TODO$_{PACKAGE_{NAME}}$

Release 0.1.2.3

Nov 02, 2018

Contents

This site contains an introduction and overview of the main features of the `TODO_PACKAGE_NAME` package. For full API documentation see the TODO_PACKAGE_NAME Haddocks.

CHAPTER 1

TODO_PACKAGE_NAME Introduction

The TODO_PACKAGE_NAME package provides TODO.

More description of package.

Once you know the basics, or if you're looking for a specific function, you can head over to the TODO_PACKAGE_NAME Haddocks to check out the full API documentation!

1.1 Provided Components

- *A*: support for A

Note: You'll need `PACKAGE >= 0.1.2` for a few of the examples. See *Version Requirements* for info on how to check which version you have and how to upgrade.

1.2 Related Packages

- TODO_RELATED_PACKAGE_1 - frobnicates the A's from this package

1.3 Looking for more resources?

If you've worked your way through the documentation here and you're looking for more examples or tutorials you should check out:

TODO:

- Links to other documentation
- Links to other tutorials

1.4 Installing and using the TODO_PACKAGE_NAME package

1.4.1 Version Requirements

For some of the examples you'll need `PACKAGE >= 0.1.2`.

You can check to see which version you have installed with:

TODO: Instructions for how to see what version you have.

1.4.2 Importing modules

All of the modules in `TODO_PACKAGE_NAME` should be imported `qualified` (TODO: maybe they shouldn't be) since they use names that conflict with the standard Prelude.

```
import qualified TodoPackage.A as A
```

1.4.3 In GHCi

Start the GHCi REPL with `ghci` or `stack ghci`. Once the REPL is loaded import the modules you want to use and you're good to go!

1.4.4 In a Cabal or Stack project

Add `TODO_PACKAGE_NAME` to the `build-depends:` stanza for your library, executable, or test-suite:

```
library
    build-depends:
        base >= 4.3 && < 5,
        TODO_PACKAGE_NAME >= 0.1.2 && < 0.2
```

and `import` any modules you need in your Haskell source files.

CHAPTER 2

A

TODO: A one paragraph description of what this module does/provides at a high level.

```
-- TODO: Data declarations of any key constructs you need to work with this
-- module.
data A x y = ...
```

TODO: Any important things to keep in mind when using stuff in this module. A few examples are included below.

Important: `A` relies on the type `x` having instances of the `Eq` and `Ord` typeclass for its internal representation. These are already defined for builtin types, and if you are using your own data type you can use the deriving mechanism.

All of these implementations are *immutable* which means that any update functions do not modify the A that you passed in, they creates a new A. In order to keep the changes you need to assign it to a new variable. For example:

```
let a1 = A.make "a" 1
let a2 = A.frobnicate a1
print a1
> A (Just "a") (Just 1)
print a2
> A (Just "a") (Just 200)
```

2.1 Short Example

TODO: 5-10 short examples of using `A`.

The following GHCi session shows some of the basic `A` functionality:

```
import qualified Data.A as A

let a1 = A.make "a" 1
```

(continues on next page)

(continued from previous page)

```
...
```

TODO: Tips that apply when using this module but aren't necessary. This could include useful language extensions. We've included an example taken from the `containers` docs.

Tip: You can use the OverloadedLists extension so you don't need to write `fromList [1, 2, 3]` everywhere; instead you can just write `[1, 2, 3]` and if the function is expecting a map it will be converted automatically! The code here will continue to use `fromList` for clarity though.

2.2 Importing A

TODO: How to import your module (example paragraph for qualified imports below)

When using `A` in a Haskell source file you should always use a `qualified` import because these modules export names that clash with the standard Prelude (you can import the type constructor on its own though!). You can always import any specific identifiers you want unqualified. Most of the time, that will include the type constructor (`A`).

```
import Data.A (A)
import qualified Data.A as A
```

2.3 Common API Functions

TODO: Any applicable tips/notes about the module. For example, noting difference between Strict/NonStrict versions of data structures.

2.3.1 Construction and Conversion

Create an empty A

```
A.empty :: A x y
A.empty = ...
```

empty creates an `A` with no data.

```
A.empty
> A Nothing Nothing
```

Create an A with data

```
A.make :: x -> y -> A x y
A.make x y = ...
```

make creates an `A` with a valid `x` and `y`.

```
A.make "a" 1
> A (Just "a") (Just 1)

A.make "containers" ["base"]
> A (Just "containers") (Just ["base"])
```

TODO: Other ways to construct an A

2.3.2 Querying

TODO: Lookup/read-only operations.

2.3.3 Modification

TODO: Modifying operations

2.4 Serialization

TODO: How to serialize an `A` (if that's a reasonable operation)

2.5 Performance

TODO: Link to more information on performance of the module.

2.6 Looking for more?

TODO: Links to follow-up reading.

Didn't find what you're looking for? This tutorial only covered the most common `A` functions, for a full list of functions see the `A` API documentation.