenses in modification operations are also composable. For instance to set the nested `foo` field in `y` we can say `y & baz . foo .~ value`.

marvin-interpolate, A simple string interpolation library

Note: The marvin interpolation library, with no dependencies on marvin itself, is separately available on hackage.

The marvin string interpolation library is an attempt to make it easy for the user to write text with some generated data in it. The design is very similar to the string interpolation in Scala and CoffeeScript, in that the hard work happens at compile time (no parsing overhead at runtime) and any valid Haskell expression can be interpolated.

TLDR and `Marvin.Prelude` specifics

By default `Marvin.Prelude` exposes two interpolators `isL` for composing messages which can be sent to the chat (produces lazy `Text`) and `isT` for composing log messages (produces **strict** `Text`).

Both require Template Haskell and Overloaded Strings which is enabled by adding the lines `{-# LANGUAGE TemplateHaskell #-}` and `{-# LANGUAGE OverloadedStrings #-}` at the beginning of your script file.

Example:

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE OverloadedStrings #-}

myStr = let x = "data" in $(isL "some string with #{x}: #{ 1 + 1 }")
-- "some string with data: 2"
```

The syntax is `$(interpolator "interpolated string")` where interpolator is either `isL` or `isT`. As in CoffeeScript you can use `#{}` to interpolate an expression. Any valid Haskell expression can be interpolated, it can reference both local and global bindings.

The result of the expression must either be a type of string or be convertible to one via `Show` or `ShowL` or `ShowT` respectivley which is true for most basic data types. More information on conversion can be found *here*

How to interpolate

The library uses the builtin Haskell compiler extension in the form of *QuasiQuoters* (QuasiQuotes language extension) and /or *splices* (Template Haskell language extension)

Some examples to start with:

```
{-# LANGUAGE QuasiQuotes #-}

import Marvin.Interpolate

str1 = [iq|some string #{show $ map succ [1,2,3]} and data|]
-- "some string [2,3,4] and data"

str2 =
    let
        x = "multiple"
        y = "can"
```

```
        z = "local scope"
    in [iq|We #{y} interpolate #{x} bindings from #{z}|]
-- "We can interpolate multiple bindings from local scope"

str2 =
    let
        x = ["haskell", "expression"]
        y = " can be"
    in [iq|Any #{intercalate ' ' x ++ y} interpolated|]
-- "Any haskell expression can be interpolated"
```

Alternatively the interpolators are available as splices

```
{-# LANGUAGE TemplateHaskell #-}

import Marvin.Interpolate

str1 = $(is "some string #{show $ map succ [1,2,3]} and data")
-- "some string [2,3,4] and data"
```

It basically transforms the interpolated string, which is `[iq|interpolated string|]` or in splices `$(is "interpolated string")` into a concatenation of all string bits and the expressions in `#{}`. Therefore it is not limited to `String` alone, rather it produces a literal at compile time, which can either be interpreted as `String` or, using the Overloaded Strings extension, as `Text` or `ByteString` or any other string type.

Interpolators and conversion

`iq` (for *interpolate quoter*) and `is` (for *interpolate splice*) is the basic interpolator, which inserts the expressions verbatim. Hence when using `iq` or `is` all expressions must return the desired string type, otherwise the compiler will raise a type error.

There are specialized interpolators, which also perform automatic conversion of non-string types into the desired string type. As an example, from earlier, if we use a specialized interpolator we dont need the call to `show`.

```
str1 = [iq|some string #{show $ map succ [1,2,3]} and data|]
-- "some string [2,3,4] and data"

-- is the same as
str2 = [iqS|some string #{map succ [1,2,3]} and data|]

-- ('iqS' is the specialized interpolator for 'String')
```

These specialized interpolators each have an associated typeclass, which converts string types (`String`, `Text` and lazy `Text`) to the target type, but leaves the contents unchanged and calls *show* on all other types before converting. This last instance, which is based on the `Show` typeclass, can be overlapped by specifying a custom instance for your type, allowing the user to define the conversion.

The naming scheme of the interpolators in general is `i<splice|quoter><pecialization?>`. I. e. `isS` expands to *interpolate splice to String* and `iqL` to *interpolate quoter to Lazy Text*.

- `iqS` and `isS` in `Marvin.Interpolate.String` converts to `String` via the `ShowStr` typeclass
- `iqT` and `isT` in `Marvin.Interpolate.Text` converts to `Text` via the `ShowT` typeclass
- `iqL` and `isL` in `Marvin.Interpolate.Text.Lazy` converts to lazy `Text` via the `ShowLT` typeclass

To import all interpolators, import `Marvin.Interpolate.All`.

Syntax for the interpolated String

Interpolation uses the quasi quoter sytax, which starts with `[interpolator_name|` and ends with `|]`. Anything in between is interpreted by the library.

The format string in between uses the syntax `#{expression}`. Any valid Haskell expression can be used inside the braces. Anything outside the braces is interpreted as literal string. And all names which are in scope can be used, like so.

```
let x = 5 in [iqS|x equals #{x}|] -- > "x equals 5"
```

Escape sequences

```
str3 = [iq|Two escape sequences allow us to write literal ##{ and |#] inside␣
↪expressions"}|]
-- "Two escape sequence allow us to write literal #{, |] and } inside expressions"
```

There are two escape sequences to allow literal `#{` and `|]`

Input	Output
`#]`	`]`
`##`	`#`

As a result the sequence `##{` will show up as a literal `#{` in the output and `|#]` results in a literal `|]`.

Differences between QuasiQuotes and splices

When using QuasiQuotes (`[i|interpolated string|]`) any character between is interpreted as literal, including this such as tabs and newlines. No escaping like `\n`, `\t` or `\\` is required.

In splices the input is interpreted as a Haskell String, therefore no newlines are allowed for instance and escape sequences such as `\n`, `\t` and `\\` are necessary. Furthermore literal `"` must be escaped also, as `\"`.

Note: The library internal *Escape sequences* are identical in QuasiQuotes and splices

Differences to/Advantages over other libraries

There are a few advantages this libary has over other string formatting options.

1. The hard work happens at compile time

   Unlike libraries like text-format and the Text.Printf module parsing the format string, producing the string fragments and interleaving data and strings happens all at compile time. At runtime a single fusable string concatenation expression is produced.

   Furthermore all errors, like missing identifiers happen at compile time, not at runtime.

2. Type Polymorphism

   The created, interpolated string has no type. It can be interpreted as any string type, so long as there is an IsString instance and the expressions inside return the appropriate type.

   This is different format string libraries like text-format and the Text.Printf module which always produce strings of a particular type and interpolation libraries like interpolate and interpol which require instances of `Show`.

3. Simple API and full Haskell support

   The interpolated expressions are just plain Haskell expressions, no extra syntax, beyond the interpolation braces `#{}`. Also all Haskell expressions, including infix expressions, are fully supported.

   This is different from Interpolation which introduces additional syntax and does not fully support infix expressions.

API breaking changes

Since version 0.3

The slack adapter no longer automatically prepends a `"#"` to channel names. This means channel resolution functions such as `resolveChannel` now work on the channel name directly. Example: to resolve the channel `#random` use `resolveChannel "random"`.

This also affects channel referencing handlers and functions such as `enterIn` and `messageChannel`. Example: what was previously the handler `enterIn "#random"` is now `enterIn "random"`.

FAQ

I added a new script, why is the functionality not present?

If you are using the automatic main file

You have to force the main file to be recompiled after adding or removing a script. You can do this by running *stack clean* (if you use stack for building) or *cabal clean* (if you use cabal for building).

If that didn't fix:

- **For user scripts:** make sure the new script is in the script directory and the file does not start with `.` or `_` (those are ignored by `marvin-pp`). See *The marvin preprocessor (marvin-pp)*
- **For external scripts:** make sure the script is mentioned in the *external-scripts.json* file. See [external scripts](external-scripts).

If you are defining the main script manually

Make sure you imported the script module in your main file and added the script to the list of scripts for the call to `runMarvin`.

This will look something like this:

```
import qualified MyScript

main = runMarvin [MyScript.script]
```

CHAPTER 5

Indices and tables

- genindex
- search