B.Comp. Dissertation

Quiver.js: Server-Side Component Architecture

By

Soares (Ruo Fei) Chen

Department of Computer Science

School of Computing

National University of Singapore

2013/2014

B.Comp. Dissertation

Quiver.js: Server-Side Component Architecture

By

Soares (Ruo Fei) Chen

Department of Computer Science

School of Computing

National University of Singapore

2013/2014

Project No: H116400

Advisor: Prof. Ben Leong

Deliverables:

Report: 1 Volume

Abstract

The majority of web applications today are written as monolithic software with little reusability. By adopting concepts from Unix Philosophy and the REST architecture with Node.js, we present a new software architecture called Quiver. Quiver is a component-based architecture that allow web applications to be written as many small and reusable components. The Quiver architecture is made of multiple layers of abstractions consist of higher order functions that extend constructs in previous layers. One of the basic type of quiver component is the stream handler, which resemble closely to the Unix process as a JavaScript function that accepts an input stream and asynchronously return a result stream. The architecture also include a component system which eliminate boilerplates and allow components to be defined using a JSON-like DSL. Quiver gathers all the best practices recommended in today's web development into one place and present them in a consistent way. The goal of Quiver is to enable a new generation of web applications that are highly modular and scalable.

Subject Descriptors:

Computer systems organization~Pipeline computing
Software and its engineering~Middleware
Software and its engineering~Layered systems
Software and its engineering~Organizing principles for web applications
Software and its engineering~Abstraction, modeling and modularity

Keywords:
Software architecture, component system, functional programming, stream processing, modular software

Implementation Software and Hardware:
JavaScript, Node.js, V8, Ubuntu 13.10

Acknowledgement

I would like to express my gratitude to my project supervisor, Prof Ben Leong, for his supervision and guidance in writing this thesis. Prof Ben has given me a lot of freedom to explore on my own in developing Quiver. He have also given me many critical feedback on presenting my ideas and seeing things from a different perspective.

I'd also like to thank my company DoReMIR Music Research AB for giving me chance to make use of Quiver for the backend development of the company's product ScoreCloud. I would not be able to iterate on Quiver so fast without first-hand feedback of running it on the production server of ScoreCloud.

Lastly I would like to extend my appreciation and thanks to my family and everyone who have supported my journey to this day. Special thanks to my mentor Prof Khoo Siau Cheng, for listening to me and helping me out when facing difficulties in my study. Also thank you Prof Teo Chee Leong and the NUS Overseas College for giving me the life-changing opportunity to work and study entrepreneurship in Stockholm.

List of Figures

Contents

Chapter 1

Introduction

Quiver.js is a server-side component system that allow web applications to be built consist of small and reusable components. It consist of a collection of small libraries written in JavaScript running on Node.js.[1]

Web applications using Quiver are easily reconfigurable by simply rewiring components in different ways with minimal coding required. Components in Quiver are written as pure JavaScript functions that are combined using functional composition techniques.

Quiver is heavily inspired by the Unix Philosophy and Unix Processes. One main component type in Quiver is the stream handler, which closely resembles Unix processes to accept an input stream and return either an error or a result stream. Quiver also have a few other component types such as filter and middleware, which are designed to solve various problems in web development such as separation of concerns and dependency management.

Quiver also make use of many unconventional programming techniques that might look alien to mainstream JavaScript programmers. For example, Quiver make use of higher order functions over object oriented programming, at the same time show that such techniques can lead to cleaner code.

Quiver is not a web framework as it does not provide opinionated subsystems such as user management and data persistence. Instead a generic architecture is designed at lower level such that components can be written independent of application-specific designs.

1.1 Motivation

Quiver was created because web development today is dominated with the approach of writing web applications using monolithic web frameworks. Web frameworks work great for new projects, because they have specialized ways to solve various problems out of the box. But this makes it very hard to use a different approach that work against the

[1] Node.js Website: http://nodejs.org/

framework. Other than that libraries written for different frameworks are incompatible with each others. This lack of reusability force many code to be rewritten every time a new framework emerge.

I take modularity and code reusability as the most important consideration in building Quiver. My approach is to avoid giving opinion on how web applications should be written. Instead I learn from the best practices of writing highly concurrent software, and provide a general sets of guidelines for building generic components.

1.2 Background

Quiver was started as a concept I came about in 2011 to make way to create web applications in multiple programming languages. The project was first called Hypershell, given that the inspiration was to bring the power of Unix shell to the hypermedia web. The original plan for Hypershell was to create similar library in C and create wrappers to allow different programming languages to develop web applications all in the same way.

The Hypershell project was never made into a full project, due to the sheer complexities required to implement a high-level concept in C and having to support all high level languages. However I made use of many of the concepts I developed while working at a music tech startup in Stockholm, DoReMIR. At DoReMIR my job was to develop a Node.js server application that perform voice and music analysis on recordings made by the company's iPhone app called ScoreCloud.

As I developed the server, patterns emerged and eventually I gathered all the patterns and created Quiver. The server code using Quiver is currently run in production servers at DoReMIR, serving thousands of unique visitors per day to deliver the functionalities of ScoreCloud.

1.3 Contributions

Quiver is available under the open source MIT license and the source code is hosted on GitHub.[2] It is currently in beta release and is still under active development. Quiver is production ready and is currently more suited for experienced developers to write their own quiver components.

In this thesis, we present Quiver as a new architecture that is radically different from the monolithic architecture of conventional web applications. With Quiver we hope to move the web development paradigm forward and create a new generation of highly scalable web applications consist of modular components.

[2]Quiver project page at GitHub: https://github.com/quiverjs

Chapter 2

Related Works

2.1 Unix Philosophy

The Unix Philosophy is the primary inspiration for Quiver to bring such concept into web development. In the Unix command line, complicated programs can be constructed by combining smaller programs written in any programming language. Such level of modularity is rarely seen in modern applications, as such there are potential to revive this approach to make modern web applications as modular as Unix processes.

There are many guidelines on Unix Philosophy that Eric Raymond summarized in his book *The Art of Unix Programming*. (Raymond 2004) Of all points he made, I highlight the following points that are particularly useful to understand the design principles behind Quiver:

"Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features."

The conventional way of developing command line applications is to write simple programs that do one thing well. When new features are needed, new programs are written instead of extending existing programs. The small programs are then combined together using shell scripts to create more complex programs.

In Quiver the same principle is applied when writing quiver components. A quiver application consist of many *handlers* and *filters* that focus on doing one thing well. For every new functionalities, a new handler is created instead of modifying existing handlers. When there are cross-cutting concerns that a handler need to do before of after processing, a filter is created to handle the concerns instead of modifying many affected handlers.

"Expect the output of every program to become the input to another, as yet unknown, program."

All Unix processes can accept arbitrary number of command line arguments plus three standard stream interfaces: STDIN, STDOUT, and STDERR. This allows different Unix processes to communicate with each others by sending binary data through the standard

streams. In a Unix pipeline each Unix process have no control of how their input/output streams are connected together, therefore the pipeline can be constructed with arbitrary pairs of programs to form complex applications.

In Quiver each *stream handlers* also accepts a standard input stream and produce a result stream. The result stream from one handler can be pipelined as the input stream of another handler. This allows complex quiver applications to be made similar to shell scripts by simply arranging the pipeline of different stream handlers.

2.1.1 Common Gateway Interface (CGI)

In the early days of the web, CGI was a popular method of implementing dynamic web applications. Having the full power of Unix shell exposed to web applications, CGI is the closest ideal that Quiver attempt to revive without taking along its weaknesses.

One primary reason that CGI was eventually replaced by newer methods was the overhead of creating new process to handle each new HTTP request. The overhead stacks up further if the CGI script consist of shell scripts that call small Unix programs to perform trivial tasks. As a result it was never practical to apply the Unix Philosophy to CGI scripts due to the computational overhead of creating new processes by the operating system.

2.1.2 Global Scope

Unix commands also have the problem of referring to other commands through their global location in the file system. As a result new commands are usually installed to a common directory that is visible to all other commands. While the environment path variable may help in localizing the context of executing new command, the semantics of the execution is fragile and may change along with the filesystem, causing unexpected bugs or security issues.

Chroot is a heavyweight solution to make processes run inside an isolated environment. In an ideal practice different Unix command should be chrooted into separated environments, but that would have too much performance overhead to be practically considered.

In Quiver each component have implicit scope and may only access specific components that they have been wired with. Therefore it is much easier to understand the component implementation and users may manipulate the local environments of each component without affecting the rest of the component graph.

2.2 Representational State Transfer (REST)

While the Unix Philosophy provide guidelines on building small and reusable components, the REST architecture applies constraints to how these components should run to be

scalable. The constraints are usually applied to web services where HTTP APIs that followed the REST constraints are called RESTful web services.

REST is an architecture design that is not restricted to a particular implementation such as HTTP. This allows RESTful principles to be applied to other systems such as Quiver. In fact, stream handler in Quiver is designed such that it can easily be converted into a RESTful HTTP API. On the contrary it is difficult to convert a non-RESTful HTTP API into a quiver stream handler.

In his dissertation Fielding describes six constraints that are applied to a RESTful architecture: (Roy Thomas Fielding 2000)(Roy T. Fielding and Taylor 2002)

2.2.1 Client-Server

The benefit client-server architecture is clearly seen in HTTP, where the communication is between a HTTP client and HTTP server. In Quiver a stream handler take the same role as HTTP server which accept requests coming from any type client. The client of a stream handler can come from external protocols such as HTTP, or it can also be called internally by other stream handlers or quiver components. This also makes Quiver a peer-to-peer architecture since stream handlers can act as client to make request to other stream handlers.

2.2.2 Stateless

The stateless constraint prevent servers from keeping track of client states. This allows a RESTful service to scale horizontally by dispatching clients to get served by different servers depending on the work load. The constraint also allow servers to gracefully fail, by which clients can simply be rerouted to another server with clean state.

Following this REST constraint, components in Quiver are explicitly assumed to be stateless. While the JavaScript language makes it possible for quiver components to maintain internal state, the system do not make guarantee that requests coming from the same client will be served by the same stream handler.

2.2.3 Cacheable

The caching constraint require a REST implementation to enable content generated to be cacheable by the client. This caching is also partially derived from the stateless constraint, as content would otherwise be non-cacheable if the server generates different content based on the state of the client. A RESTful web service that have its content cacheable would allow clients to retrieve content from the cache layers instead of requesting it from the server all the time, thereby significantly reduce the work load of the server.

Quiver itself do not enforce the cacheability constraint on its stream handlers. However stream handlers that follow the RESTful principles would be able to benefit from the

caching facilities in Quiver and make the component scalable.

2.2.4 Layered System

The layered system in REST allow arbitrary intermediaries to stand between a client-server communication. This allows intermediaries such as cache and load balancer to improve the scalability of the server without the client noticing.

Quiver implements the REST layered system by introducing generic filters. Quiver separates the concerns such as caching and authentication by delegating each concerns to different stream filters. For example a cache stream filter would able to intercept the result stream returned a stream handler and store it in a cache. The same filter would later intercept incoming requests to the stream handler and return the cached result without the request reaching the original stream handler.

2.2.5 Uniform Interface

The uniform interface constraint requires client and server to maintain the same external interface so that implementation can be changed independently of each others. In RESTful web services resources are usually represented as RESTful URIs that can be manipulated through different HTTP methods, namely GET, POST, PUT, and DELETE.

For this constraint Quiver actually deviates from the RESTful principles and instead adopt the Unix philosophy of creating components that do one thing well. In Quiver, access to resources is localized and components can only retrieve resources from stream handlers that are connected with them.

Quiver follows the Unix command-like approach of doing only one specific task. Instead of having one stream handler to handle the create, read, update, and delete of a specific resource, the tasks are divided into four different stream handlers that have specialized manipulation of a given resource.

2.2.6 Code on Demand

The code on demand constraint allows servers to send executable code such as JavaScript code to the client. This constraint has allowed the creation of rich client web applications especially with the rise of HTML5 in recent years.

This is the only optional constraint in REST, and Quiver do not have this constraint implemented at the current moment. Although through features in JavaScript it is possible to pass higher order functions between different quiver components, this practice is explicitly discouraged by Quiver at the current moment.

This is mainly because there is no simple way to serialize a JavaScript function and pass it over the network. As a result components that rely on passing functions around cannot be separately placed at different servers through the load balancing facilities in

Quiver.

2.3 Flow Based Programming

Quiver shares many similarity with flow based programming (FBP). As the inventor of FBP, J. Paul Morrison, describes:

> "In computer programming, flow-based programming (FBP) is a programming paradigm that defines applications as networks of"black box" processes, which exchange data across predefined connections by message passing, where the connections are specified externally to the processes. These black box processes can be reconnected endlessly to form different applications without having to be changed internally. FBP is thus naturally component-oriented." (Morrison 2010)

The main difference between Quiver and FBP is that Quiver accepts only one input stream and return one result stream, while in flow based programming model there may be more than one input or output streams. According to Fielding, the architecture model for Quiver is called Uniform Pipe and Filter (UPF) while FBP have the Pipe and Filter (PF) model. (Roy Thomas Fielding 2000, 41–42)

In Quiver I choose the UPF model over multiple input/output mainly for simplicity. Components with uniform interfaces are easier to be composed. On the contrary components that accept different number of input/output can only be composed with other components with the same number of input/output.

2.3.1 Comparison with FBP

Following is a brief comparison of the differences between FBP and Quiver:

- *Server-side Application* - Quiver currently focus exclusively on server-side web applications. The design of quiver components follow the same single-request/single-response model of HTTP. It also follow RESTful principles and the Unix Philosophy way of creating scalable and modular components.
- *Visual Programming* - Quiver currently do *not* rely on any visual programming elements to make it easier for programming. Rather Quiver focus on the classical way of creating many design patterns that developers can apply when writing their components.
- *Protocol Neutral* - Quiver stream handlers are designed to be protocol neutral. The input and result stream from a stream handler can be adapted to come from any stream source, such as HTTP or command line.

- *Graph Configuration* - Quiver component graph is configured *once* during setup time. The same component graph is then reused for multiple concurrent stream processing. Each processing starts with an input stream, which data arrive in chunks asynchronously.

- *Dependency Management* - Quiver component graphs are composed at constructor level. Quiver constructors are plain functions with standard signatures to accept at least a `config` parameter, which is used to inject dependencies to the component. Therefore it is very easy to reconfigure components simply through basic programming.

- *Input/Output Count* - Quiver components have only one input and one output. Nevertheless it is possible to create component graph of multiple input/output by splitting or multiplexing it using libraries provided. However graphs with extra input/output are hidden from external view, or one have to design non-standard function signatures to enable such composition.

- *Polymorphic Input/Output* - Quiver allows components to accept/return polymorphic input/output with the streamable object, which can act as either stream or plain JavaScript objects like string or json.

- *Intermediary Composition* - Quiver filters can wrap around other components and intercept their input, output, and error. The pattern is similar to Aspect-Oriented Programming and allow separation of concerns in component implementation.

- *Error Handling* - Quiver has explicit error handling mechanism in all its components using the standard asynchronous callback error convention in Node. This allow complex component graph to fail in a fast and consistent way. It also allow intermediaries to intercept errors and make possible recovery.

- *Back Pressure* - Quiver streams have implicit back pressure control without having to rely on techniques like pause method. Therefore the component graph have implicit flow control and do not suffer from problems like producing result streams faster than the network bandwidth.

- *Stream Forwarding* - Quiver components forward the whole stream as an object to another component. It avoids the overhead of having to forward individual data packets across too many layers of intermediaries.

- *Manual Composition* - The Quiver component system is completely optional to use. It is possible to manually combine components together without help from the system, albeit with a lot boilerplate code.

2.3.2 Kamaelia

Kamaelia is a FBP implementation in Python, developed by BBC to serve petabytes of media content to millions of viewers. Components in Kamaelia are written as Python generator functions that achieve concurrency through the `yield` statement. The yielding of components would return control to a higher layer Kamaelia scheduler component. This makes Kamaelia a cooperative multitasking system. (Sparks 2005)

One problem with the concurrency model in Kamaelia is the restriction of `yield` being usable only within the generator function body. This makes it hard for components to perform non-synchronous tasks except by delegating it to other components. Because of this Kamaelia is more framework-like in the sense that Python programmers have to buy into its entire concurrency model and would have difficulty in reusing code with other concurrency frameworks like Twisted.

Nevertheless because there is lack of better concurrency model available in Python, Kamaelia's architecture open possibility to develop highly concurrent Python applications that otherwise cannot be done through traditional synchronous programming.

2.3.3 NoFlo

NoFlo is a FBP implementation in CoffeeScript that was developed in 2011 and recently funded through Kickstarter.[1] The primary selling point of NoFlo is its graphical interface that allows programmers to create complex applications by wiring components graphically without writing code.

NoFlo adopts the FBP architecture and in some ways is similar to Quiver. It compiles from CoffeeScript to JavaScript and run on Node.js. NoFlo have similar programming style as Kamaelia, with a major difference that it make use of event-based concurrency model. Due to the flexibility of asynchronous and event-based concurrency in Node.js, it is easier for programmers to mix NoFlo code with other Node.js libraries.

There is however some significant differences in how code is written in NoFlo than in Quiver. NoFlo makes use of event-based pattern and use object-oriented classes to implement its components. By contrast, the asynchronous quiver stream in Quiver provide better flow control and is not using object oriented programming. Other than that Quiver have made other improvement over FBP as described earlier.

2.4 Microservice

The concept behind Quiver is not new. They have been slowly developed for many years and is collectively known as the Microservice architecture. The term microservice was first popularized in 2013 in an article written by James Hughes (Hughes 2013). Following

[1] NoFlo Kicktarter project page: http://www.kickstarter.com/projects/noflo/noflo-development-environment

that Martin Fowler best summarize microservice as follow:

> *"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare mininum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."* (Fowler and Lewis 2014)

Microservice is sometimes considered as one form of Service Oriented Architecture (SOA), although some proponents reject the SOA label altogether due to SOA means too many things.

On the conceptual level, Quiver has very similar goal in mind with microservice. However Quiver takes much different approach on implementing the concepts. This makes Quiver architecture very distinct from other systems that implement the microservice architecture.

One obvious difference is the lack of protocol defined for quiver components to communicate across network. Instead quiver components communicate through standardized function interfaces. The local interface gives the appearance of all quiver components are communicating locally in the same system. At the same time the interface is designed to allow the creation of custom proxy components to serialize requests using custom protocols.

Since Quiver allow local communication between components by default, the overhead is significantly lower than typical microservice systems which only allow remote communication. Furthermore the component interface in Quiver is designed with optimization in mind, which allow the passing of local JavaScript objects when both components exist locally.

Chapter 3

Architecture

3.1 Overview

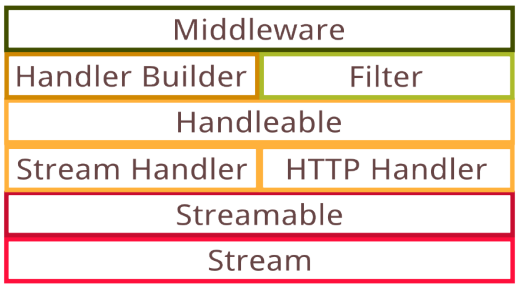

Figure 3.1: Quiver Architecture Layers

When developing Quiver the most difficult challenge was to breaking down problems into smaller parts and solving each of them at a different layers. Many of the problems are interconnected, and therefore it is tempting to create one big solution that can solve all problems together. By splitting the architecture into several layers, the whole solution become radically simplified to just a collection of seemingly trivial function signatures.

The Quiver architecture is consist of six fundamental layers building on top of one another. This chapter will present each of the layers in detail and show how they are relate to each others. We will first discuss about the design of quiver stream and how it differs from conventional stream implementations. The streamable construct will then be introduced as an opaque stream supertype that is convertible between stream and other JavaScript objects. Following that various design of stream processing function signatures

is discussed, and the stream handler construct is introduced.

We will also talk about http handler construct which is made different from the simpler stream handler. The handleable construct will then be introduced as a handler supertype for both stream and http handler. The issue of dependency management is then discussed next, and the handler builder construct will be instroduced as the constructor function for handlers.

Finally the techniques of handler extensions is discussed with the introduction of filter, which accept a handler and extend it through composition. Middleware is the last construct to be introduced, with the power of extending existing handler builders.

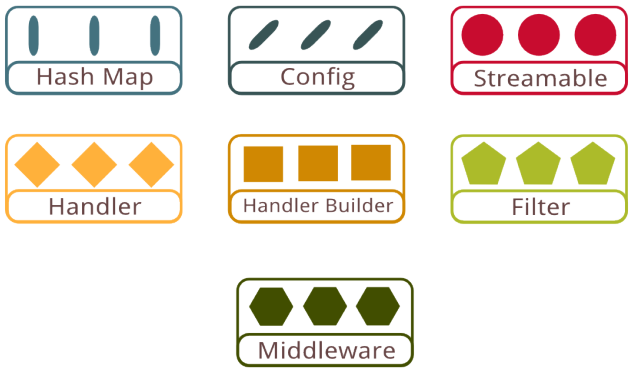

Figure 3.2: Quiver Object Types

3.1.1 Illustration Guide

Different object types are illustrated in different colors and shapes to make it easy to identify the relationship. At the basic level, an object is shown as a rectangle box containing text label and three identical colored shapes. The different shapes used are mainly to aid viewing of this literature printed on black-and-white paper.

The number of identical shapes in a component figure do not carry significant meaning. Instead the difference in number is used to differentiate multiple objects of the same type but with different content.

As an example, figure 3.3 illustrates the interface of a stream handler object, shown as orange box with diamond shape. The arrows indicate that the stream handler is a function that accepts an args object and a streamable object, and asynchronously return a new streamable object. Notice that the two streamable objects have different number of circle shapes inside them, indicating that the streamable returned by stream handler may be different from the streamable that it received.

Figure 3.3: Stream handler

3.1.2 API Convention

A pseudocode coding convention is designed to make it easy to describe APIs available in Quiver. Following is a specification for an example API function:

```
api asyncFunction = function(args, callback(err, result));
```

The keyword `api` is used to indicate that the given code is an API specification. The `function` keyword following the equal sign `=` indicates that the API is a JavaScript function of the given signature. As Quiver runs on Node which is asynchronous in nature, its API functions also typically expects a callback function passed as the last argument. This is specified with the `callback` keyword followed by bracket containing arguments that the callback function expects. Following the Node convention, a callback function typically expects an error object `err` as its first argument, followed by the result.

```
api plainObject = { fieldName, [optionalField], ... }

api syncFunction = function(args) -> result;
```

There are also some API for objects and synchronous functions. The example code above shows the API interface for a plain object, with a compulsory field `fieldName` and an optional field `optionalField`. The object may also have other optional fields, as indicated by the `...` at the end. The next example is a synchronous function API, which the result returned by the function is indicated after the arrow sign `->`.

It is usually self-explanatory of argument types in a function parameter, as they can usually be deduced based on the argument name.

3.2 Stream

Stream is the most basic building block in Quiver which allows the transfer of data streams. The stream architecture in Quiver follows a simple producer-consumer pattern in which a one producer writes data into the stream channel to be read by a single consumer on the other side.

A quiver stream is created as a pair of write stream and read stream as distinct objects. The linkage between the write stream and read stream is encapsulated. By making use of the scope constraints in JavaScript, it is made not possible to obtain the write stream

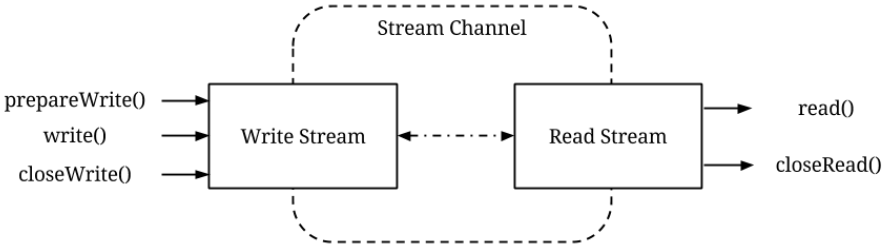

Figure 3.4: Stream channel

counterpart from a read stream and vice versa.

Reading and writing to quiver streams is also asynchronous. When a reader calls the `read()` method on the read stream, the data is returned asynchronously to the provided callback only when the writer has written data to the write stream. Similarly by best practice writers may optionally call the `prepareWrite()` method of the write stream and get notified in the callback when reader on the other side is ready to read the data.

The design of quiver stream is very different from the stream implementation in many other libraries and frameworks. In most implementations the concrete streams are typically implemented as a single stream object which can be upcasted into either read stream or write stream depending on the function that accepts the stream object. Other than that there are typically many different stream implementations available based on the source or destination of the stream, i.e. file streams and network streams.

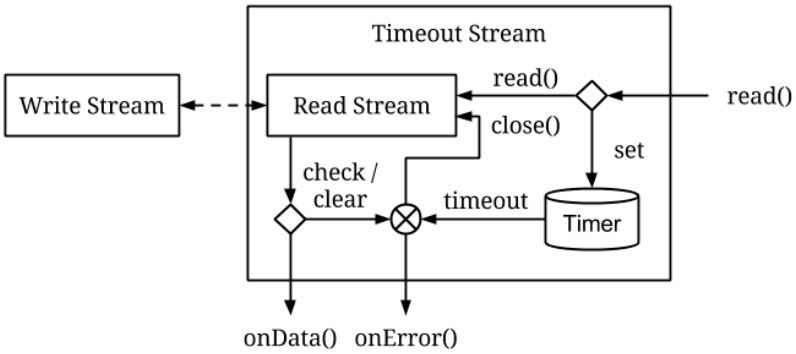

Figure 3.5: Timeout Stream

On the contrary, quiver streams are solely plain channels made for transferring data

between two entities. Additional functionalities to the stream is built on top of it using functional composition. For example, instead of creating a stream implementation that supports read time out, figure 3.5 shows a utility function is made to accept a read stream and return a new read stream that guarantees read callback after certain timeout.

As Quiver do not make use of object-oriented programming, there is no abstract class available for programmers to "extend" into different specialized streams. Instead, programmers create functions that read raw data from specific data source and then write them to a quiver write stream. The read stream counterpart is then returned to the user, making it appear as if the function creates a specialized stream.

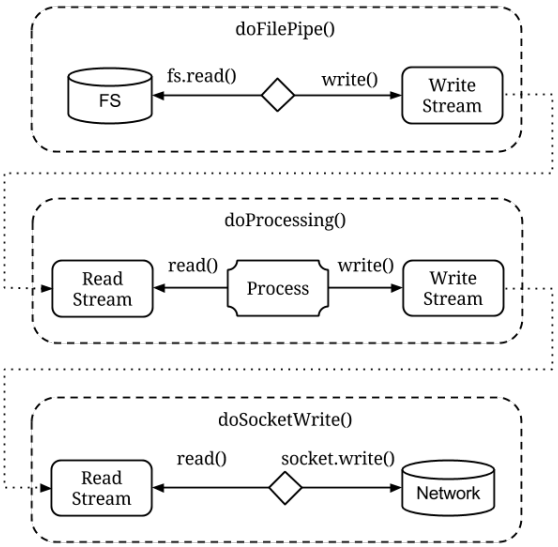

Figure 3.6: Example Pipeline

Figure 3.6 shows the architecture of an example program that reads content from the file system, process the data and write it to the network. The data flow is split into three functions. The first function reads the content from the file system and writes it to a write stream. The second function reads the read stream that was connected to the previous write stream, process the data and then write the result to another write stream. Finally the third function reads the previously connected read stream and write the data to the network socket.

One of the benefits of quiver streams is the implicit rate control that limits the rate of functions producing data to write to the stream. Before programs can write data to a write stream, it may first call the `prepareWrite()` method of the write stream and

get notified when reader on the other side is ready for reading. in Figure 3.6 example the network is slow at writing the produced data to the socket. This would in turn slow down the rate of processing, which slow down the rate of reading from the file stream. This design also allow programmers to add a throttling component to the stream in ways similar to the stream timeout implementation.

3.2.1 Conventional Stream

Contrary to the architecture in Quiver, figure 3.7 shows the architecture of the same program implemented in conventional streams. Typically there is an abstract base class for the read/write (input/output) streams. Programmers then have to implement concrete stream classes that implement actual functionalities of how to read/write data from the stream. In the example we have a concrete FileInputStream class and NetworkOutputStream class implemented. In typical implementations the abstract stream class would have non-overridable `read/write` method, which will internally call some protected `_read/_write` method that is implemented by the concrete child classes. Finally a process function is used to read from the file input stream and write to the network output stream.

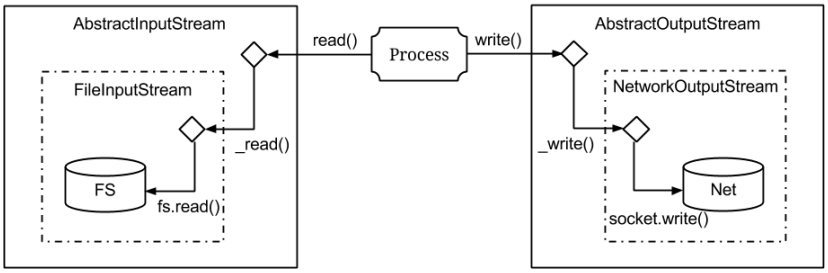

Figure 3.7: Conventional Pipeline

The two examples above show a vast distinction between the approach used for implementing streams. There are several problems in the conventional implementation of streams. The first problem is that it influence the programmer to think in terms of implementing a read/write method. It is also a passive approach as the code can only run when the user calls the method. The naming is also unnatural and confusing, as from the implementor's perspective he is really performing the opposite action. On the other hand Quiver allows programmers to implement the read/write actions happening in a natural way, making it easy to understand both sides of the code.

Another problem of the conventional stream pattern is that it is really hard to divide the processing into more than three components. Because there is no simple channels that redirect data written to another read stream, the processing code in the code have no choice but to implement all processing code as one monolithic function. Implementors of intermediary processing have to otherwise implement their code as a transform stream subclass of the original concrete class. Such approach gets complicated quickly and programmers quickly get lost in connecting the code through confusingly named inherited methods.

In conclusion, the minimalist approach to design quiver streams is what makes the implementation so powerful. The minimal interface specification of a quiver stream allows programmers to create wrapper around it to extend its functionality. It also requires minimal amount of code to create an alternative implementation of the stream that can be transparently replaced. The minimal functionality and implementation details also makes it easy to ensure the implementation is bug free, and extensibility through composition makes sure that new bugs can only be introduced at the higher layer of the code.

One could argue that there are some other features in conventional stream that otherwise can't be implemented in quiver streams. In the following sections I will show how these problems are solved by delegating them to higher level constructs.

3.2.2 API Specification

```
api channel = { readStream, writeStream };

api readStream = { read, closeRead };

api writeStream = { write, prepareWrite, closeWrite };
```

Create a pair of connected read/write stream through the `quiver-stream-channel` library. The two end points are usually then passed to different functions that are responsible for the read or write operation. Content written into the write stream will be read from the read stream at the other end.

```
api readStream.read = function(callback(streamClosed, data));
```

Attempt to read data from a readStream, which the read result will be pass to readCallback asynchronously.

`streamClosed` is non-null if when the end of stream is reached. If error occured during read, the error value can be retrieved from `streamClosed.err`. Otherwise the data variable contain the read value, which is typically a single buffer object. `streamClosed` and data exist exclusively from each other, which mean if is streamClosed non-null then data is guaranteed to be null.

```
api readStream.closeRead = function(err);
```

Close the read stream optionally with an error value.

```
api writeStream.write = function(data);
```

Write data to a write stream immediately regardless of whether the reader on the other side is ready to read it. Calling this without prepareWrite gives the risk of having too much data buffered within the stream without reader reading it quick enough.

```
api writeStream.prepareWrite = function(callback(streamClosed));
```

Prepare to write to the write stream once the reader on the other side is ready, in which writeCallback will be called asynchronously. It is prohibited to call the write stream's `write()` method while `prepareWrite()` is waiting for its write callback. Such action do not have well defined semantics to the stream channel state and will cause an exception to be thrown.

Non-null streamClosed indicates that the stream has been prematurely closed and the writer should cancel all write operations. Otherwise it is now optimal for the writer to call `writeStream.write()`.

```
api writeStream.closeWrite = function(err);
```

Close the write stream with an optional error value.

3.3 Streamable

One of the problems of passing streams around as function arguments is that the stream typically have to be as a whole parsed into data structures that the function body can understand. On the other hand when a function produces data that it wish to write to the stream, the result data is typically held in a data structure that is later serialized into raw bytes.

Since parsing and serializing data takes a non-trivial amount of computation, what we want is to associate a data structure with a stream so that the two can be retrieved interchangibly depending on the format required. Moreover, there are usually more than two ways to represent a piece of data.

Take Json for example, a json data is a stream of Unicode characters written in a specific format. That same data can be represented as a plain JavaScript object, a JavaScript string, or a stream of raw bytes. When passing a Json data object around, it has to be possible to convert between these different formats.

In Quiver instead of directly passing stream objects around, the problem is solved by wrapping the stream inside another object called *streamable*. A streamable is a plain JavaScript object that is at least convertible into a read stream through a compulsory `toStream()` method. Other than that, data structures that are equivalent to the stream content are retrievable with methods of format `toXXXX()`, where `XXXX` is the type of data structure.

Figure 3.8 shows an example of a json streamable object, containing the basic `toStream()`, `toJson()`, and `toText()`[1] methods. On top of that the streamable has a

[1]The reserved method name `toString()` have a different usage in JavaScript therefore the method

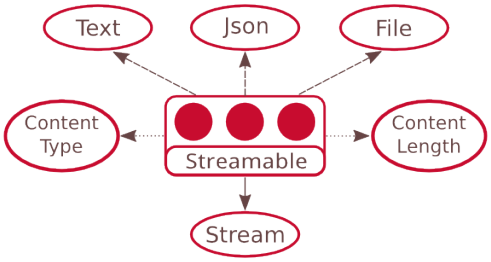

Figure 3.8: Streamable

`toFile()` method returning a file path, indicating that the streamable was constructed from a json file on local filesystem. Other than that the example streamable object also carry the optional `contentType` and `contentLength` attributes, which are metadata about the stream which can be useful for tasks such as constructing HTTP headers.

Streamable is designed as a plain JavaScript object, meaning that it do not contain any implementation detail of the data structures it represent. The `toXXXX()` methods perform one-way conversion return independent data structures that have no connection back to the original streamable object.

```
streamConvert.streamableToJson = function(streamable, callback) {
  if(streamable.toJson) return streamable.toJson(callback)

  streamableToText(streamable, function(err, text) {
    if(err) return callback(err)

    var json = JSON.parse(text)

    streamable.toJson = function(callback) {
      callback(null, json)
    }

    callback(null, json)
  })
}
```

Being plain old objects also allow external functions to easily add new formats to the streamable object by adding a new key to it. The above code demonstrates how the `quiver-stream-convert` library converts a streamable to json object. When first processing a streamable that contain only raw Json stream, the code can check whether

`toText()` is used for converting streamable to string.

the streamable object has a `toJson()` method which it can make use of. If not it will obtain the raw stream through `toStream()` and attempt to convert the stream into a single string. (which can optionally be done through `toText()`.) Once the string is obtained it is parsed using `JSON.parse()`. If the streamable is reusable, the resulting Json object is then stored to the original streamable by adding a new `toJson()` method that return a new copy of the Json object each time it is called.

The `toStream()` method on streamable also makes it possible to open a data stream multiple times for different readers. This is however on the condition that the streamable is reusable, as indicated by its `reusable` attribute. By default streamables are non-reusable because it would otherwise have to hold reference to the entire stream content until the streamable is garbage collected, which can cause memory performance issues. However when enabled, the reusable `toStream()` method makes streamable a convenient way to pass to functions that may have multiple readers. With this it also solves the problem of single ownership requirement of quiver streams by enabling multiple-ownership at a higher layer.

The design of streamable is contrary to conventional ways of implementing objects that represent multiple data structures. Conventionally such object is implemented through multiple inheritance or implementing multiple interfaces. The object would then have to either inherit the concrete methods or implement proxy methods to emulate the behavior of all formats it represent. This usually leads to complicated implementation that are prone to various problems such as naming conflict. Implementations inheriting from mutable data structures also have the problem of users mutating one of its representations and causing stale updates.

In conclusion, by having one-way conversion to independent data formats, streamable provides a flexible way to merge different data structures without conflicting implementations. The conversion methods also prevent stale updates by returning a copy of data structure that cannot be linked back regardless of mutation.

3.3.1 API Reference

```
api streamable = { toStream, [toText], [toJson], ... };

api streamable.toStream = function(callback(err, readStream));
```

Returns a quiver read stream representation of the data inside the streamable.

```
api streamable.toXXXXX = function(callback(err, result));
```

If a function with prefix “to” exist on the streamable, it can be used to convert the streamable object to the respective data form. Example of methods are `toText()` and `toJson()`.

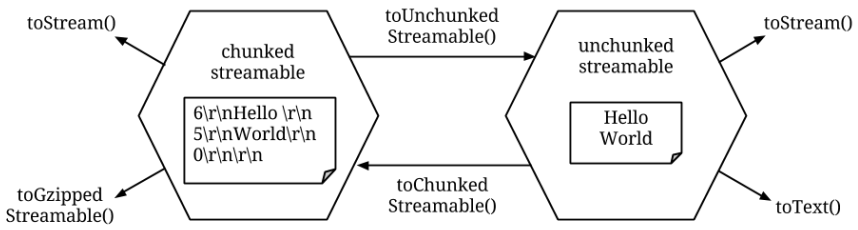

Figure 3.9: Chunked Streamable Example

3.3.2 Streamable Enccoding Conversion

One requirement for streamable is that the data structure attached to it must have equivalent value with all other representation of the same stream, including the raw stream content. This creates an issue when trying to convert between encoded stream and raw stream.

There are many stream encoding that are designed to be transported efficiently over the network, for example compression, endianness, and the chunked transfer encoding. The stream content has to be decoded before converting the stream to their respective data structure.

It does not make sense to attach conversion functions to a streamable containing encoded stream, because the encoded stream is not directly equivalent to the respective data structure. On the other hand it is sometimes desirable to cache both the encoded and decoded content of the stream so that there is no need to re-encode/decode the content when transferring it over network.

The solution we have with this problem is shown in Figure 3.9. The figure shows two streamables that are convertible between each others through the chunked transfer encoding/decoding. The chunk encoded streamable has a `toStream()` method that returns the encoded stream content. On the other hand only the decoded streamable has a `toText()` method that converts the decoded stream content into string.

This design of coupled streamables not only cleanly separate the semantics between encoded and decoded objects. It also allow stacking of multiple encodings to form through a connected chain of streamables. By navigating through the chains one can convert between all data formats and encodings without facing any ambiguity in their actions.

3.3.3 Streamable Best Practice

Stream is the primary interface for different quiver components to communicate. The simplest concept of stream communication is with a function that reads data from a read stream and write its result to a write stream.

```
api processStream1 = function(readStream, writeStream);
```

A naive signature for such a function is to accept a read stream and a write stream as its argument. However it is more difficult to to compose functions with side effects as compared to pure functions. Even though Quiver is not strictly purely functional, for most practical purposes read streams can be treated like a read-only array of data, making it as effective as immutable arguments.

```
api processStream2 = function(readStream,
  callback(err, readStream));
```

A better approach of writing the same function is to create a function that accepts a read stream and asynchronously return a result read stream. The result stream can be created within the function by creating a pair of read/write stream and return the read stream part to the caller. The function would then have exclusive access to the write stream and be confident that the reader on the other side will receive the exact data it sent.

The asynchronous approach is also better at error handling through the first err argument in the callback. This is more clear than the first synchronous example, which the function has no way to refine the error scope other than closing the write stream immediately.

```
api processStream3 = function(streamable,
  callback(err, streamable));
```

With the introduction of streamable, there is an even better approach to rewrite the same function to accept streamable in place of read stream in the argument and result. Through this simple indirection this function signature become capable of accepting any serializable data type without sacrificing performance penalty of unnecessary conversions.

3.4 Stream Handler

Figure 3.10: Stream handler

Quiver stream handler is designed following the Unix Philosophy. Unix process is well known for their extensibility through the uniform stream interfaces `STDIN`, `STDOUT`, and

`STDERR`. Similarly stream handler is designed to have uniform interface across all kinds of programs to make them as easily composable as Unix processes. All quiver stream handler accept an `args` as their first argument, which is the equivalent of command line arguments. The second argument inputStreamable is a streamable object corresponds to STDIN. Finally the handler asynchronously returns either an error (STDERR) or a resultStreamable (STDOUT).

3.4.1 API Specification

```
api streamHandler = function(args, inputStreamable,
  callback(err, resultStreamable));
```

The `processStream3()` function signature arrived in the previous section is very close to the interface of a stream handler. Other than accepting and returning streamable, a stream handler also accepts an additional `args` argument which serve as key-value argument. The `args` argument allow stream handler functions to accept arbitrary number of keyword arguments while all have the same function signature.

3.4.2 Stream Handler and HTTP

Stream handler works similarly with typical HTTP handlers, which accept a HTTP request and return a HTTP response. In a quiver web application some stream handlers are mapped to the front-facing HTTP server to process HTTP requests. But most of the time stream handlers are used internally by forming a graph of stream handlers that communicate with each others. More importantly, stream handler is designed to be protocol-neutral and can be adapted to use for protocols other than HTTP.

Stream handler is not obligated to understand standard HTTP headers such as transfer encoding, making it much simpler to implement and understand. The `args` argument is more often mapped to the query string parameters in HTTP request, but is also sometimes mapped to certain HTTP headers as long as it does not affect the interpretation of the input streamable content.

Other than that stream handler only return a result streamable which correspond to the HTTP response body, but there is no equivalance of HTTP response headers. Although the result streamable may contain optional meta data such as content type and content length, it is not allowed to carry information that affect the interpretation of the stream content such as transfer encoding and cookies.

On error the stream handler return an error object optionally containing HTTP error status code and a short error message to be written to the response body. By default undefined or invalid error code will cause the server to return status code 500 Internal Server Error.

Stream handler is designed to handle a subset of HTTP that follows Fielding's RESTful architecture design. With minimal protocol constraint, implementing stream handlers

is almost as simple as writing plain JavaScript functions. When stream handlers are implemented to be pure from side effects, there are many functional programming techniques that can be applied to the stream handler to gain significant performance optimization from it.

3.5 HTTP Handler

Figure 3.11: HTTP handler

Stream handler is designed to free users from the complexity of HTTP and allow handler functions to be composed more easily. While the `args` argument can loosely related to HTTP request header, in the handler callback there only exist the resultStreamable which loosely corresponds to HTTP response body but there is no correspondance of HTTP response header. The lack of response header equivalent is intentional as HTTP request and response headers alter the semantics of the input and result stream and therefore make it hard to understand.

Instead of using stream handler, Quiver has the HTTP handler construct used for handling full HTTP requests.

3.5.1 API Specification

```
api httpHandler = function(requestHead, requestStreamable,
  callback(err, responseHead, responseStreamable));
```

Unlike the original Node http handler, quiver http handler separates the HTTP header part from the HTTP body. It is designed such that it is easy for intermediaries to strip/add HTTP headers that alter the content of the body, such as Content-Encoding and Transfer-Encoding, and supply a different stream body independent of the original header.

The `requestHead` and `responseHead` format is compatible with the existing node API for the `request` object, minus the node stream and event API. They are also made of plain JavaScript objects and can be easily created using object literals.

3.6 Handleable

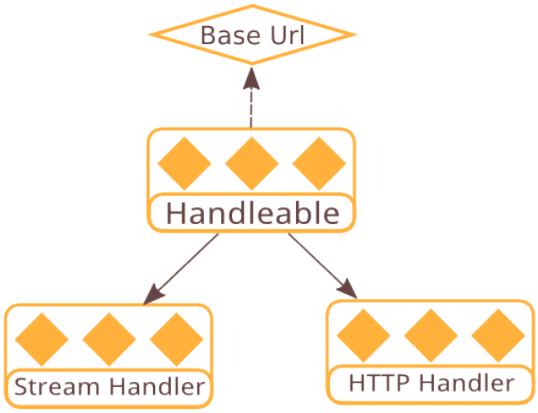

Figure 3.12: Handleable

There are currently two types of handlers in Quiver: stream handler and http handler. Handler types are differentiated based on their function signature, and there is no class hierarchy or any common feature shared among handlers. There are times when one might want to create a handler that is usable as both stream and http handler, or write filters or middlewares that can be applied to both types of handlers.

There is also common needs of attaching meta information to a handler, such as serializing it to URL or filesystem path. When composing handler components together using higher level constructs, it is also necessary to ensure the handler type of different components matches. To solve these problems Quiver make use of the same concept as streamable and create a handler supertype called *handleable*.

3.6.1 API Specification

```
api handleable = { [toStreamHandler], [toHttpHandler], ... };

api handleable.toStreamHandler = function() -> streamHandler;

api handleable.toHttpHandler = function() -> httpHandler;
```

Like streamable, a handleable is a plain JavaScript object that has `toXXX()` methods to convert it to the right handler type. But unlike streamable, handleable convertion function return in synchronous style.

3.7 Handler Builder

Figure 3.13: Handler Builder

Stream handler and HTTP handler makes it easy for writing composable handler functions. While it is useful to pass handler functions around and compose them directly, one would find that handler implementations typically have dependency on application-wide configurations such as database credentials and file system paths. Before getting the handlers there is a need for the handlers to receive these configurations without exposing them on global scope. With that the *handler builder* is designed as a construction function to provide uniform way of configuring handlers.

3.7.1 API Specification

```
api handlerBuilder = function(config, callback(err, handler));
```

A Handler builder accepts a plain JavaScript object `config` and asynchronously return a handler instance. The `config` parameter should contain all parameters the handler needs, and the handler builder creates a handler function closure that captures the configurations.

3.7.2 Dependency Management

Handler builder also solves the dependency management problem. Very often different parts of web application have dependencies on some global configuration, such as database credentials. The usual solution for many web frameworks is for users to define application-wide globals to deliver the dependencies to the code in framework-specific ways.

Quiver solves the dependency problem by operating at the handler builder level, which enable explicit passing of configuration through the constructor function. With that the handler builder function can return instantiated handler function that capture the dependencies inside closures.

```
var handlerBuilder = function(config, callback) {
  var database = createDatabase(config.dbUrl,
```

```
    config.dbUsername, config.dbPassword)

  var handler = function(args, inputStreamable, callback) {
    database.query(...)
  }

  callback(null, handler)
}
```

The example code above shows a handler builder function that instantiate a database instance based on the database credentials provided in config. It then create a handler function closure which capture the database variable and use it when handling requests.

One problem with the example shown is that the handler builder have hard dependency to the database constructor, as it needs to know the dependencies of the database constructor and know how to build the database instance correctly. There is a better way to handle such dependency: the handler builder can leave the database construction as external dependency and instead require dependency on instantiated database instance:

```
var handlerBuilder = function(config, callback) {
  var database = config.database
  ...
}
```

3.7.3 Handleable Builder

The handler builder constructor pattern can be applied to any kind of handler or object in general. For easy reference the unqualified name "handler builder" is used to refer to handler constructor of any kind, including stream handler builder, http handler builder, and handleable builder.

Of all the handleable builder is used in Quiver as the supertype of all handler builders. As explained in coming sections, the quiver component system convert all handler builders into handleable builders and make use of the handleable type system to verify the type of handlers created.

3.8 Filter

Handlers and handler builders are just plain functions without class. Therefore it is not possible to extend a handler through classical inheritance. Instead, the *filter* construct is introduced to allow flexible handler extension through functional composition.

3.8.1 API Specification

```
api filter = function(config, handler,
```

Figure 3.14: Filter

```
callback(err, filteredHandler));
```

A filter have similar signature as handler builder, except that it has an additional parameter that accepts the original handler that it extends. Based on the supplied handler and the filter's configuration dependencies, the filter create an extended *filteredHandler* closure to encapsulate the original handler.

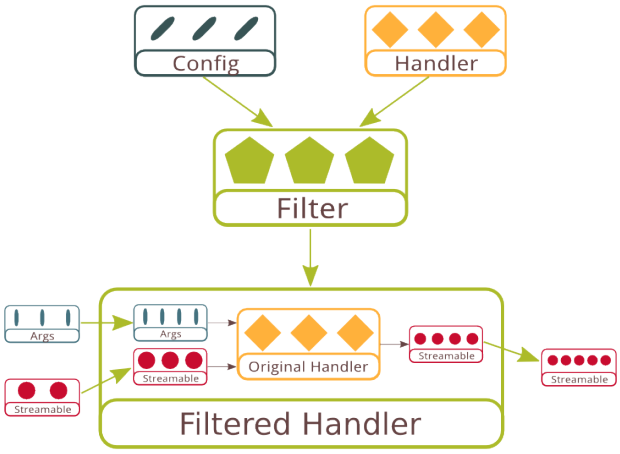

Figure 3.15: Filter in Details

Figure 3.15 shows a more detailed view inside of the filtered handler returned. The filter accepts its original handler as a black box but has full power to access all its input and output. In other words, it can make changes to `args` and `inputStreamable` before passing it to the original handler, and it can intercept `err` and `resultStreamable` produced by the original handler before returning to its caller. Moreover, a filter can decide to skip the original handler entirely and produce result on behalf of it.

Filter solves the problem of separation of concern by allowing developers to implement cross-cutting concerns as reusable filters. One can implement any of the `before()`,

`after()`, or `around()` paradigms in Aspect Oriented programming all in one unified function interface.

3.8.2 Example

One of the common use case for filter is to implement security concerns such as permission control. Below is an example of a stream filter that does permission checking based on a simple array of allowed users in the config.

```
var permissionFilter = function(config, handler, callback) {
  var allowedUsers = config.allowedUsers

  var filteredHandler = function(args, inputStreamable, callback) {
    if(allowedUsers.indexOf(args.user) == -1) return callback(
      error(403, 'Forbidden'))

    handler(args, inputStreamable, callback)
  }

  callback(null, filteredHandler)
}
```

3.8.3 HTTP Filter

Quiver http filter is simple but powerful construct that can be used to build the same intermediary chains exist in other powerful web servers such as Apache. The major difference is that HTTP filters are plain JavaScript functions that can easily be manipulated. Filters can also be easily enabled, disabled, or rearranged by simply manipulating the filter composition chain.

Nevertheless, it is recommended to write http filter only when the functionality is related to HTTP. Otherwise stream filter serve as much simpler alternative for implementing application-specific concerns.

3.8.4 Handleable Filter

Similar to handler builder, the general term "filter" is refered for all handler types in general. Handleable filter is the supertype of all filters. Filters are converted to handleable filter by the quiver component system to make handler type checking possible. For example a stream filter is converted to a handleable filter that ensure the input handleable has a `toStreamHandler()` function before extending the handler.

3.9 Middleware

Figure 3.16: Middleware

Filters are great at extending handlers from the outside, but the handlers they receive are blackboxes that already been instantiated. As a result, a filter has no way to know how the handler itself was instantiated, or how many other filters have been applied to a handler before it arrive. Filter is designed to be easily composable, but there is a problem of ensuring a filter is only applied once in a composition chain.

To solve the problem, interception need to be done at handler construction time and alter the way handler is instantiated. *Middleware* is the superset of filter that does the extension both before and after handler construction.

Unlike filter, a middleware accepts a *handler builder* in place of handler in its second argument. In other words, it has full power to alter the how a handler builder receive its *configuration*, as well as extending the produced handler result.

With it, filter become a simplified middleware that only intercept on the result produced by a handler builder. It is trivial to convert a filter into middleware by making a converted middleware that manipulate handler result returned from handler builder.

3.9.1 API Specification

```
api middleware = function(config, handlerBuilder,
  callback(err, handler));
```

3.9.2 Dependency management

Since middleware are capable of altering the `config` parameter before forwarding it to the handler builder, it is also useful for dependency management by instantitating services the handler builder depend on and putting them into the config.

```
var databaseMiddleware = function(config, handlerBuilder, callback) {
  config.database = createDatabase(config.dbUrl,
    config.dbUsername, config.dbPassword)
```

```
    handlerBuilder(config, callback)
  }
```

The example code above is a database middleware which instantiates a database based on credentials specified in the config. It then injects the database instance into the config and pass it to the handler builder so that it can make use of the database immediately.

Internally, Quiver make extensive use of middleware to preconfigure a user-written handler builder or middleware. For instance it allow filters to depend on other filters while making sure that each filter are only applied once at its outermost composition chain. Middleware also enable Quiver to provide dependency injection service by instantiating other handlers that a handler depend on and injecting it into their `config`.

Middleware are also easily composable with handler builder to create new handler builder:

```
var combineMiddlewareWithHandlerBuilder =
  function(middleware, handlerBuilder) {
    var combinedHandlerBuilder = function(config, callback) {
      middleware(config, handlerBuilder, callback)
    }

    return combinedHandlerBuilder
  }
```

3.9.3 Handleable Middleware

In Quiver handler builder and middleware are the cornerstone building block for all quiver components. Using composition techniques it is possible to reduce a complex handler network graph all the way down to one encapsulated handler builder, all without knowing configuration ahead of time.

The quiver component system convert all components into handleable builder or handleable middleware. This greatly simplifies the component system as at the highest level it only have to handle two component supertype.

3.10 Component System

The previous sections introduced the foundation architecture for Quiver. On top of that there is a final component system built to link components of different types together. The quiver component system is designed to allow components to have complex dependencies among each others. For example, a handler component may have dependency on a filter component, which in turn may have dependencies with other handlers or filters.

The steps of defining and using quiver components is separated into several phases:

- Component definition
- Component loading
- Component conversion
- Component registry

3.10.1 Component Definition

The quiver component system has a JSON-like DSL, which define components in simple key-value pairs. The main difference of the component DSL and JSON is that the component DSL usually has one field with function value. That is used to reference the actual function body of a component, which may be defined separately from the component definition.

A component manifest has another two essential fields: name and type. The name field serves as a unique identifier for the component so that it can be referenced from other components by name. Since the component name is defined as a string, space is often used for easier typing and reading. The type field tells the component system what kind of component it is, so that the system knows how to manage it.

Quiver components are usually defined in separate source files. Each of the source files will make use of Node's module system to export the components inside a `quiverComponents` array.

Following is an example of a simple handler component definition:

```
exports.quiverComponents = [
  {
    name: 'my handler',
    type: 'stream handler',
    handlerBuilder: function(config, callback) { ... }
  }
]
```

3.10.2 Component Loading

There are several library functions available to import quiver components from multiple source files. A component loader provide custom ways to search for the components and then combine them into a single array containing all components.

For example, the function below synchronously load components from all source files under a directory:

```
var components = loadComponentsFromDirectorySync('/path/to/dir')
```

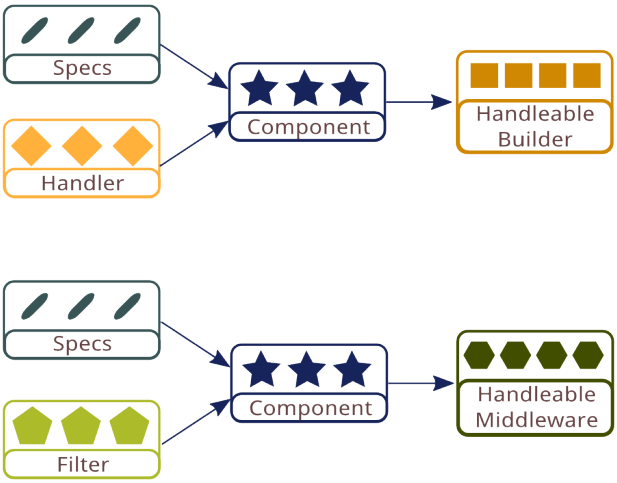

Figure 3.17: Component Conversion

3.10.3 Component Conversion

Quiver component system manages different types of components by converting all of them into two universal component supertypes: handleable builder and handleable middleware. All handler and their builder types, including stream handler and http handler, are converted into handleable builders. Similarly all filter and middleware types are converted into handleable middleware.

The conversion to the handleable supertype allows the quiver component system to not have to deal with each of the specific types separately. It also makes it easy to introduce new handler types in future, since all handler types can be wrapped into the handleable supertype.

The handleable type system also allow implicit type checking against component composition of different handler types. For example if a http filter is requested to be applied to a stream handler, the component system would not type check the filter application at conversion time. Instead the http filter would be converted into a handleable middleware which expects a handleable builder. It then receives the converted handleable builder during component instantiation time, and attempt to convert the result handleable into a http handler. It is by this time that the handleable middleware detects that the result handleable do not have a `toHttpHandler()` method, and therefore produce a runtime error.

3.10.4 Component registry

After converting components into handleable builders and handleable middlewares, the component system stores the result in a registry made of plain object. All handleable builders are stored as a table inside the registry's `quiverHandleableBuilders` field, with the component name as the key. Similarly handleable middlewares are stored inside the registry's `quiverHandleableMiddlewares` field.

As explained in the next section, the registry is later returned and used as a config object for component instantiation. Because of this it is also refered as `componentConfig` or simply `config`. A common step after getting the component registry is to merge it with the user config to get a single config object:

```
installComponents(components, function(err, componentConfig) {
  if(err) return ...

  var config = mergeObject([componentConfg, userConfig])
  ...
})
```

3.10.5 Component Dependencies

Quiver manages component dependencies by name instead of having hard function reference. Because of that the component system is able to convert individual components without having full knowledge of the complete list of components. For example if a handler component depends on a filter component, the system would simply convert it into a managed handleable builder, which later retrieve the converted handleable middleware component by name when it receives a config. it then applies the middleware with the original handler builder to get the desired final result.

```
{
  name: 'my handler',
  type: 'stream handler',
  middlewares: [
    'my filter'
  ],
  handlerBuilder: myHandlerBuilder
}
```

To demonstrate the concept, the component definition above would be converted into a managed handleable builder that is equivalent to the manual composition below:

```
var myHandlerBuilder = function(config, callback) { ... }

var myHandleableBuilder = streamHandlerBuilderToHandleableBuilder(
  myHandlerBuilder)
```

```
var myManagedHandleableBuilder = function(config, callback) {
  var handleableMiddleware = config.quiverMiddlewares['my filter']

  handleableMiddleware(config, myHandleableBuilder, callback)
}

var componentConfig = {
  quiverHandleableBuilders: {
    'my handler': myManagedHandleableBuilder
  }
}
```

It is also worth noting that the component system is similar to a macro system that expands component definition with many boilerplate code. However Quiver is able implement the same effect using functional composition without relying on any macro facilities (which is not supported in Javascript).

3.11 Summary

In this chapter eight types of constructs have been introduced to build the foundation of the Quiver architecture. The architecture may be applied to other programming languages and developers would still benefit from the architecture without the libraries from Quiver.

Following is a short summary for each of the constructs that have been introduced:

- *Stream* is implemented as a single producer-consumer channel. The quiver stream is designed to be minimalistic and delegate extensibility to higher layers.

- *Streamable* encapsulates stream into plain object that may contain conversion methods to other object types. Streamable enables local optimization by allowing local components to pass objects without the overhead of parsing raw streams.

- *Stream Handler* is similar to Unix process that accepts single input/output streams. It accepts input in the form of streamable and produce result by returning the result as another streamable.

- *HTTP Handler* accepts a request head along with request body as streamable, and return a response head along with response body as streamable. It is used when there is need to manipulate the HTTP headers.

- *Handleable* acts as a supertype of stream handler and http handler. It is a plain object with optional conversion functions to convert to specific handler types.

- *Handler Builder* is constructor function for handler. It solves the dependency management problem and provide uniform way to instantiate handlers.
- *Filter* extends existing handlers through composition. A returned filtered handler can intercept all input/output of the original handler and manipulate them to produce augmented results.
- *Middleware* is one more level ahead of filter by extending handler builders instead of handlers. It can affect the way handler is constructed as well as extending the handler result just as filter does.
- *Component* acts as a glueing layer to connect together different types of handlers, filters and middlewares. It eventually convert all components into handleable builder or handleable middleware and compose them based on the component definition.

Chapter 4

Design Patterns

The foundation Quiver architecture is designed as a general programming style without specific solution. Instead, many design patterns are made on top of the architecture to solve specific problems in web development. In this chapter a few of such common patterns are briefly discussed to give readers an idea of how a Quiver application might look like.

4.1 Pipeline

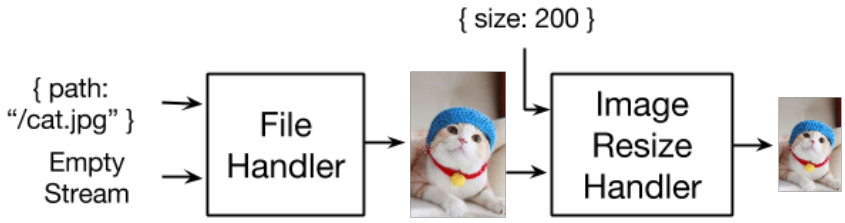

Figure 4.1: Image Resize Pipeline

Stream pipeline is the simplest pattern to demonstrate the basic concepts of Quiver. It consist of combining two or more stream handlers and feed the output of one stream handler as the input of the next stream handler.

Figure 4.1 shows an example pipeline of two handlers to serve image thumbnails from a static file directory. The first handler is a file handler that serves the original image located on a given file path. The second handler is an image resize handler which accepts an image stream and resize the image.

4.2 Config Dependencies

As first shown in chapter 3.16, middleware can be used as a construct to inject configuration dependencies to a handler builder. Common types of dependencies include database connection, remote API credentials, and distributed logging. The code below demonstrates a logger middleware which create custom logger object so that a handler component can be free of concern of where to place error logs.

```
var loggerMiddleware = function(config, handlerBuilder, callback) {
  config.logger = createLogger(...)

  handlerBuilder(config, callback)
}
```

4.3 Args Filter

A common use case in Quiver applications is that stream handlers often need to retrieve complex information from simple arguments before doing actual processing. For example a handler may need to retrieve information about a user based on the user ID provided in its arguments.

Quiver can simplify this kind of common operation and eliminate boilerplate code by using a stream filter. In the example code below, a user info filter is created to extract user information from the user ID provided in `args`. It then insert the user info into `args.userInfo` before forwarding the request to the target handler. With that a handler can retrieve the user information from `args` directly without knowing how to retrieve that information.

```
var handler = function(args, inputStreamable, callback) {
  var userInfo = args.userInfo

  ...
}

var filter = function(config, handler, callback) {
  var filteredHandler = function(args, inputStreamable, callback) {
    var userId = args.userId

    getUserInfo(userId, function(err, userInfo) {
      ...

      args.userInfo = userInfo
      handler(args, inputStreamable, callback)
```

```
    })
  }
  callback(null, filteredHandler)
}
```

4.4 Caching

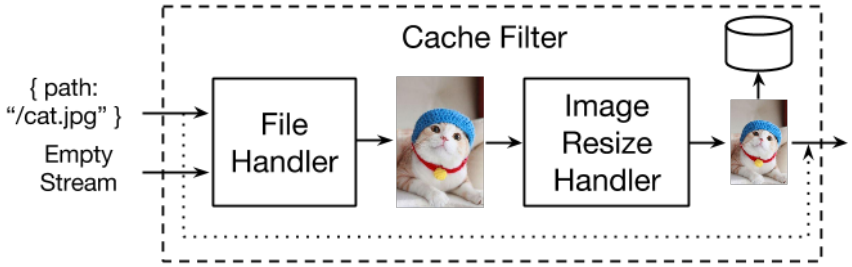

Figure 4.2: Cache Filter

Quiver filters can also be used to cache content returned by a stream handler so that future requests can be served from a cache. Figure 4.2 shows the same image resize pipeline as earlier example, exept that the entire pipeline is now wrapped inside a cache filter.

The previous image thumbnail handler example has a problem that the entire resize pipeline is executed for every requests. As a result the handler may not scale because image resizing is processor intensive. However by simply applying it with a cache filter, the resized image may be cached in different strategies and allow the actual image resizing to be performed only once.

```
var cacheFilter = function(config, handler, callback) {
  var cacheTable = { }

  var filteredHandler = function(args, inputStreamable, callback) {
    getCacheId(..., function(err, cacheId) {
      ...

      var cacheEntry = cacheTable[cacheId]
      if(cacheEntry) return callback(null, cacheEntry)
```

```
        handler(args, inputStreamable, function(err, resultStreamable) {
          ...

          cacheEntry[cacheId] = resultStreamable
          callback(null, resultStreamable)
        })
      })
    }
    callback(null, filteredHandler)
  }
```

The example code above shows an in-memory cache filter which cache result streamables returned from stream handlers inside a cache table.

4.5 Permission Control

Quiver filter is also useful for other kinds of separation of concerns, such as permission control and input escaping. Using the `quiver-filter` library, it is also possible to simplify the writing of stream filters that only modify incoming `args`.

The example code below shows a permission filter which ensure a user have sufficient privilege to access its inner stream handler. If insufficient permission is found, an error is then returned and access to the inner handler is skipped.

It should also be noted that this permission filter can depend on the user info filter defined earlier to provide a user's information and read from `args.userInfo` directly.

```
var filterLib = require('quiver-filter')

var permissionFilter = filterLib.argsFilter(
  function(args, callback) {
    var userInfo = args.userInfo

    checkPermission(userInfo, function(err) {
      if(err) return callback(err)

      callback(null, args)
    })
  })
```

4.6 Proxy

The proxy pattern replaces an original handler component with a proxy handler component that forward the request to a remote server using some external protocol. Figure 4.3

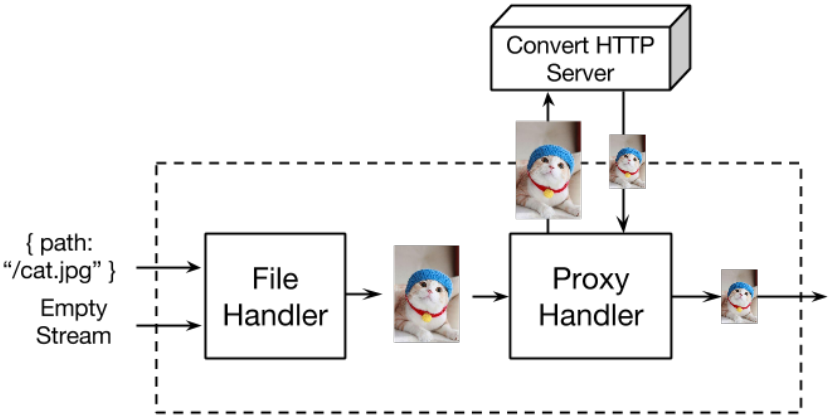

Figure 4.3: Proxy Handler

shows a variant of the image resize pipeline which forwards the image stream to a remote HTTP server for resizing. As image resizing is processor intensive, the proxy enable such operation to be handled behind a server farm to scale the handler.

Quiver currently allow component replacement through a feature called handler alias. It is also possible to rename the component inside the pipeline component definition to make use of the proxy handler directly.

4.7 Tee Pipeline

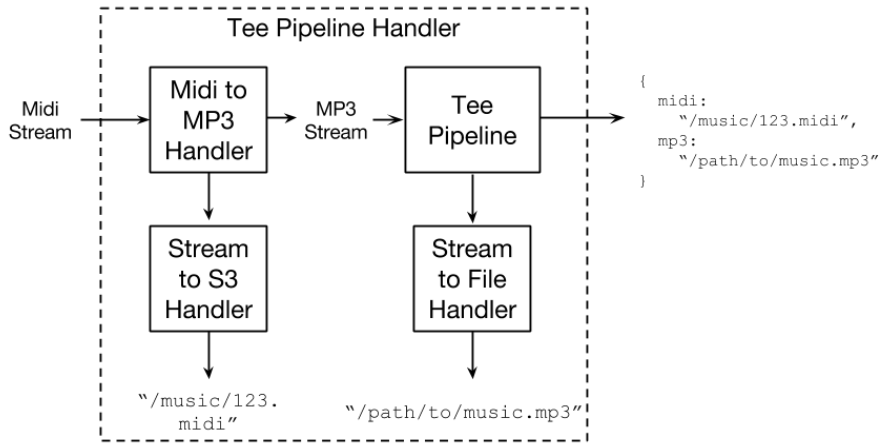

Figure 4.4: Tee Pipeline

A common query from the academics when it comes to stream processing is the Unix `tee` command. `tee` is a Unix tool which saves an incoming stream into file while forwarding it to the output stream at the same time. With that it resembles a T-shape pipeline which splits the stream processing into two paths.

In Quiver there is a pattern similar to the `tee` command and it is called the tee pipeline. The tee pipeline handler is actually a custom handler component which form a multi-staged tee pipeline based on provided handler component names. The tee pipeline make use of two handlers for each stage of the pipeline - one serializer handler and one stream transform handler. The serializer handler is responsible for transforming the input stream into a serialized ID, while the stream transform handler do the actual stream transformation. The serialized results are then combined by the tee pipeline handler and returned as a json result.

Figure 4.4 shows an example tee pipeline of a midi to mp3 transformaton. At the first stage the incoming midi stream is split into two paths, with one going to a serializer handler which stores the midi stream to Amazon S3. The second part of the midi stream goes to the actual midi to mp3 handler which produce an mp3 stream result. As that is the last stage of the tee pipeline, the pipeline handler sends the mp3 stream to another serializer handler which serialize the mp3 to local filesystem. Finally the tee pipeline returns a json result containing the local and remote path of the midi and mp3 content respectively.

Chapter 5

Performance Evaluation

Since Quiver runs on Node.js, it takes the same benefits and disadvantage of typical Node applications. Many performance benchmarks has been done in comparing the performance of Node against other platforms such as PHP. As this thesis is not about the performance benchmark of Node, instead this section will mostly focus on comparing the performance of Quiver against bare Node.

5.1 Node.js Performance

A quick mention that in general, Node has not been advertised as for running the fastest HTTP server. Instead the strength of Node has always been its scalability and high concurrency. Node mostly benefit from its asynchronous model, which minimize blocking in web applications and allow it to handle many concurrent requests.

5.2 JavaScript Closure Performance

One thing that differentiate Quiver from conventional Node applications is on its heavy use of JavaScript closures and the absence of object-oriented programming. This bring some implications to the performance of Quiver, as closure is not as well optimized in modern JavaScript engines as classical OOP.

Due to the popular usage of classical OOP by the mainstream, JavaScript engines have applied aggressive optimization strategies such as just-in-time (JIT) compilations to speed up method call of known classes. This works great for developers code in styles that are optimizable by the JIT compiler, but not so great for other programming styles including functional programming.

Other than that, JavaScript engines currently implement captured environment of closures using a chain of `Context` objects for each nested closures. Therefore access to captured variables become slower as the nesting level of closures increase. Quiver handlers

typically have at least two levels of closures - one for capturing the config during handler building time, and one for handling individual requests. (Egorov 2012)

Because of this, we do not expect Quiver to take full advantage of the optimizations available in current JavaScript engines. Instead our hope is to push the adoption of functional programming to the mainstream JavaScript community by demonstrating that it is possible to write complex applications without relying on OOP. We also believe that as the adoption of functional programming increase, JavaScript engines will eventually implement closure-specific optimizations in the future.

5.3 Benchmark Setup

Our performance benchmark is done using the ApacheBench (`ab`)[1] benchmarking tool. ApacheBench is originally designed to test the performance of Apache HTTP server, but its feature is generic useful enough to test any web server.

Our benchmark runs on the Intel Core i7-4500U CPU with 8GiB of RAM. The benchmark do not fully utilize the hardware and only run in single thread, mainly due to Node.js not having multithreading support. The benchmark perform stress test on the HTTP server with 20,000 total requests running 1,000 concurrent requests at a time.

5.4 Hello World Benchmark

On average the performance for Quiver is about 2~3 times slower than bare Node. This is to be expected since Quiver runs on Node and thus cannot be faster than Node itself. As applications become more complex, the performance gap is expected to decrease since performance bottlenecks usually occured application code rather than the libraries used.

Type	Mean	SD	Median
Bare Node	47	187	12
Node Stream	62	222	14
Node Handler	76	239	18
Express Handler	110	360	19
Http Handler	125	468	24
Simple Handler	182	601	29

Table 5.1: Hello World Benchmark in ms

[1] ApacheBench website: http://httpd.apache.org/docs/2.4/programs/ab.html

Table 5.1 shows the performance benchmark of a simple hello world server implemented in different ways. The tested server code contains minimal logic and just return a simple HTTP response containing the text "hello world" in the response body.

- *Bare Node* - This server runs bare Node.js code with no external dependencies. This is expected to be the fastest and is used as baseline for other measurements.
- *Node Stream* - This server adds only the Quiver stream construct to the server and involve converting Node streams to Quiver streams. The result shows that it is not much slower than bare Node. Therefore it can be concluded that the performance of Quiver streams alone is fast enough.
- *Node Handler* - This server implements the HTTP handler in the Quiver http handler signature, and then convert it into a Node handler. The performance is only slightly slower than the Node Stream benchmark. This shows that the Quiver http handler construct do not introduce much performance impact.
- *Express Handler* - This server uses the Express framework[2] to handle requests. Result shows that Express at its minimal is about 2x slower than bare Node. Unlike Quiver, it is not possible to break down Express into smaller components to test with. This also show that if performance is important, one can choose to use only certain base layers of Quiver and achieve performance faster than monolithic frameworks like Express.
- *Http Handler* - This server runs the full Quiver component system loaded with a simple http handler component. Unlike the earlier Node Handler benchmark, the server runs from the `quiver-server` command line tool and the http component is declared instead of being used directly. This shows that the performance of the full Quiver system is roughly the same speed as Express, at about 2x slower than bare Node.
- *Simple Handler* - This server is similar to the Http Handler benchmark, except that it is running a component of type simple handler. Simple handler is a specialized stream handler which implicitly convert streams into specific format such as text and json. There are two additional layers in simple handler - one forwarding http request to stream handler, and one converting stream to simple formats. Therefore the simple handler benchmark shows that it is quite a bit slower than http handler, with speed of 3.5x slower than bare Node. The slow down of simple handler shows that there are still some performance bottleneck in converting streamable to internal representation.

[2]Express is a popular web framework for Node.js. Website: http://expressjs.com/

5.5 Demo App Benchmark

Type	Mean	SD	Median
Node Handler	149	478	25
Express Handler	189	430	59
Express with Middleware	200	431	88
Quiver Handler	450	830	76

Table 5.2: Demo App Benchmark in ms

Other than simple hello world benchmark, a simple demo application is also written with more complex functionalities. In this demo application, the HTTP server gives custom greetings to the user based on their user id. The server also accepts user submitted content via HTTP POST, perform HTML escape on it, and finally echo the escaped input back in the response.

The demo Quiver application consist of a graph of 10 quiver components. On the other hand a corresponding Node server is written with all same functionalities implemented in just one monolithic function. With the additional complexity the benchmark shows that the Quiver application runs about 3x slower than the monolithic Node server.

Table 5.2 also shows the benchmark of the same Node application runs under Express, which is just slightly slower than bare Node. We also try another customized Express server with the code refactored into three Express middlewares. The result is slightly slower than the Express server with single handler function. With that Quiver is shown to be about 2.5x slower than Express due to the heavy use of many small components.

5.6 Benchmark with Database Access

Type	Mean	SD	Median
Node Handler	177	412	103
Express Handler	243	643	132
Quiver Handler	474	988	208

Table 5.3: Demo App With MongoDB, Benchmark in ms

The previous benchmark stress tested the performance of Quiver without external factor. However real world applications tend to be much more complicated, as otherwise there will be no need to use Quiver to manage the complexities. The benchmark at 5.5 used an in memory database called NeDB to provide mock database access. To get a more practical performance analysis, we replace that with a remote MongoDB server and repeated the benchmark.

The new benchmark result at table 5.3 shows that the additional database call introduce some linear delay to the performance of Node and Express handler. On the other hand the additional delay for Quiver handler is relatively small as compared to the previous benchmark. This shows that Quiver do not introduce as much performance penalty as the complexity of components increase.

5.7 Benchmark with setImmediate

Type	Mean	SD	Median
Node with setImmediate	578	352	587
Express with setImmediate	601	406	615
Quiver with setImmediate	1081	417	1100

Table 5.4: Demo App With MongoDB and setImmediate, Benchmark in ms

Lastly to demonstrate some quirks in Node, we repeat benchmark 5.6 with an additional call to `setImmediate()`. `setImmediate()` is a Node built in function that schedules an async callback for immediate execution after other I/O events. As setImmediate does some internal checking to prevent infinite recursion, it also introduce some performance quirks and significantly impact benchmark results.

Table 5.4 shows that with just one call to setImmediate would slow down the performance of all benchmarked applications by more than a factor of two. The main objective for this benchmark is to demonstrate how just one slow function call may significantly impact the performance of libraries and applications.

In fact, we discovered the performance penalty of `setImmediate()` in some earlier benchmarks with Quiver. By replacing the calls with `process.nextTick()`, a similar builtin Node function with better performance, we were able to significantly improce the performance of Quiver. We believe that there are more performance quirks in Node and V8 that are hidden within Quiver libraries, and finding those bottlenecks might produce radically different performance for Quiver in future.

5.8 Conclusion

The benchmarks show that in simple applications, Quiver currently runs about 2~3x slower than conventional approaches. Nevertheless we do not see any significant red flags that impacted by the architecture design of Quiver. As there is not much focus on optimization in the current Quiver code base, we are optimistic that the performance of Quiver will improve as optimizations be made in the future.

It is also shown that Quiver applications slow down as the number of quiver components used increases. This is to be expected since each component introduce additional layer of indirection. However in real world applications the performance of such complexity is expected to be significantly reduced. This is because real world quiver components would typically have much more complex implementation, which would bring more performance impact than the Quiver libraries itself. As demonstrated in benchmark 5.7, even seemingly simple operations like remote database access or call to `setImmediate()` can have much more significant impact on application performance than Quiver itself.

Quiver is designed to be best suited in complex applications where the overhead of the libraries become negligible. By breaking down complex applications into small and reusable components, Quiver makes it much easier for developers to perform application optization by modifying the component graph. As a result the benefits of adding new quiver components far outweight the performance gained by implementing everything in one monolithic component.

Chapter 6

Current Progress

6.1 Core Libraries

There are currently 45 quiver libraries consist of over 8,000 lines of code with 5,000 lines of test cases that have been developed to lay out the foundation architecture for Quiver.

6.1.1 Utilities

- *quiver-copy* - copy simple JavaScript objects that do not have non-trivial prototype.
- *quiver-merge* - Merge keys from multiple plain objects into one object.
- *quiver-error* - Provide convenient syntax for creating async errors.
- *quiver-callstack* - Simple way of capturing and printing callstacks.
- *quiver-safe-callback* - Ensure that callback is called only once and never synchronously.

6.1.2 Foundation

- *quiver-stream-channel* - Create quiver read/write stream pairs to transfer data.
- *quiver-stream-convert* - Convert stream or streamable into various objects such as text and json.
- *quiver-pipe-stream* - Pipe content from read stream to write stream.
- *quiver-split-stream* - Split a Quiver read stream to be read by multiple readers.
- *quiver-node-stream* - Convert native Node.js streams to Quiver streams.
- *quiver-node-handler* - Create Node.js request handler from Quiver.js HTTP handler
- *quiver-subrequest* - Perform client HTTP subrequest.

6.1.3 Constructs

- *quiver-config* - Retrieve quiver-related config from config object safely.
- *quiver-filter* - Convert custom filters into standard stream filter.
- *quiver-middleware* - Manage dependencies by creating various middlewares.
- *quiver-handleable* - Conversion functions between handlers and handleables.
- *quiver-simple-handler* - Convert simple handler to standard stream handler.
- *quiver-router* - Router stream handler that dispatch requests to different handlers based on supplied path.

6.1.4 Component System

- *quiver-component* - Component system to allow creation of complex interconnected components.
- *quiver-module* - Module system for exporting quiver components and managing package dependencies.
- *quiver-command* - Command-line wrapper for stream handler components.
- *quiver-server* - HTTP server wrapper for quiver components.

6.1.5 Component

- *quiver-file-component* - Create various filesystem-related stream handlers and filters.
- *quiver-command-component* - Quiver components that wrap around command line tools.
- *quiver-cache-component* - Filter components to cache the result of handlers.
- *quiver-template-component* - Generic template component to convert input using any template compiler.

6.2 ScoreCloud

Quiver has been used in DoReMIR, the company I am working at, to implement the server backend for our product ScoreCloud. At this moment there are over 130 quiver components been created that consist of over 4,000 lines of code.

The implementation of ScoreCloud server using Quiver has provided a lot of first-hand feedback on the things that are working well as well as things that still require improvement. We find new needs to implement new features in Quiver to make the server code more elegant. Apart from some boilerplates and workaround, implementing new quiver components do proved to be easy and highly modular.

I also find a few challenges along the way on problems that Quiver has yet to solve. In particular it is hard to verify if the component definitions adhere to application requirements. I found that such problem cannot simply be solved through unit testing, and new

methodology is required to efficiently audit and verify the correctness of quiver component definitions.

6.3 Future Work

Despite all the work done, Quiver is currently still in its early stage of development. It was only recently that we finished the foundation quiver libraries and started developing standard quiver components, particularly the file and command components. More standard quiver components is needed before Quiver can become simple enough to be used by beginners.

Our plan is to slowly build more quiver components, such as a full HTTP stack, and also accept community-contributed components. Other than that, there are many more design patterns that are currently still under development, such as a general caching pattern.

We will also put much effort in building protocols on top of streamable and handleable. There are also work to be done for better integration between stream handler and http handler components. Lastly, a new routing system is currently being developed to dispatch requests to different stream or http components.

Chapter 7

Conclusion

In this thesis we have shown the architecture of Quiver and how it enable modular applications to be developed. We went into each architecture layers of Quiver and discussed its various constructs, from stream to handler to middleware. Finally we shown several common design patterns in Quiver to demonstrate how real world Quiver application is written.

The Quiver architecture lays the foundation for new kind of programming paradigm in web development. With that we hope to influence the mainstream web developer community about alternatives beyond object-oriented programming and model-view-controller (MVC) paradigm.

While functional programming has existed for a long time, it is only until recently being discussed more widely in web development literatures. Nevertheless due to the lack of practical examples, in the mainstream developer community there has always been doubt on the possibility of writing applications without using OOP. With Quiver we aim to push further the functional programming movement by demonstrating it with practical applications.

In the modern web architecture landscape dominated by monolithic frameworks using MVC paradigm, we also hope to break through the barrier and offer a solution that might look radical to most people. One thing certain is that some paradigms used in Quiver have already been actively discussed by many people for some time. However despite the active discussion, such paradigms are largely implemented in adhoc manners by more experienced developers. With Quiver we hope to bring together all the best programming paradigms together and present them in a consistent way.

In conclusion, we hope that Quiver is just the beginning of our journey to build a new generation of web applications.

References

Egorov, Vyacheslav. 2012. "Grokking V8 Closures for Fun (And Profit?)." http://mrale.ph/blog/2012/09/23/grokking-v8-closures-for-fun.html.

Fielding, Roy T., and Richard N. Taylor. 2002. "Principled Design of the Modern Web Architecture." *ACM Transactions on Internet Technology* 2 (2) (May): 115–150. doi:10.1145/514183.514185. http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf.

Fielding, Roy Thomas. 2000. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis, University of California. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Fowler, Martin, and James Lewis. 2014. "Microservices." http://martinfowler.com/articles/microservices.html.

Hughes, James. 2013. "Micro Service Architecture." http://yobriefca.se/blog/2013/04/29/micro-service-architecture/.

Morrison, J. Paul. 2010. "Flow Based Programming Website." http://www.jpaulmorrison.com/fbp/.

Raymond, Eric Steven. 2004. "The Art of Unix Programming." Boston: Addison-Wesley. http://www.faqs.org/docs/artu/.

Sparks, Michael. 2005. "Kamelia: Highly Concurrent and Network Systems Tamed." White Paper. http://www.bbc.co.uk/rd/publications/whitepaper113.

Appendix A

Architecture summary

The following figure presents a collection of all essential figures that has been introduced in this chapter:

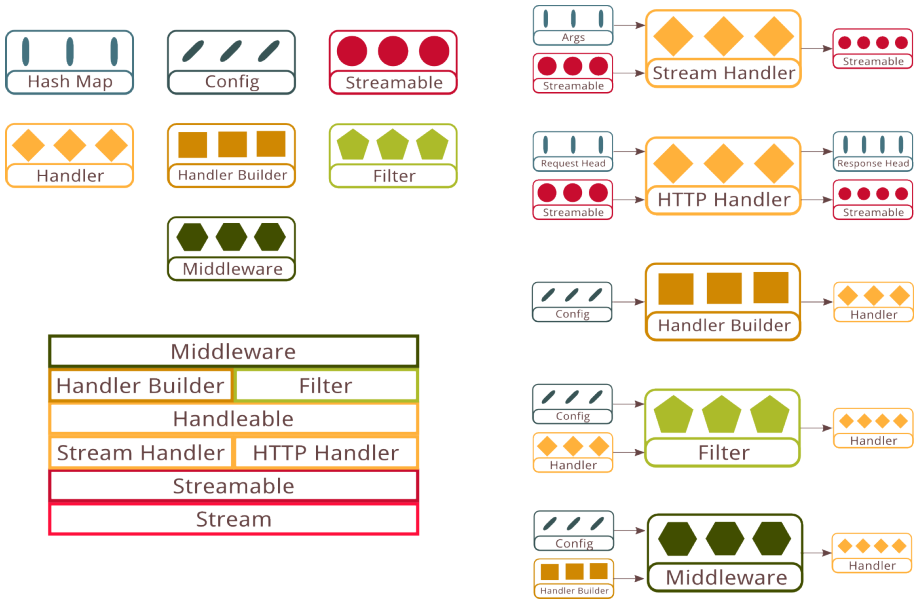

Figure A.1: Architecture Summary

Appendix B

References

Egorov, Vyacheslav. 2012. "Grokking V8 Closures for Fun (And Profit?)." http://mrale.ph/blog/2012/09/23/grokking-v8-closures-for-fun.html.

Fielding, Roy T., and Richard N. Taylor. 2002. "Principled Design of the Modern Web Architecture." *ACM Transactions on Internet Technology* 2 (2) (May): 115–150. doi:10.1145/514183.514185. http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf.

Fielding, Roy Thomas. 2000. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis, University of California. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Fowler, Martin, and James Lewis. 2014. "Microservices." http://martinfowler.com/articles/microservices.html.

Hughes, James. 2013. "Micro Service Architecture." http://yobriefca.se/blog/2013/04/29/micro-service-architecture/.

Morrison, J. Paul. 2010. "Flow Based Programming Website." http://www.jpaulmorrison.com/fbp/.

Raymond, Eric Steven. 2004. "The Art of Unix Programming." Boston: Addison-Wesley. http://www.faqs.org/docs/artu/.

Sparks, Michael. 2005. "Kamelia: Highly Concurrent and Network Systems Tamed." White Paper. http://www.bbc.co.uk/rd/publications/whitepaper113.