# Quickly Documentation

***Release 0.1***

**Michael Spencer**

March 28, 2016

Contents

Quickly is a build tool and QML module with provides an NodeJS-like ES6 environment for Javascript used in QML. The goal of the project is to allow you to write awesome modern ES6 Javascript taking advantage of classes, decorators, arrow functions, and best of all, many of the vast array of NPM packages available using the standard ES6 module imports. You can then take that code and use in directly from QML, just as you would with plain, old, QML-specific Javascript. You can even build a library using ES6 and NPM packages, and then distribute that as a standard QML module or QPM package for other developers to use in regular QML or QML-specific Javascript.

For those who would prefer to stick with standard QML-specific Javascript, you can also do that and still use the Quickly library, which gives you promises, the fetch API, and many polyfills. This is great for longtime QML developers or existing projects that just want to drop in some easy-to-use features from modern JS core libraries.

CHAPTER 1

# Offline Reading

Download the docs in pdf or epub formats for offline reading.

## 1.1 Tutorial

### 1.1.1 Installing

If you don't have `qmlify` installed yet, install it using `npm`:

```
$ npm install -g qmlify
```

### 1.1.2 Setting up Babel

Let's start by setting up the Babel configuration to transpile ES6 into standard JS. Create a file called *.babelrc* in your project's directory and put the following in it:

```
{
    "presets": ["es2015", "stage-0"],
    "plugins": [
        "transform-decorators-legacy"
    ]
}
```

Now, install the `babel` command by running:

```
$ npm install -g babel-cli
```

And install the Babel plugins and presets using:

```
$ npm install --save-dev babel-preset-es2015 babel-preset-stage-0 babel-plugin-transform-decorators-l
```

You're all set up! Time to write some code!

### 1.1.3 Creating your first ES6 module

Let's start off with something simple. Create a file in `src` and call it `app.js`:

```
export class Person {
    constructor(name) {
        this.name = name
    }

    hello() {
        console.log(`Hello, ${this.name}!`)
    }
}
```

Now create a QML file named `main.qml`:

```
import QtQuick 2.0
import "app.js" as App

Item {
    Component.onCompleted: {
        var person = new App.Person('Michael')
        person.hello() // Prints "Hello, Michael!"
    }
}
```

### 1.1.4 Using polyfills

Let's get some fake users from <http://jsonplaceholder.typicode.com/>. In plain JS, you'd have to use XMLHttpRequest. With Quickly, you can use the fetch API. Add the following to your `app.js` file:

```
export function getUsers() {
    return fetch('http://jsonplaceholder.typicode.com/users')
        .then(response => response.json())
}
```

And add the following code to your `main.qml` file, inside the `Component.onCompleted` block:

```
App.getUsers().then(function(users) {
    var name = users[0].name
    console.log('The first user is: ' + name)
})
```

Isn't that nicer and easier than XMLHttpRequest?

### 1.1.5 Using NodeJS core modules

Let's say we want to parse a URL and get the domain name. Easy, just add the following to your `app.js` file:

```
import * as url from 'url'

const data = url.parse('http://www.google.com')
console.log(data.host) // prints 'www.google.com'
```

### 1.1.6 Using an NPM package

Lodash is a popular library for doing all sorts of useful stuff. The chunk method happens to be the first method in the documentation, so let's try that:

import _ from 'lodash'

const chunks = _.chunk(['a', 'b', 'c', 'd'], 3) // Returns [['a', 'b', 'c'], ['d']]

## 1.2 QMLify Transpiler

### 1.2.1 Babel setup and basic usage

To use `qmlify`, you will need to the `babel` and `qmlify` CLIs installed globally using NPM:

```
$ npm install -g babel-cli qmlify
```

You will need to add a `.babelrc` file to tell `babel` (used by `qmlify`) which transformations to apply. Here is a sample `.babelrc` file with ES6 and some additional features enabled:

```
{
    "presets": ["es2015", "stage-0"],
    "plugins": [
        "transform-decorators-legacy"
    ]
}
```

Based on the this config file, you will need the following NPM packages saved locally as dev dependencies:

```
babel-preset-es2015
babel-preset-stage-0
babel-plugin-transform-decorators-legacy
```

Now just run *qmlify* on your src directory like this:

```
$ qmlify -d src
```

This will transpile all JS files and copy any other files to the `build` directory. Now, run or reference your main QML file from the `build` directory instead of the *src* directory.

### 1.2.2 Usage without Babel

Coming soon!

### 1.2.3 Integration with CMake

Coming soon!

## 1.3 Core JS Modules

Quickly provides a polyfills library and a set of the core NodeJS modules for use in your JS or QML code.

### 1.3.1 Polyfills

The following pollyfills are included (links are to documentation, usually on MDN):

- Array.from

- Array.prototype.find
- Array.prototype.findIndex
- Array.prototype.includes
- Number.isNaN
- Number.isFinite
- Object.assign
- String.prototype.endsWith
- String.prototype.startsWith
- String.prototype.includes
- WeakMap, Map, WeakSet, and Set
- Reflect
- Symbol
- Promise
- fetch, Request, Response, Headers

### 1.3.2 Core Modules

- assert (exposed as Assert to qml)
- url (exposed as Url to qml)
- querystring (exposed as Querystring)
- punycode (exposed as Punycode)
- path (incomplete, exposed as Paths, not Path, because of the QtQuick.Path)
- fs (incomplete, exposed as Filesystem)

### 1.3.3 Usage

Depending on whether you're working in ES6, plain JS, or QML, there are different ways to use the Quickly core modules. ES6 is definitely the easiest, so we doing as much work as you can from ES6 wherever you can.

#### Using from QMLified ES6

This is the easiest way to use the core modules. All the polyfills are available on the global scope, so use them just as you would with a built-in class in QML/JS. The polyfilled methods are added directly to Array, Object, etc:

```
const promise = new Promise((resolve, reject) => {
    promise.resolve('Hmm, why again did we need a promise here?')
})

promise.then((response) => {
    console.log(response)
})
```

To use the core NodeJS modules simply import them as you would with any ES6 module:

```
import * as url from 'url'

const data = url.parse('http://www.google.com/search?q=Quickly')
```

### Using from standard Javascript

Using from standard Javascript is a bit more complicated. You will need to import the Quickly QML module. The polyfills are on a Polyfills type, although the fetch API is also on the Http type for convenience:

```
.pragma library
.import Quickly 0.1 as Quickly

// This saves typing, but isn't necessary
var Polyfills = Quickly.Polyfills
var Promise = Quickly.Polyfills.Promise
var Http = Quickly.Http
var Url = Quickly.Url

var promise = new Promise(function(resolve, reject) {
    var set = new Polyfills.Set()
    resolve(set)
})

Http.fetch('http://www.google.com')
    .then(function(response) {
        return response.text()
    }).then(function(text) {
        console.log(text)
    })

var url = Url.parse('http://www.google.com')
```

### Using from QML

Like with standard JS, the polyfills and core modules are available via the Quickly QML module:

```
import QtQuick 2.4
import Quickly 0.1

Item {
    Component.onCompleted: {
        var promise = new Promise.Promise(function(resolve, reject) {
            var set = new Polyfills.Set()
            resolve(set)
        })

        Http.fetch('http://www.google.com')
            .then(function(response) {
                return response.text()
            }).then(function(text) {
                console.log(text)
            })

        var url = Url.parse('http://www.google.com')
    }
}
```

## 1.4 Distributing your Module

In QML, you might have a JS library file named `test.js` that looks like this:

```
.pragma library

function test() {
    console.log('test')
}
```

Along with the corresponding `qmldir` file:

```
module ExampleModule

Test 0.1 test.js
```

If you wanted to use the `test()` function in another JS file in your app or in another QML module, you'd do something like this:

```
.pragma library
.import ExampleModule 0.1 as Example

Example.Test.test()
```

In ES6, you would do something like this:

```
// test.js
export function test() {
    console.log('test')
}
```

And use it in your file like this:

```
import {test} from 'test'

test()
```

To bridge the gap between the QML way of doing things and the ES6 style, `qmlify` lets you "export" specific files from QML modules as ES6 modules. To implement the previous example in QMLified ES6, you'd do something like this:

```
// test.js
export function test() {
    console.log('test')
}

// qmldir
module ExampleModule

Test 0.1 test.js

// quickly.json
{
    "exports": {
        "test": "ExampleModule/Test"
    }
}
```

And install the `quickly.json` file (from the build folder, NOT the original one in your source folder) along side the `qmldir` and QML/JS files. Now in your app you'd do the following:

```
// app.js
import {test} from 'test'

test()

// quickly.json
{
    "dependencies": {
        "test": "0.1"
    }
}
```

If a default version is available and you want to use that, you can leave the dependency out of the `quickly.json` file.

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search