f_

Asynchronous programming made easy

Sam Machiels
November 5 2015

Table of Contents

Preface

Programming asynchronous application logic can be hard, especially when you want to maintain a clean, testable and loosely coupled modular code base. Improper asynchronous programming often results problems like hard to test code, a lack of modularity and tightly coupled application logic. In order to tackle these problems, "f_" was created, f_ is targeting the the Node.js platform, which runs on the V8 JavaScript engine.

This project was not designed the eliminate all the problems programmers face. It's there to provide a guideline whilst writing software that has to run asynchronously and or on multiple cores.

Since f_ is set up to be targeted by other tools, it's easy to write tools which utilize the f_ API. Because of this f_ can be used as an underlying framework to ease asynchronous and multiple core utilizing code development whilst not using it for application level structure.

Simplicity, flexibility and low impact are kept in mind at all times while developing f_.

Introduction

Over the years it has become easier to give instructions to a computer by writing software, because of abstraction. Nowadays anyone can open a browser, type some JavaScript in the developer console and for example test and use a calculator program. This abstraction results in the rise of new problems, a shift of what programmers have to focus on. This together with the increase in CPU cores per devices which requires programmers to balance CPU loads across different cores and threads instead of having to write software that runs on a single thread, is a major shift of focus.

The result of programmers getting to focus more on actually translating an idea to a computer language instead of focusing on how to do the translation, results in growth of usable computational tasks for mankind to use. A way to contribute to this growth of usable computational tasks, is to tackle problems which arose (or became more significant) because of new technological advancements. Some examples are: software that has to utilize multiple CPU cores and non blocking application development.

And then there is the Internet, which also resulted in huge changes in what programmers have to focus on. Computers have to perform different actions in a different manner, sometimes computers aren't even optimized to perform such actions. Let alone giving the computers instructions on how to perform these actions. That's where software comes in, building layers of abstraction so programmers get to focus on actually realizing ideas (again).

Asynchronous programming

When an application interacts with the real world. Take for example an HTTP request handler. It's of course unwanted behavior that the whole application freezes whilst handling an HTTP request. All other HTTP requests would be blocked, this is especially bad when big computational tasks have to be completed whilst handling HTTP requests It can result in a not responding HTTP request handler (server) for hundreds of milliseconds. In order to allow more HTTP requests to be handled at simultaneously, they have to be handled asynchronous. Once an HTTP requests comes in, it will be put to the background so another piece of software can do the complex computational task, when it's completed, a function is called (*callback function*). All the complex computations will be executed in the background, and not by the main application process. Using this strategy, a heavy HTTP request will not block all other HTTP requests from coming in.

When taking the example described above, utilizing multiple CPU cores becomes a breeze. If one makes sure the complex HTTP request handler code is utilizing multiple cores, you don't have to worry about utilizing multiple cores in higher level application code. This means concerns are separated, more on this in the chapter: Separation of concerns.

Separation of concerns

In order to keep software source code organized and maintainable it is important to make sure every

piece of code is doing just what it needs to do, and nothing more. This results in code that is easy to read and understand. Take for example a username validator, which could be written in a single function called *isValidUsername*. This function will perform operations like checking for forbidden characters and a check for the length of the username. If the function would be written in 20 lines of code a lot of the operations like regular expression matching would be hard to understand at a glance. If those operations would be abstracted away in new functions like *hasForbiddenCharacters* and *isRequiredLength. In* for example a *username_validator* module, it would become easier to look at the *username_validator* module source code and quickly grasp what is going on. Separating concerns can bring problems of its own, this will be discussed in the chapter: Loosely coupled application logic.

Loosely coupled application logic

When a piece of code changes within software, it's important to know what effects these changes will cause. When working on software with large code bases, this tracking of effects can become hard. If 20 pieces of some piece software depend on the result of one piece of code, and that one piece is going to produce different results, the other 20 pieces have to be changed manually so they can work with the new results received when executing the changed piece of code. This is called tightly coupled application logic.

In order to overcome this problem, the application logic has to be coupled loosely. This can be done by implementing APIs. The 20 pieces of code that are dependent on the result of the changed piece of code will make calls to an API instead of calling the code functionality manually. This means that as long as the API won't change, the 20 pieces of code won't have to change. Whilst it is now possible to change the code behind the API. Changes to the code behind the API will only require the API to be updated.

Problem to solve

Programmers may have a difficult time writing asynchronous and multiple core utilizing code within larger code bases, where separation of concerns is important in order to keep the code base structured, testable and maintainable.

When writing code, programmers shouldn't have to spend more time writing asynchronous and multiple core utilizing code than they would be when writing synchronous and single threaded code. This in order to focus more on the goal instead of the road towards the goal.

Solution to the problem

A simple, lightweight and flexible library which makes it easy to write small pieces of asynchronous and multiple core utilizing code within projects that require a structured, testable and maintainable code base.

API

getConstructor

The getConstructor function is used to get an f_ task list. All configuration before execution is done by passing it an *options* object. The *options* object can have certain properties which configure the f_ task list, they are described here.

initializer

The *initializer* property is an optional way to provide a class initializer function. This function will be passed the arguments passed upon initialization of an instance from the class returned by *getConstructor* function.

function_flow

The tasks which are supposed to run can be defined by passing the *getConstructor* function a *function_flow* property in the *options* object when calling it. The elements in the array are objects with 2 required properties: *name* and *function.* The *name* property is used for feedback purposes and the *function* property is the actual task list item (function) to execute.

custom_prototype

This property is used to set custom prototype properties to the returned *f_Constructor.*

f_Constructor

The return value when calling *getConstructor* is an *f_Constructor* function. The *f_Constructor* function is essentially a constructor which constructs f_ task list instances. The *f_Constructor* function has the modules described below in the instance_modules chapter bound to its prototype. As well as the methods described in the chapter methods.

instance_modules

A few modules and their required data will be applied to instances of f_ task classes. These are described below.

err

The *err* module is function that holds all errors in a property called *data* which is an array. These errors are runtime level errors thrown by the engine, but also errors thrown by the f_ task list items. Errors are objects containing an error object and an property called *time* which holds the time the error occurred in Epoch format. Upon calling *err*, the *history* method is called and is passed the *err* object. The *err* objects are stored in the *data* array in the order of occurrence .

log

The *log* module is a function with a property called *data*. The *data* property is an array containing log objects. The log objects has two properties: *time* and *message.* Where *time* is the time the *log* function is called in Epoch format. And *message* is the argument passed to the *log* function when calling it. The *message* property can be of any data type. Just like with calling *err,* calling *log*, calls the *history* function and is passed the *log* object.

History

The history module is a function containing all *log* and *err* objects. Just like *log* and *err* data is stored in the order of occurrence. It can be useful to have a log of everything that happened whilst running a task list, and not just 2 separate logs for errors and notifications.

methods

All f_ its functionality for f_ task lists is contained in the methods described below. These methods are bound to the top level scope of the f_ task list instance, they are prefixed with *f_*. So for example if you want to continue to the next f_ task in the f_ task list, *f_next* is called. To all of these methods besides *go*, event handlers can be bound. This is done by using the Node.js events functionality. All retry events send the optional *err_object* as the first argument the bound event handler receives.

next

When *next* is called, the next function in the f_ task list its *function_flow* array will be called. Which function is called depends on the position of the *function_flow* iterator. When there is no function to be called due to the completion of the f_ task list, the *finish* method is called. The arguments passed when calling *next* are passed to the next function in the *function_flow* array. This functions emits the *next* event.

go

The *go* method is an alias for the *next* method, it calls the *next* method. This method exists because of the reason it sometimes makes the code needed for the initialization of an f_ task list class more expressive. This function causes the *next* method to be called, which then emits the *next* event.

abort

This method is can be called within an f_ task to abort the f_ task list. Calling *abort* results in the f_ task list to be stopped, no new functions from the *function_flow* array are called. This method takes an error as an argument. When the *abort* function is called, the *abort* event is fired. A property will be added to the task list: *f_aborted_at*, which contains an Epoch time stamp.

finish

Calling this method will finish a f_ task list, which means no more functions from the *function_flow* are getting called. A property will be added to the task list: *f_completed_at*, which contains an Epoch time stamp. This method emits the *finish* event.

retryAll

The *retryAll* method starts the current f_ task list from the start, meaning the first function in the *function_flow* array. This method takes an error as an argument. This method emits the *retryAll* event.

retryThis

The *retryThis* method re-runs the current f_ task. An alias could be *retryFromHere*. The f_ task list will be continued like normal when the current f_ task calls the *next* method. The *retryThis* method will increment the task it was called from its try counter. This method takes an error as an argument. This method emits the *retryThis* event.

retryFrom

When this method is called and passed an f_ task name from the f_ task list, the f_ task list will be restarted from the f_ task matching the name passed. This method can be useful if a certain part of the f_ task list has successfully been executed and it's not necessary to retry those f_ tasks. This method takes an error as an argument. This method emits the *retryFrom* event.

Examples

These examples are for demonstration purposes only. They contain minimal functionality in order to showcase how f_ works, with both synchronous and asynchronous code.

Minimal

This example shows the minimal functionality of f_. By creating an f_ *Tasks* class, which has functions in the passed *function_flow* array property. These functions get called when the *next* or *go* methods of the *tasks* instance is called. The *task* list is started by calling the *go* method, after the *tasks* instance is drawn from the *Tasks* class.

```
const Tasks = f_.getConstructor({
  function_flow: [
    {
      name: 'method1',
      function: function method1() {
        this.f_next();
      }
    },
    {
      name: 'method2',
      function: function method2() {
        this.f_next();
      }
    }
  ]
});

let tasks = Tasks();

tasks.on('finish', () => console.log('Finish'));

tasks.f_go;
```

Source code collector

This example demonstrates the use of f_ its asynchronous capabilities. Once a new tasks instance is drawn from the Tasks class, it sends an HTTPS request to the URL passed in the *custom_data* property. If no errors occur, the next method is called, *saveSource*. Which then writes the source code *d.body* to a file which has the passed *file_name* as its file name. Events are handled by the event handlers bound to the *error, abort,* and *finish* events.

```
const Tasks = f_.getConstructor({
  initializer: function initializer(o) {
    this.o = o.url;
    this.d.body = '';
  },
  function_flow: [
    {
      name: 'getSource',
      function: function getSource() {
        https.get(this.o.url, (res) => {
          res.on('data', (chunk) => this.d,body += chunk);
          res.on('end', () => this.f_next());
          res.on('error', (err) => this.f_retryThis(err));
        });
      }
    },
    {
      name: 'saveSource',
      function: function saveSource() {
        fs.writeFile(this.o.file_name, this.d.body, (err) => {
          if (err) {
            return this.f_retryThis(err);
          }
          this.f_next();
        });
      }
    }
  ]
});

let tasks = new Tasks({
  custom_data: {
    url: 'https://github.com',
    file_name: 'github'
  }
});

tasks.on('error', (err) => console.log('Error', err));
tasks.on('abort', (err) => console.log('Abort', err));
tasks.on('finish' () => console.log('Finish'));

tasks.f_go();
```

HTTP request handler

In this example f_ instances are used as HTTP request handlers. The initializer takes an options object *o*. The options object *o* contains the HTTP request its request and response object as the key value pairs *req* and *res*. After the initializer is done, the *go* method is called. Calling the first function in the *function_flow* array, *extractInfo*. This functions stores the request its URL and method in the data namespace *d* and calls the *next, w*hich then calls the *logInfo* function. The *logInfo* function simply logs the request its URL and method to the console and calls the *next* method, which calls the final function in the *function_flow* called *writeRes*. The *writeRes* function writes the request its method and URL in a concatenated string as the HTTP request its response, calling the *next* method for the last time, finishing the f_ task list.

```
const HttpReqHandler = f_.getConstructor({
  initializer: function initializer(o) {
    this.o = o;
  },
  function_flow: [
    {
      name: 'extractInfo',
      function: function extractInfo() {
        this.d.req_url = this.o.req.url;
        this.d.req_method = this.o.req.method;
        this.f_next();
      }
    },
    {
      name: 'logInfo',
      function: function logInfo() {
        console.log(this.d.req_method, this.d.req_url);
        this.f_next();
      }
    },
    {
      name: 'writeRes',
      function: function writeRes() {
        this.o.res.write(this.d.req_method + ' ' + this.d.req_url);
        this.o.res.end();
        this.f_next();
      }
    }
  ]
});

// Create an HTTP server, pass a function that initializes a new
// HttpReqHandler every time an HTTP request is made to server.
// The HTTP request and response objects are passed to the HttpReqHandler
// instance as key value pairs in an object. Finally, start listening for
// incoming HTTP request on port 3333.
http.createServer((req, res) => (new HttpReqHandler({ req, res })).f_go())
  .listen(3333);
```

Application development

In order to build and maintain this project a set of development tools are used, like code linters and hosted code coverage services. These tools will be discussed in this chapter.

Continuous integration

Continuous integration stands for the continuous supply information about an application its development status. Using continuously integrated tools whilst developing an application offers a lot of help to programmers, they can automate a lot of tedious yet necessary tasks and processes. Some examples are: building the application on multiple platforms and running test suites.

Integration tests

In order to ship software which is supposed to run everywhere, a lot more variables have to be taken in account when a final verdict has to be made about whether the software is ready to ship or not. Some examples are: multiple operating systems, multiple CPU architectures, differing GPU clock speeds, etc. When testing software, it's important to take things like these in account. This can be a time consuming process, proper integration tests can automate these processes. Of course, a client won't pay if the software runs on your computer and not on theirs. That's where integration tests come in, testing whether everything integrates nicely within (common) (computer) setups.

Travis-ci

Travis-ci is a (free for open source projects) hosted continuous integration build service. Travis-ci builds application according to which language/environment is used. If I'd for example wanted Travis-ci to build a C application for me, I'd instruct Travis-ci run the *make* file. Every time an update is made to the remotely hosted repository, Travis-ci will download the latest source code and tries to build the application. The build status can be viewed here.

Unit tests

Testing application functionality is important. How does a programmer know a piece of software is working as expected without testing it? The answer is simple: they don't. Every piece of software can be tested manually, every time an update is done to the source code. This of course requires a lot of time and effort, which results in programmers having less time to focus on the actual translation of an idea to a computer language. When for example a function its functionality won't be changed in the near future, a test suite can be written for it. This test suite will execute specific pieces of code and checks whether the results are as expected. When a result differs from what was expected, the programmer will be notified.

Mocha

Mocha is an open source JavaScript testing framework which works together nicely with other tools like code linters and code-coverage information collectors. It offers a simple API with which test suites can be described. When written correctly, test code looks a lot like English. Take for example some tests which would be written for a custom math library/module called *mather*.

```
describe('mather (custom math library)', () => {
  describe('.add', () => {
    it('adds argument1 and argument2 together', () => {
      assert.equal(mather.add(3, 5), 8);
    });
  });
  describe('.mul', () => {
    it('multiplies argument1 with argument2', () => {
      assert.equal(mather.mul(3, 5), 15);
    });
  });
});
```

If I'd for example made some changes to *mather* its *mul* method which would result in a different return value when 3 and 5 are passed as arguments, it'd be clear that the code is broken, by running the tests. Since the assertion, calling *mather.mul* with the arguments 3 and 5 should equal 15, was specified in the test function its body.

Code coverage

Code coverage tools checks which code is executed when unit tests are run. These tools check for dead code (unused code). Having statements, branches or functions in code which are not covered by tests could result in unwanted and sometimes really difficult to trace side effects. So it's important to always cover high percentages of an application its source code, this could percent a lot of trouble later on. The nice thing about a lot of code coverage tools is that they require little to no effort to be implemented within an already existing work flow.

Istanbul

Istanbul is a code coverage tool for JavaScript which tracks coverage of functions, statements and branches. A lot of other (JavaScript) code coverage tools just track statements and/or branches. This is not enough when source code like JavaScript has to be evaluated, since JavaScript treats functions as first class citizens. Istanbul also provides a browser based interface in which it's easy to view code coverage information for every file in the software its source code.

Coveralls

Coveralls is a (free for open source projects) hosted code coverage tracking service. It shows metrics about the history of software its code coverage. Coveralls works together nicely with Mocha and Istanbul, once all Mocha test suites are completed and code coverage information has

been collected by Istanbul, an HTTP request is made to the remote Coveralls API which will then process the code coverage information. The generated code coverage report can be viewed on Coveralls here.

In-line documentation

In-line documentation makes sure that code is easy to understand for everyone (including the writer) by looking at the documentation present in the source code without trying to understand the code manually. Something very complex can sometimes easily be described with a few sentences in a spoken language. The generated documentation can be viewed here.

JSDoc

JSDoc generates documentation by parsing comments written in a specific style.

If you take the simple *ensureObject* function below, it requires at least some experience with JavaScript to understand what is going on.

```
function ensureObject(x) {
  return typeof x === 'object' && !(x instanceof Array) ? x : {};
}
```

But when a code block containing a lot of information about the function is present, almost anyone can understand what is going on. The example below shows a comment that makes the *ensureObject* function easy to understand for anyone with some programming experience. Commenting code like this also makes sure the code is easy to understand after a period of time.

```
/**
 * Returns x when x is of type object else a new object. This function
 * also checks whether x is an instance of the Array class, this is
 * required because every array in JS inherits from the Object class.
 * @param {any} x - Gets checked for type object
 * @return {object} - x if x is an object else a new object
 * @example
 * ensureObject([]) // {}
 * ensureObject(34) // {}
 * ensureObject({ a: true }) // { a: true }
 */
function ensureObject(x) {
  return typeof x === 'object' && !(x instanceof Array) ? x : {};
}
```

Inch-ci

Inch-ci is a tool which checks an application its source code for in-line documentation. Inch-ci notifies programmers about the presence or absence of proper in-line documentation and gives tips about improving in-line documentation. The Inch-ci page for the f_ project can be found here.

Code quality

When developing software, it is important to keep the source code of high quality. This includes a lot of things, some examples are: a consistent code style, proper documentation, a high percentage of code coverage and no code code duplication.

Codacy and Code Climate

Codacy and Code Climate are remotely hosted and free for open source, services that will inform programmers about code quality metrics. Both services offer a detailed interface in which every issue its location plus information about the issue is shown. If possible there is also information about possible ways to fix the issue. The Codacy page for the f_ project can be found here, and the Code Climate page can be found here.

JSLint

JSLint is a tool used to check code styles. It is used to enforce a certain style of programming JavaScript code. This makes sure an issue like an unclosed statement will be noticed immediately. It is also important to use a single code style when working on a project with multiple programmers, this way its easy for members of the team to share code.

Version control

Git and GitHub

Git is used as a version control tool to manage a local f_ repository. GitHub is used to remotely host the local git repository. The repository can be found here. GitHub sends HTTP requests to 3rd party service hosts when for example two branches merge. These HTTP requests could trigger something like an evaluation of the updated repository its source.

Application realization

Pre-alpha

Whilst developing software with lots of asynchronous code, it was sometimes hard to keep a clear view on everything that was going on between all the different asynchronous code that was written. A lot of logic was hidden in Christmas tree code. Separation of concerns was always kept in mind, but it was hard to do this on a big scale with all the nested callbacks and whatnot . A lot of libraries were used which ease the writing of asynchronous code. But they lacked something, asynchronous application structure development. They offered me a way to write small pieces of asynchronous code without any structure.

The f_ project started whilst developing a YouTube downloader which featured a way to extract and reverse engineer logic about where Google stores its YouTube video and audio binary data. The YouTube download required a lot of asynchronous programming in order to allow a machine dependent finite amount of downloads to run alongside each other. It was working nicely, but maintenance was a pain because of a lack of structure. So a way to manage development of this YouTube downloader had to be thought of. It had to be simple, flexible and lightweight.

Alpha

All tools which were to be used whilst developing f_ were chosen and integrated within the work flow. And after some experiments with some basic prototypes of f_, a basic API was developed which would not change to much over time, whilst the underlying logic would be actively developed. As soon as the basic API and logic were finished and decently optimized the alpha version of f_ was released.

Beta

The alpha version was used for multiple projects by multiple programmers. Feedback was collected and optimizations were made and bugs were fixed. A few API changes were discussed and if it seemed reasonable, the API changes were applied. After active development f_ was ready for a beta release. All APIs were frozen until a bug fix required unfreezing them.

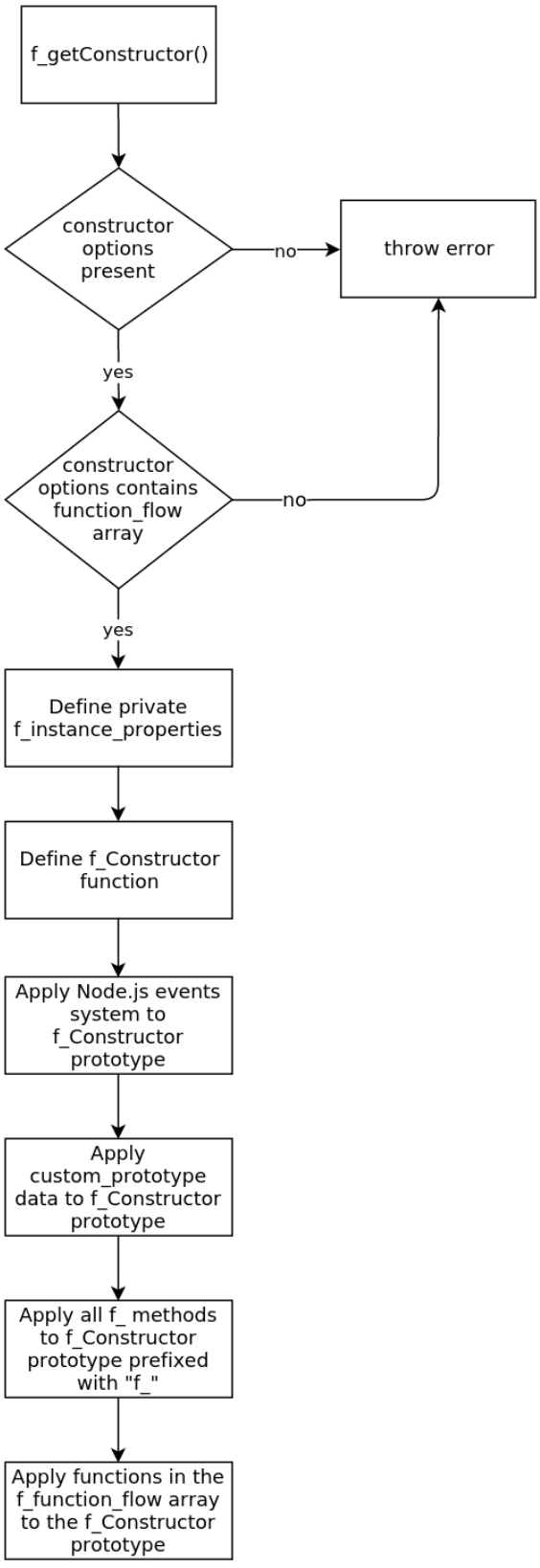
Production release

After the use of the f_ beta release, a lot of optimizations were made. None of those changes affected the API, since every API is frozen until a bug fix is required. A lot of optimizations were made concerning execution speed, since it is possible thousands of f_ instances are running simultaneously. With this in mind the code resulted to be a little less abstract, in a sense that some logic which could be abstracted away with a new function now isn't.

Flowcharts

getConstructor

This simple flowchart demonstrates what happens when *f_.getConstructor* is called.



@EXTRA FLOWCHARTS

Project maintenance

This project its API won't change. It could be possible that new features are added, but 100% backward compatibility will always be maintained. Bug fixes will of course happen, though they won't change the functionality nor will they change f_ its API, they can be submitted using the GitHub issue tracker.

It is expected the f_ project won't require much changes in order to continue working over the years since the code base has been reduced to a minimum in order to run lightweight.