

Implementation and Analysis of a Web
Request Pipelining Framework
For a degree of Master of Science in Computer Science
at Southern Connecticut State University

Sean P. Kenny
Thesis Advisor: Dr. Hrvoje Podnar

October 2012

*To my daughters, Paige and Jocelyn - for inspiration I will
cherish always.*

*To my wife, Kim - who's selfless love, support, and
encouragement has allowed me to fulfill new dreams and goals I
never would have been able to imagine without her.*

Contents

1	Introduction	6
2	Implementation and Analysis Tools	11
2.1	Git	11
2.2	PHP	11
2.3	phpDocumentor	12
2.4	Apache	12
2.5	FireFox	13
2.6	CodeIgniter	13
2.7	Selenium	14
3	Test Applications	15
3.1	Static Sample App	15
3.2	Basic Pipeline	17
3.2.1	MVC Pipeline (PVC)	17
4	Architectural Components	19
4.1	Server components	19
4.2	Client components	20
5	Pipelets	22
5.1	The Root Pipelet	22
5.2	Nested Pipelets	25
5.3	Pipelet Priority	26
6	Class Interfaces	28
6.1	Output	29
6.2	Framework Adapter	31

7	Design Patterns	32
7.1	Strategy Pattern	32
7.2	Factory Pattern	33
8	Sequence Diagrams	35
8.1	Server	35
8.2	Client	36
9	Data Objects	40
9.1	Transmitted Data	40
10	Things to Consider	42
10.1	PHP Output Buffering	42
11	Analysis	44
11.1	Data Collection Methodology	44
11.2	Simulating Load	47
11.3	Results	48
12	Next Steps	50
12.1	Expanded Output API	50
12.2	Minification	51
12.3	Addition of framework adapters	51
12.4	Language agnostic Apache server extension	51
13	Code Listings	52

List of Figures

1.1	Layers of the Open Systems Interconnection model (OSI model).	8
1.2	Comparison of a traditional HTTP web request with the operation of an HTTP web request Pipeline. A pipeline approach will deliver visible content sooner, as all parts of the document are delivered in parallel.	10
3.1	Feature rich OpenPipe sample application to illustrate nested pipelines	16
3.2	Basic OpenPipe sample application to illustrate a basic pipelines	17
3.3	OpenPipe adapter setup for CodeIgniter	18
4.1	Server side tecnology stack with OpenPipe components	20
4.2	Client side tecnology stack with OpenPipe components	21
5.1	Sample pipelet containing HTML, CSS, and JavaScript	22
5.2	Root pipelet HTML	23
5.3	Root pipelet layout	24
5.4	Nested Pipelets with a depth of 3	26
5.5	Sample pipelet containing HTML, CSS, and JavaScript	27
6.1	A generalization of the OpenPipe output object	29
6.2	A generalization of the OpenPipe adapter object	31
7.1	The strategy pattern utilized by OpenPipe for the main OpenPipe runner object	33
7.2	Instantiation and running of an OpenPipe_Runner object	33
7.3	The factory pattern utilized by OpenPipe	34
8.1	OpenPipe Runner sequence diagram	36
8.2	OpenPipe output sequence diagram	38

8.3	OpenPipe client side pipelet load calls	39
9.1	The client segment data object	41
10.1	PHP function that helps bypass PHP output buffering that blocks the HTTP pipelining of data to the client browser . . .	43
11.1	A selenium script that retrieves performance and timing data from websites	45
11.2	DOM performance timing data made available via JavaScript [1]	46
11.3	DOM performance timing data show as linear request [1] . . .	47
11.4	Calculated response and load time in milliseconds. Data is based on timing data collected from automated browser runs via Selenium	49
12.1	An explicit version of an OpenPipe output API	50

Chapter 1

Introduction

Modern web sites and web applications provide valuable information and services to an ever growing web audience. One of the most important requirements of web sites and web applications trying to deliver this important content is speed. Increased speed of a website can often determine an end user's perception of overall quality and value of a web-based service. Fast and responsive sites will be more likely to achieve higher monthly page views, adoption rates, and overall success

If speed is recognized as a critical factor for today's web then the current HTTP protocol used to deliver this web content should be examined as the source of a possible bottleneck when trying to deliver it. Web content has become increasingly more dynamic and interactive over the last ten years, and this current standard for web content delivery can be tailored for meeting the current pressures of the modern web.

Since the introduction of HTTP, web developers have built an ever-growing repertoire of tips and tricks to squeeze the most out of the HTTP-based web. These methods include, but are not limited to the following:

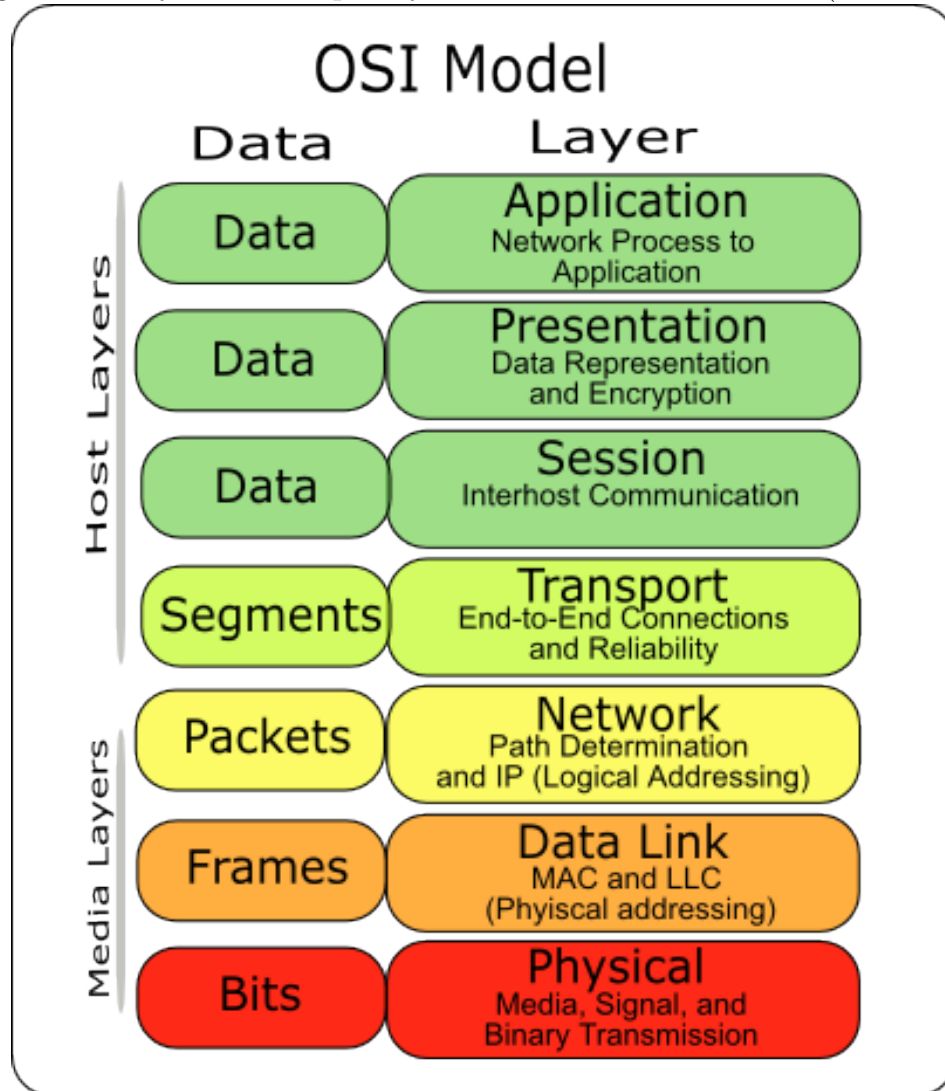
- Use a content delivery network
- Add an Expires header (HTTP 1.1)
- Gzip components
- Put CSS at the top of the page
- Avoid CSS expressions

- Reduce DNS lookups
- Minify JS
- Avoid redirects
- Remove duplicate scripts
- Turn off ETags
- Make AJAX cacheable and small

Today's web sites and web applications employ many of these techniques to help deliver content to end-users in an efficient manner. HTTP Pipelining is another technique that could be employed alongside any of these methods to increase web performance.

The HTTP protocol is part of the application layer of the OSI model [see figure 1.1]. When an HTTP request is issued from a web browser for an initial HTML document, related elements such as images, JavaScript, and Cascading Style Sheet (CSS) are subsequently retrieved using additional HTTP requests. Therefore, the rendering of a single web page could involve many HTTP web requests. An HTTP pipeline is presented as a thin layer of application logic on top of HTTP which attempts to optimize the request cycle in a manner that allow pieces of a full web page to load and display independently. This added layer is developed as a software library containing both server-side and client-side application code.

Figure 1.1: Layers of the Open Systems Interconnection model (OSI model).

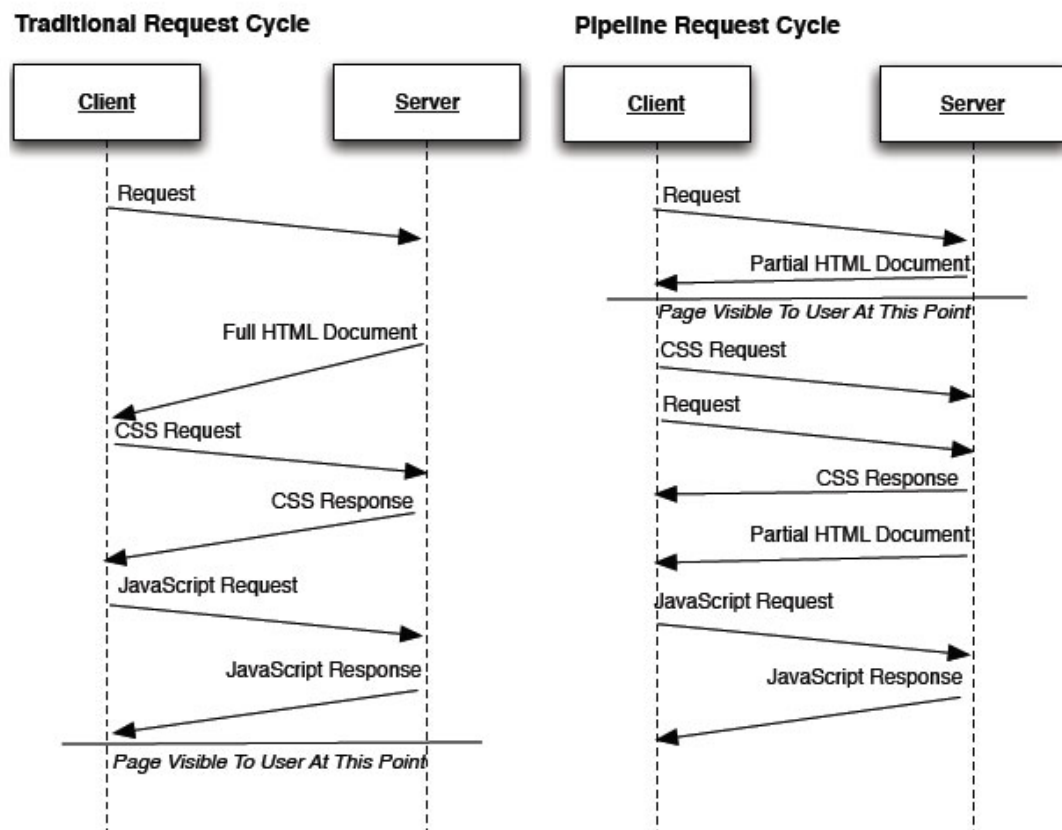


The resulting framework processes, requests, and delivers responses in an optimized manner taking into account current HTTP limitations that are inherent to the overhead created by the connect and request style of communication. The result of this response optimization through a pipeline is an increase in perceived speed that is accomplished by displaying fully functional content as quickly as possible – even before the entire document is completely processed by the web server.

HTTP pipelining is inspired from traditional pipelining technologies utilized by today's modern CPU's, where an instruction's life cycle is broken into multiple stages. Instead of instructions, HTTP pipelining breaks the page generation process into several stages [see figure ??] which include [SOURCE-FB?]:

1. Request parsing: web server parses and sanity checks the HTTP request.
2. Data fetching: web server fetches data from storage tier.
3. Markup generation: web server generates HTML markup for the response.
4. Network transport: the response is transferred from web server to browser.
5. CSS downloading: browser downloads CSS required by the page.
6. DOM tree construction and CSS styling: browser constructs DOM tree of the document, and then applies CSS rules on it.
7. JavaScript downloading: browser downloads JavaScript resources referenced by the page.
8. JavaScript execution: browser executes JavaScript code of the page.

Figure 1.2: Comparison of a traditional HTTP web request with the operation of an HTTP web request Pipeline. A pipeline approach will deliver visible content sooner, as all parts of the document are delivered in parallel.



Chapter 2

Implementation and Analysis Tools

2.1 Git



The OpenPipe project makes use of Git to store the current codebase, examples, documentation, and research data. Git is a modern distributed revision control and source code management tool. Through utilizing Git, developers are free to checkout, and even branch the existing code base to meet current and future needs.

Git (a command line tool) is available with many Linux distributions, and is freely available to download and compile. Pre-built binary installers, and GUI interfaces exists for all major operating systems. The SSH URL for accessing this project via git is `git@github.com:polycoder/OpenPipe.git`. All fork requests can be placed by visiting `HTTPs://github.com/polycoder/OpenPipe`.

2.2 PHP



The core server components of OpenPipe were all written with PHP 5.3. PHP is a general-purpose server-side scripting language originally designed for web development to produce dynamic web pages.

PHP can be deployed on most Web servers and as a standalone shell on almost every operating system and platform free of charge. PHP is installed on more than 20 million Web sites and 1 million Web servers [SOURCE]. These statistics make PHP a very good option to build and develop new web technologies and frameworks which will then be available to a large community of software developers.

2.3 phpDocumentor



OpenPipe uses phpDocumentor to generate all of the available PHP class documentation. phpDocumentor is a tool with which it is possible to generate documentation from your PHP source code using a standardized set of source code commenting conventions. With this documentation you can provide developers with more information regarding the functionality embedded within your source code. phpDocumentor is heavily inspired by the Javadoc tool available with the Java SDK.

2.4 Apache



OpenPipe has been built and tested using the Apache HTTP server. OpenPipe is not limited to running on this architecture, and can theoretically be run on any web server that supplies integration with PHP.

Apache is a web server software notable for playing a key role in the initial growth of the World Wide Web. Apache is developed and maintained by an open community of developers under the Apache Software Foundation.

Apache is available with many Linux distributions, and is freely available to download and compile. Prebuilt binary installers exist for all major operating systems.

Since April 1996 Apache has been the most popular HTTP server software in use. As of March 2012 Apache was estimated to serve 57.46% of all active websites and 65.24% of the top servers across all domains [SOURCE]

2.5 FireFox



All client side testing and analysis (manual and automated) of OpenPipe was performed using the FireFox web browser. FireFox supplies a very advanced toolset for profiling HTTP requests, viewing and editing HTML, and debugging JavaScript. FireFox is freely available to download and compile. Prebuilt binary installers exist for all major operating systems.

2.6 CodeIgniter



The OpenPipe framework that has currently been developed provides an adapter that interfaces with the CodeIgniter framework. Using this adapter it is fairly straight forward and simple to convert an existing CodeIgniter application to take advantage of OpenPipe HTTP request pipelining. CodeIgniter is a powerful PHP framework with a very small footprint, built for PHP

coders who need a simple and elegant toolkit to create full-featured web applications.

2.7 Selenium



Selenium automates browsers by providing a common API that is provided in the form of a, 'WebDriver'. WebDrivers exist for every major browser including:

1. Firefox
2. Chrome
3. Safari
4. Internet Explorer
5. Android
6. iOS

Selenium is primarily used for the automated testing of web applications, and is often used in conjunction with a unit testing framework. Selenium is however not limited to this set of tasks and can be extremely useful for other tasks such as performance testing and analysis. OpenPipe utilized Selenium to help clarify the performance gains and penalties when using the framework under different usage scenarios and server load. Utilizing Selenium it became trivial to automate web page loading, and record large datasets of performance data.

Chapter 3

Test Applications

3.1 Static Sample App

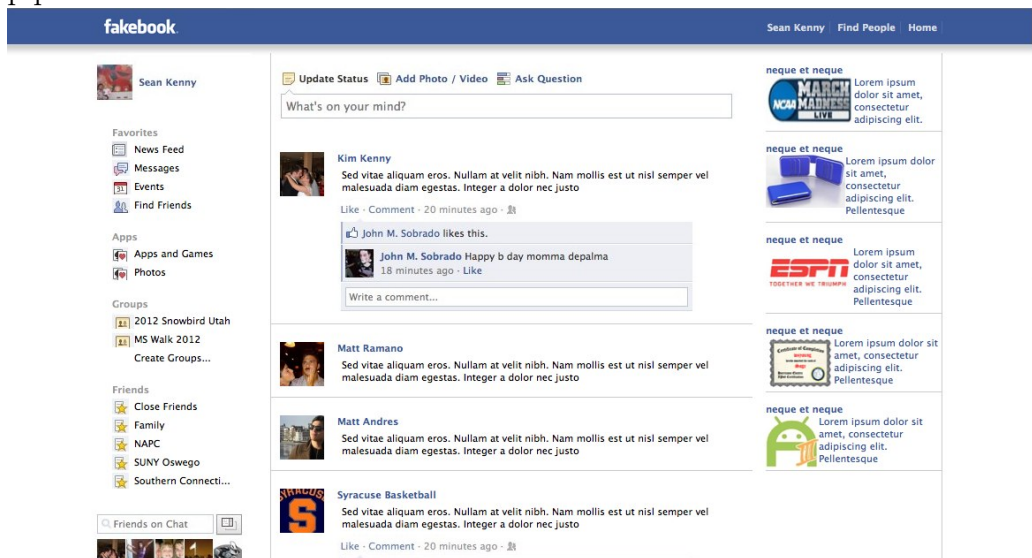
Before development on the server and client components was started a static web site was built using plain HTML and CSS [see figure 3.1] . This static site's main purpose is to provide a foundation for testing, while adding a clean visual experience that adequately illustrates the optimization of a pipelined HTML page. The main requirement for this static site was that it be composed of many individual components and page sections. OpenPipe is geared towards pages that load information that has many cross cutting concerns per request. The static sample application meets this requirement, and has various components that fall into the following defined sections:

1. Header
 - (a) Navigation - main navigational links available to the user.
2. Main Content Area
 - (a) Post Input - an input box used for submitting posts of various types.
 - (b) Posts list - a listing of main posts that the user has received.
 - (c) Post comments - each post contains a potential list of comments that have recorded.
3. Left Sidebar

- (a) Favorite - a sidebar item containing the areas most accessed by a user.
- (b) Apps - application currently installed by the user.
- (c) Groups - groups a user is a member of.
- (d) Friends - groupings of friends the user is related to.
- (e) Friends Search - A search input box for finding friends.
- (f) Friends Face-box - A graphical view of friends through their profile pictures.

4. Right Sidebar

Figure 3.1: Feature rich OpenPipe sample application to illustrate nested pipelines

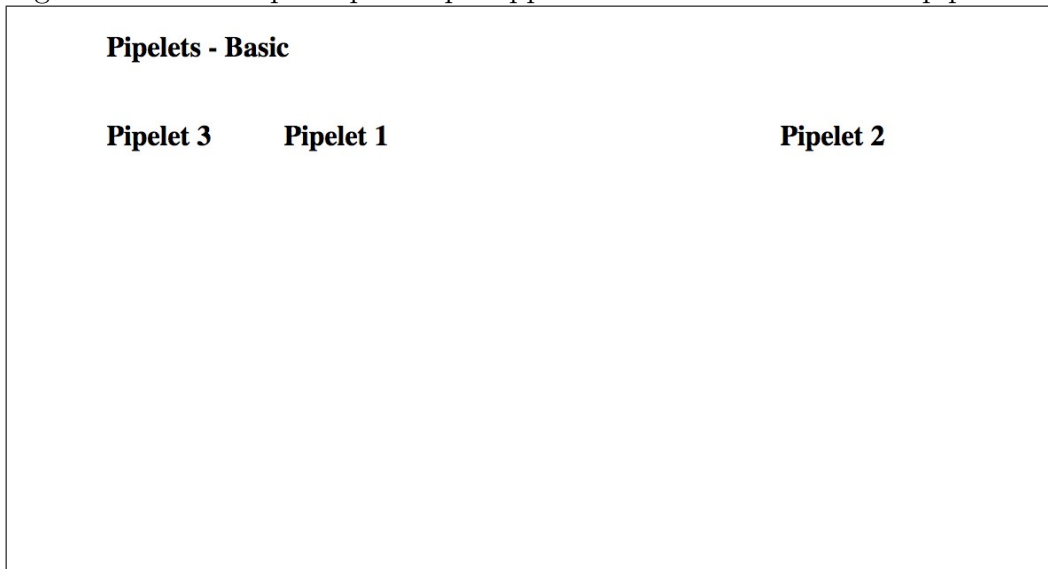


The static application contains a diverse amount of sections. The resulting HTML, CSS, and JavaScript code created during this page creation was migrated to an MVC Based system that implements the OpenPipe Adapter interface.

3.2 Basic Pipeline

During the initial prototyping phase for OpenPipe a basic pipelined base site was built to illustrate the core components that are applied during the main pipeline process. The basic pipeline example is composed of one default layout, and three separate page pipelets [see figure 3.2].

Figure 3.2: Basic OpenPipe sample application to illustrate a basic pipelines



This simple design did not integrate with any existing PHP frameworks and relied on a very simple paradigm available in the PHP language called includes. Each pipelet being loaded is placed in an include file within a defined folder specified at runtime. The root Pipelet layout is placed within the layout.php include file, which is also specified at runtime. This simple and effective OpenPipe application was used to verify and test all the components of the OpenPipe system including the client side JavaScript library.

3.2.1 MVC Pipeline (PVC)

The final stage in the development cycle was to implement an adapter for an existing PHP MVC framework. This adapter essentially retrofits the existing framework, and allows it to take advantage of HTTP pipelining provided by the OpenPipe framework.

The PHP MVC framework chosen to create an adapter for is named CodeIgniter. CodeIgniter is a lightweight open-source MVC framework, that is relatively simple to work with and extend.

Utilizing CodeIgniter's same system of controllers and views, a developer can easily convert any CodeIgniter page to an HTTP pipelined request. This is made possible by a simple series of installation steps illustrated in the provided sample application. The main index file ends up looking like figure 3.3, once the OpenPipe integration has taken place.

Figure 3.3: OpenPipe adapter setup for CodeIgniter

```
<?php
require_once(dirname(__FILE__).'../../../../../server/php/OpenPipe
    /Adapter/Pvc/CodeIgniter.php');
require_once(dirname(__FILE__).'../../../../../server/php/OpenPipe
    /Output/Piped.php');
require_once(dirname(__FILE__).'../../../../../server/php/OpenPipe
    /Output/Standard.php');
require_once(dirname(__FILE__).'../../../../../server/php/OpenPipe
    /Runner.php');

$openPipeAdapter = new OpenPipe_Adapter_Pvc_CodeIgniter(
    dirname(__FILE__));

if(isset($_GET['nopipe'])){
    $openPipeOutput = new OpenPipe_Output_Standard();
}else{
    $openPipeOutput = new OpenPipe_Output_Piped('../../../../../
        client/js');
}
```

Chapter 4

Architectural Components

4.1 Server components

The server is composed of four main layers:

1. **PHP** - The underlying scripting language. Run as a component of the web server.
2. **OpenPipe_Runner** - The main loop of an OpenPipe application. The runner orchestrates actions between the adapter being utilized, and the output that is generated.

```
//starting an openpipe runner
$openPipeRunner = new OpenPipe_Runner($openPipeAdapter,
    $openPipeOutput); $openPipeRunner->run(); //outputs
openpipe content
```

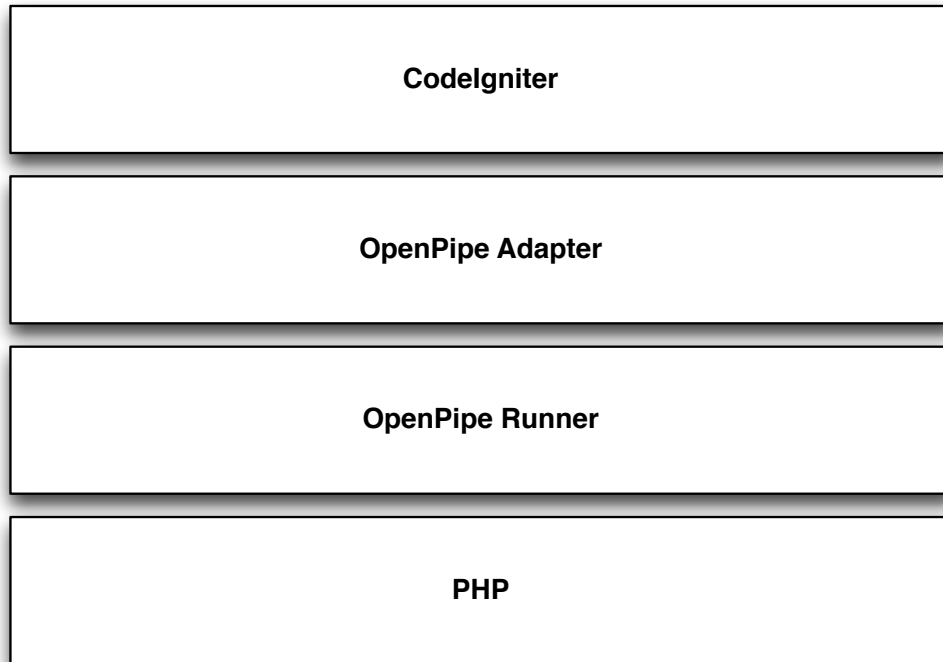
3. **OpenPipe_Adapter** - A pluggable interface which retrieves and returns pipelets components from the underlying web framework. An adapter can be used to interface pre-existing PHP application frameworks with OpenPipe, or whole new OpenPipe-centric application frameworks.

```
//loading an openpipe adapter
$openPipeAdapter = OpenPipe_Adapter_Pvc_CodeIgniter(
    dirname(__FILE__));
```

4. **Framework** - The framework being utilized with the OpenPipe adapter. The framework normally provides core web application components

such as database libraries, request routing, session handling, and form validation. The CodeIgniter MVC system is an example of framework.

Figure 4.1: Server side technology stack with OpenPipe components



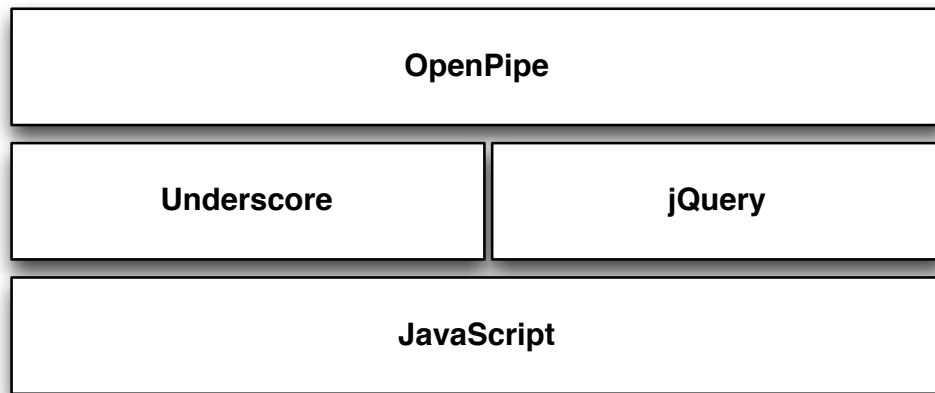
4.2 Client components

The client is composed of three main layers:

1. **JavaScript** - The host environment (web browser) and core JavaScript libraries.
2. **Vendor Libraries** - OpenPipe makes use of two very popular, reliable, and lightweight cross-browser JavaScript frameworks.
 - (a) **Underscore.js** - A utility-belt library for JavaScript that provides functional programming support.

- (b) **jQuery** - Provides simple and elegant client side scripting and manipulation of HTML DOM.
- 3. **OpenPipe** - A client side library which is responsible for receiving events from an OpenPipe based server. These events are processed and associated data for these events is loaded into the HTML DOM as HTML, CSS, and JavaScript.

Figure 4.2: Client side tecnology stack with OpenPipe components



Chapter 5

Pipelets

Every OpenPipe HTTP request cycle is composed of pipelets. A pipelets represents an atomic composition of HTML, CSS, and JavaScript [see figure 5.1] . A web page request can be composed of one to many pipelets.

Figure 5.1: Sample pipelet containing HTML, CSS, and JavaScript

```
<!-- a simple pipelet containing css, javascript, and html
-->
<div pipelet-id="pipelet-1" ></div>
  <link rel="stylesheet" type="text/css" href="css/pipelet-1.
    css" />
  <script type="text/javascript" src="js/pipelet-1.js" ></
    script>
  <h1>Hello world!</h1>
</div>
```

5.1 The Root Pipelet

Every pipelined HTTP request contains at least one initial pipelet. This initial pipelet is known as the root pipelet, and is the source for extraction and retrieval for all other pipelets. The root pipelet is special because it defines the overall layout of the page, from which all pipelets will be loaded and placed into [see figure 5.2 and 5.3].

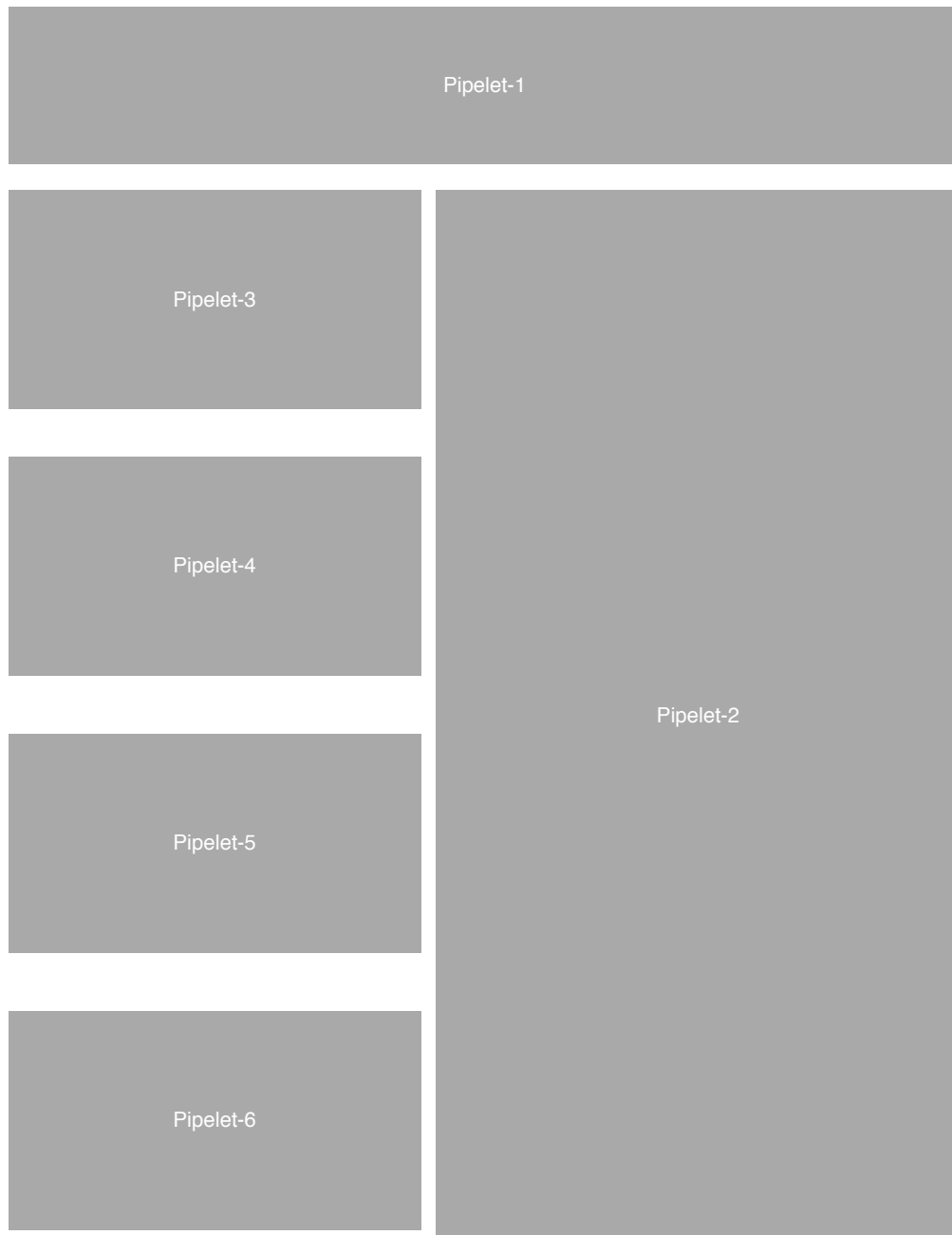
The root pipelet contains the root `<html />` element and its immediate children - `<head />` and `<body />` . Since the root pipelet defines the

head section it is capable of setting extra page meta information through specialized head tags, and linking or including CSS and JavaScript shared between pipelets.

Figure 5.2: Root pipelet HTML

```
<html>
<head>
<title>Root Pipelet!</title>
<link rel="stylesheet" type="text/css" href="css/global.css"
/>
<script type="text/javascript" src="js/app.js" ></script>
</head>
<body>
<h1>Hello World!</h1>
<div pipelet-id="pipelet-1"></div>
<div pipelet-id="pipelet-2"></div>
</body>
</html>
```


Figure 5.3: Root pipelet layout

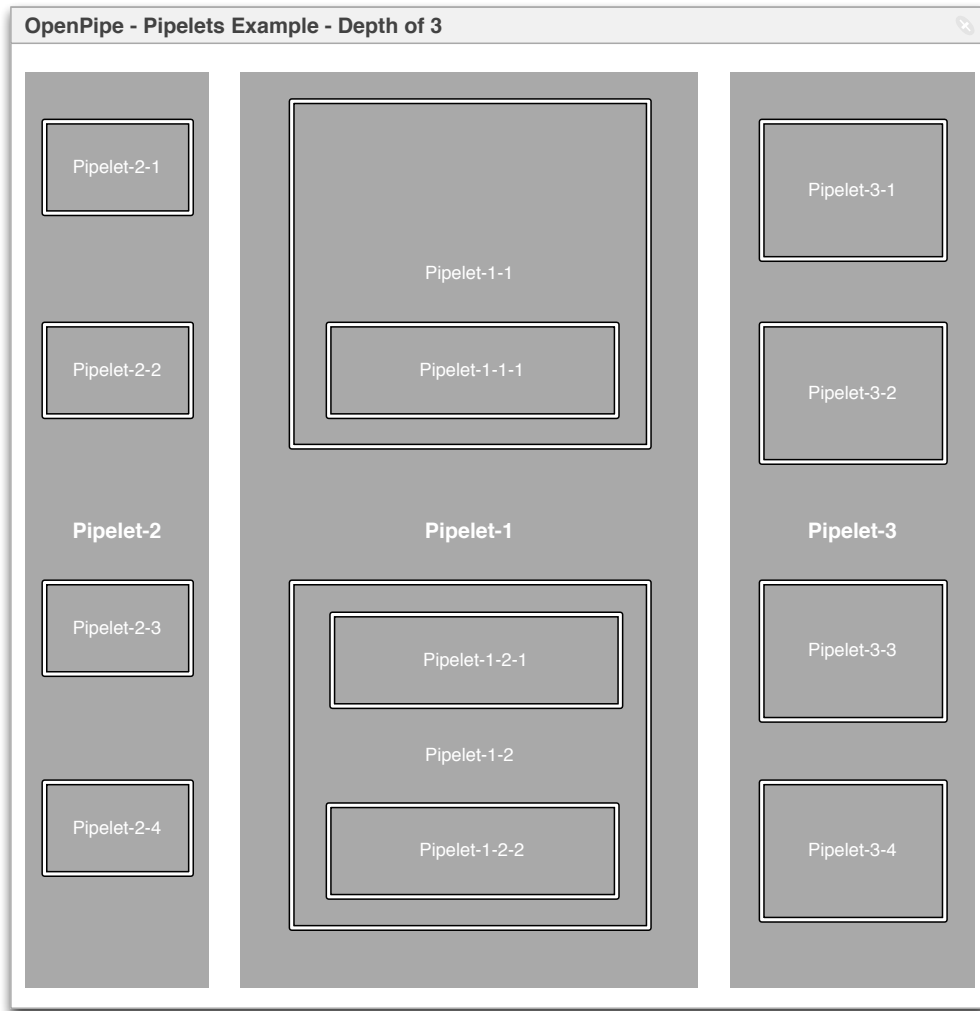


5.2 Nested Pipelets

Pipelets can be nested within other pipelets. This provides a mechanism to display content that contains n -levels of content depth, and allows for a great deal of flexibility when dealing with nested retrieval and display of data within a web page. Using nested pipelets the page being rendered can start to be divided into subcomponents, and those subcomponents can have more subcomponents of their own.

Figure 5.4 illustrates a nested pipelet page of depth 3. Each set of pipelets is loaded using a breadth first loading algorithm. So all the pipelets at depth n will be loaded and rendered before the system continues to load pipelets at depth $n+1$. Its also important to note that loading of JavaScript for all pipelets will be deferred until all pipelet content (HTML and CSS) has been loaded for the given depth.

Figure 5.4: Nested Pipelets with a depth of 3



5.3 Pipelet Priority

Pipelets that are part of the same depth can be prioritized explicitly using the `pipelet-priority` OpenPipe HTML attribute [see figure 5.5]. By default all pipelets are loaded in ascending alphanumeric order of the `pipelet_id` OpenPipe HTML attribute. The addition of an explicit `pipelet-priority` tag allows for a greater degree of control when loading pipelet components and, most

importantly allows for the developer to choose which pieces of the page should be loaded and transmitted first.

Figure 5.5: Sample pipelet containing HTML, CSS, and JavaScript

```
<!-- a root pipelet -->
<html>
<head>
<title>Root Pipelet!</title>
  <link rel="stylesheet" type="text/css" href="css/global.css
    " />
  <script type="text/javascript" src="js/app.js" ></script>
</head>
<body>
  <h1>Hello World!</h1>
  <div pipelet-id="pipelet-1"></div>
  <div pipelet-id="pipelet-2" pipelet-priority="1" ></div>
</body>
</html>
```

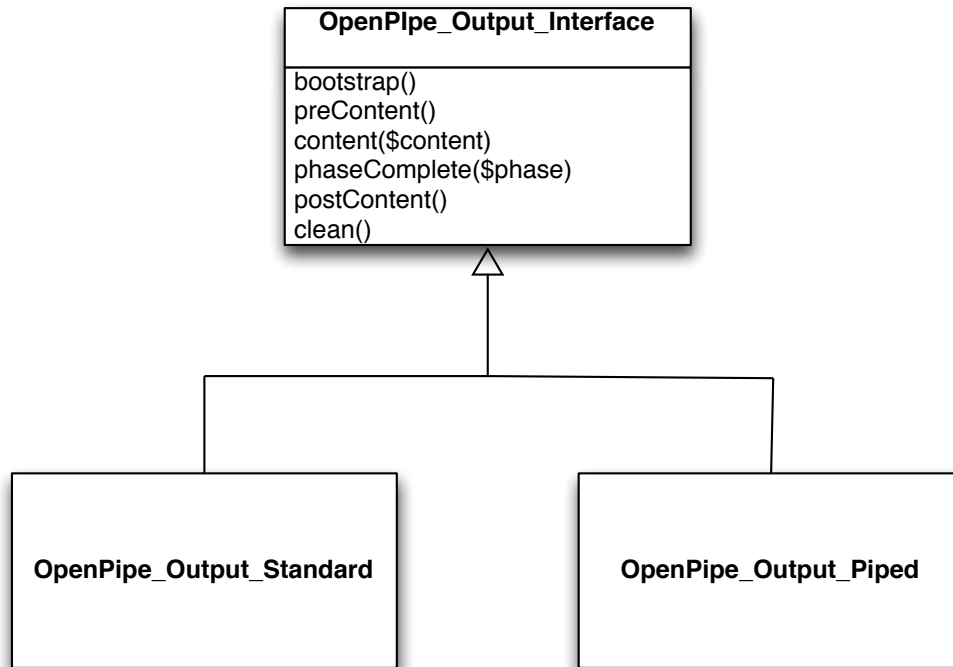
Chapter 6

Class Interfaces

OpenPipe defines core interfaces for pluggable components of the system. Through utilizing provided interfaces developers can contour and extend the framework to meet new and existing needs.

6.1 Output

Figure 6.1: A generalization of the OpenPipe output object



The output interface [see figure 6.1] defines how to conform to a set of output principles that allow for desired output functionality depending on the needs and capability of the client accessing an OpenPipe based web page. The output interface defines the following methods:

1. **bootstrap** - Allows implementor to setup and output any data before the content phase begins.
2. **preContent** - Called immediately before any content is to be outputted through the associated `content()` method.
3. **output** - Called when content is ready for output - This content is already generated HTML string.

4. **phaseComplete** - Called when an output phase is complete. A phase represents a layer of data (each layer of data can contain n number of deeper layers).
5. **postContent** - Called immediately after all data has been sent for output.
6. **clean** - Allows implementor to do any final cleanup and output. This is the last step in the output process.

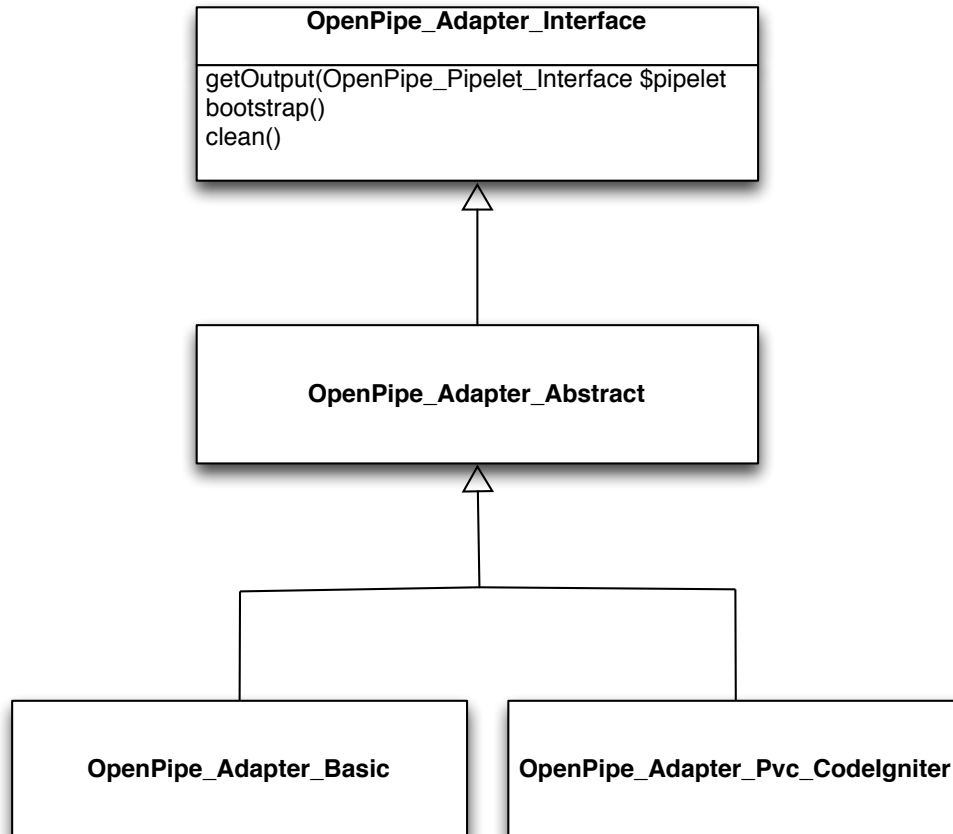
Out of the box the core OpenPipe library provides two output interface implementations:

1. **OpenPipe_Output_Piped** - Implementation of an OpenPipe output interface that sends data by way of an HTTP pipeline. This is done by loading the openpipe.js client library and associated libraries. The output handler handles extracting pipelet HTML data, and transmitting it as packed JSON object, which will be unpacked by the client openpipe.js library.
2. **OpenPipe_Output_Standard** - Implementation of an OpenPipe output interface that sends data as a standard HTML document. Content pieces are used to construct a complete HTML document, placing CSS and JavaScript in proper placement, and inject each content piece within a pipelet place holder on the server side. It's important to note that no javascript is required to complete output on the client web browser while using this output implementation.

This illustrates the power of decoupling the output system into an interface which is chosen at runtime based on the needs and capabilities of a given client receiving the output information. Through the use of the same output interface the OpenPipe runner can transparently interface with JavaScript capable devices and non JavaScript capable devices such as web crawlers and bots without needing to alter any information retrieved from a corresponding framework interface.

6.2 Framework Adapter

Figure 6.2: A generalization of the OpenPipe adapter object



A framework's adapter [see figure 6.2] is a crucial component of the OpenPipe library that converts web requests into underlying framework routing requests. These routing requests result in output. The output for each request is returned by the framework adapter to the OpenPipe output interpreter where it is parsed for more web requests that need to be retrieved by the framework adapter. This process continues until all output is retrieved.

Chapter 7

Design Patterns

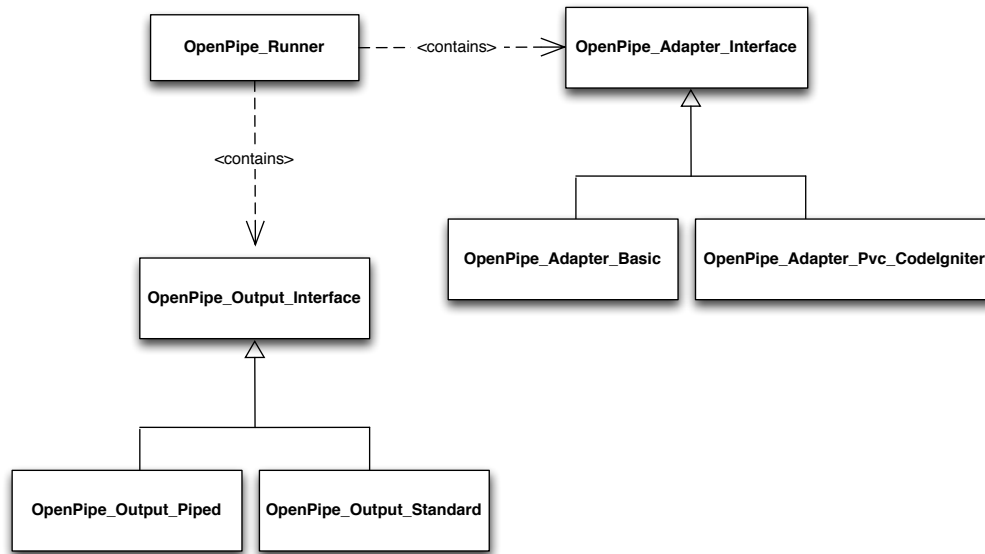
The OpenPipe library utilizes design patterns where appropriate to make the available components easier to comprehend from a conceptual level, and also easier to extend for future use. The patterns explained below were chosen to provide maximum flexibility to the underlying system when integrating with existing PHP web application systems and frameworks.

7.1 Strategy Pattern

OpenPipe uses the strategy design pattern to define families of objects that can be utilized by the `OpenPipe_Runner` class at runtime [see figure 7.1]. This helps effectively decouple the `OpenPipe_Runner` from rendering and output concerns. These two concerns can vary depending on:

1. The type of HTTP client accessing the web page (web browser, bot, crawler).
2. The type of framework that OpenPipe is using to access and render web page information (CodeIgniter, Zend Framework, CakePHP).

Figure 7.1: The strategy pattern utilized by OpenPipe for the main OpenPipe runner object



Another benefit of this design pattern implementation is that new adapters and output classes can be developed and utilized in the future. Once created they only need to be passed as parameters to the `OpenPipeRunner` class when it is instantiated [see figure 7.2].

Figure 7.2: Instantiation and running of an `OpenPipeRunner` object

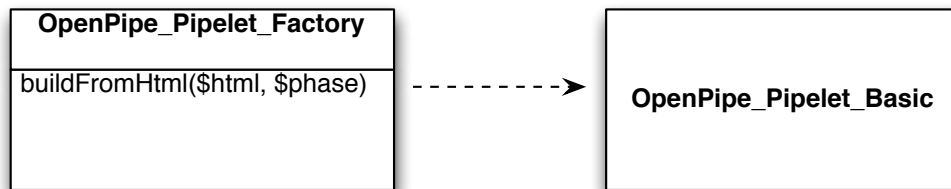
```
//starting an openpipe runner
$openPipeRunner = new OpenPipe_Runner($openPipeAdapter,
    $openPipeOutput);
$openPipeRunner->run(); //outputs openpipe content
```

7.2 Factory Pattern

OpenPipe utilizes the factory design pattern to construct pipelets from output received from a framework adapter [see figure 7.3]. The factory receives raw HTML data and current phase information. The factory then continues to parse embedded pipelet information from the HTML and return an array

of Pipelets that conform to the OpenPipe_Pipelet_Interface

Figure 7.3: The factory pattern utilized by OpenPipe



Chapter 8

Sequence Diagrams

8.1 Server

Every open pipe request cycle is handle by a core class named `OpenPipe_Runner`. The runner is responsible for orchestration communication with a class that implements the `OpenPipe_Adapter` interface. This communication results in:

1. Notification of bootstrapping and cleanup stages in the request cycle.
2. Retrieval of the root pipelet content (layout).
3. Retrieval of nested pipelets within the root pipelet.

Once the `OpenPipe_Runner` class has received information from the `OpenPipe_Adapter` its passes any renderable content to a class that implements the `OpenPipe_Output` interface. The `OpenPipe_Output` class handles sending data to the client, and removes any special concerns for how this data is transmitted away from the `OpenPipe_Runner`.

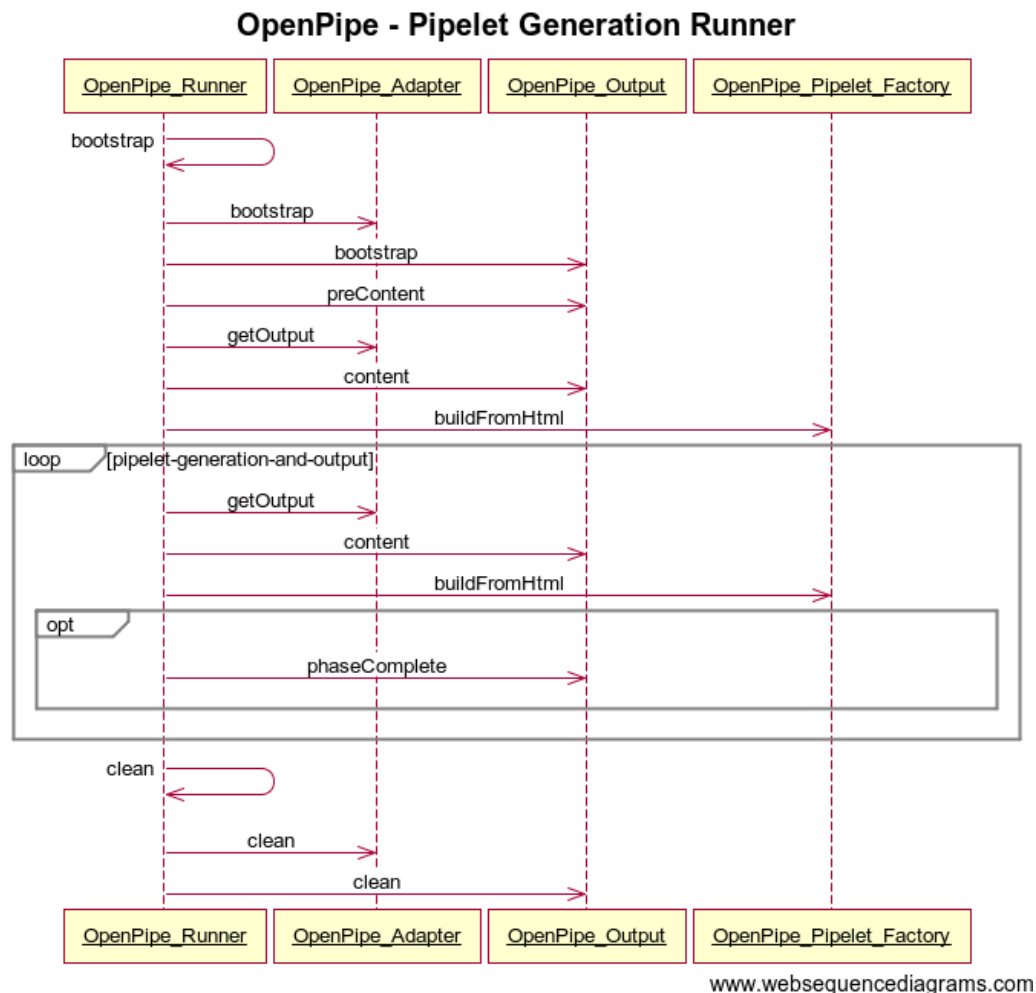
The `OpenPipe_Runner` is also responsible for calling upon a `Pipelet_Factory` to build pipelets from content gathered from the `Pipelet_Adapter`. The `Pipelet_Factory` returns an array of pipelets which contain information that can be sent to the `OpenPipe_Adapter` to gather the pipelet HTML and data, and then passed to the `OpenPipe_Output` class for rendering. The factory process is the main loop in the `OpenPipe_Runner` application.

All components of this process are pluggable and determined at runtime. The `OpenPipe_Runner` class depends only on the individual interfaced defined in the `OpenPipe` library, and utilizes specific design patterns such as

the strategy and factory patterns to decouple it directly from any class instantiation.

The sequence diagram for a complete OpenPipe request is outline in figure 8.1

Figure 8.1: OpenPipe Runner sequence diagram



8.2 Client

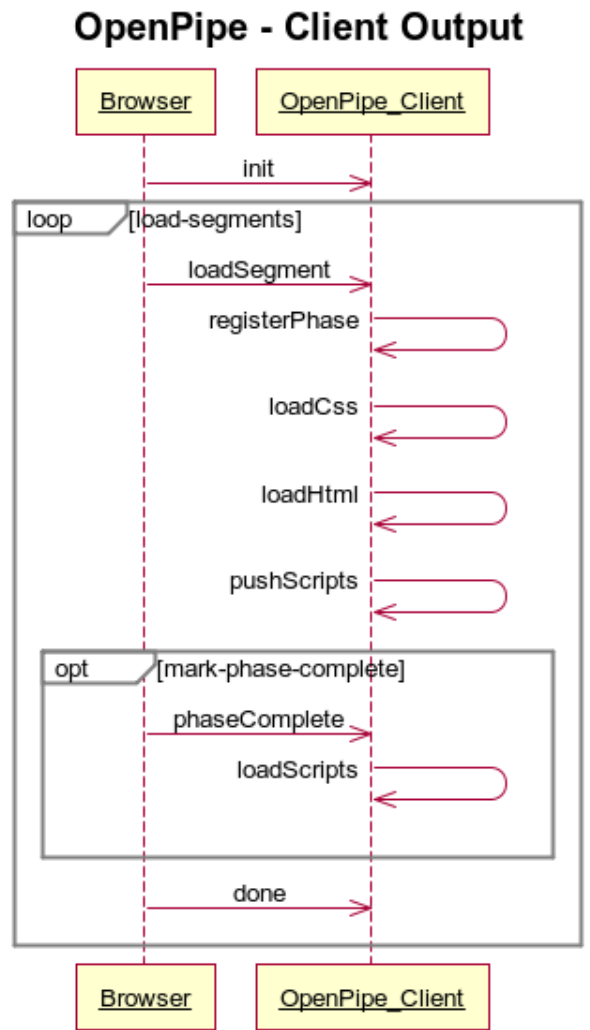
Once a request is handled and output is sent through an **OpenPipe_Output** class, this output must be interpreted and acted upon by a the **OpenPipe**

client library. The client library is built entirely of JavaScript and contains the following API calls:

1. **init** - this is called during the main page layout initialization stage. It handles setting up the layout and initializing the client library so it can load additional segments.
2. **loadSegment** - this is the main method call, which handles loading pipelet data from the server and rendering it within a web browser. It accepts segment data, which are essentially predefined JSON objects.
3. **registerPhase** - called when a segment is loaded. If the segment part of a new phase then the phase is recorded as starting and any script received will be queued until this phases content (HTML and CSS) has been loaded and rendered.
4. **loadCss** - loads a given array of CSS elements (<link /> and <style /> tags). The CSS information is inserted directly into the <head /> section of the HTML document].
5. **loadHtml** - load a given HTML document into the layout's pipelet placeholder. The placeholder to insert content to is determined using the segments corresponding id.
6. **pushScripts** - pushes a segment's set of script tags (both inline and external) onto a stack for later retrieval. JavaScript is loaded at the end of each loading phase. This allows for content represented as HTML and CSS to be loaded and viewable first before any possibly long JavaScript execution takes place.
7. **phaseComplete** - marks a phase as complete. When a phase is marked complete all JavaScript queued from segments loaded during the same phase will be appended to the <head /> section of the HTML document and executed in the order received.

The sequence diagram for a complete OpenPipe client processing cycle is outlined in figure 8.2

Figure 8.2: OpenPipe output sequence diagram



www.websequencediagrams.com

Figure 8.3: OpenPipe client side pipelet load calls

```
<html>
<head>
  <title>OpenPipe Sample Page</title>
  <script type="text/javascript" src="js/libs/jquery.js"></script>
  <script type="text/javascript" src="js/libs/underscore.js"></script>
  <script type="text/javascript" src="js/openpipe.js"></script>
</head>
<body>
  <div id="container" ><!-- root layout data. --></div>
  <!-- followed by openpipe client request calls -->
  <script type="text/javascript">
    op.load({...});
  </script>
  <script type="text/javascript">
    op.load({'id':'pipelet1', 'html': '...', 'css': [], 'scripts': []});
  </script>
  <script type="text/javascript">
    op.load({'id':'pipelet2', 'html': '...', 'css': [], 'scripts': []});
  </script>
  <script type="text/javascript">
    op.load({'id':'pipelet3', 'html': '...', 'css': [], 'scripts': []});
  </script>
  <script type="text/javascript">
    op.phaseComplete(1); op.done();
  </script>
</body>
</html>
```


Chapter 9

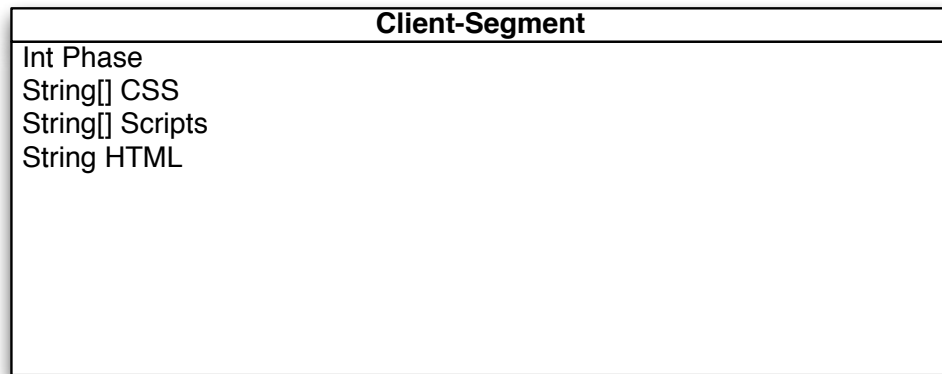
Data Objects

9.1 Transmitted Data

Pipelet data is transmitted as JSON to the client. Each individual pipelet transmitted as JSON is referred to as a segment in the OpenPipe.js client library. A segment contains the following data elements [see figure 9.1]:

1. **ID** - the id of the pipelet the information in the segment pertains to.
2. **CSS** - an array of inline and external css HTML tags. This information is extracted from the server output and organized before transmission to the client as a segment.
3. **Scripts** - an array of inline and external script HTML tags. This information is extracted from the server output and organized before transmission to the client as a segment.
4. **HTML** - raw HTML data that is left after css and script data extraction.

Figure 9.1: The client segment data object



Chapter 10

Things to Consider

10.1 PHP Output Buffering

PHP has built in functionality which prevents output data from being transmitted to a client connection until a certain amount of data has been placed into an output buffer. This however is counter productive to the operation of OpenPipe since each pipelet needs to be transferred immediately after is ready for output. This allows for the illusion of page rendering speed. If the buffer was left in place PHP would not output the data in a continuous stream and the experience would be similar to non HTTP Pipelined pages.

To get around this issue OpenPipe utilizes a custom output utility class which provides an output function named `echoNow` [see figure 10.1]. `echoNow` performs similarly to the standard `echo` function in PHP, but it is output buffer aware. Every time the `echoNow` function is called it queries the current PHP output buffer size and pads the output string with as much data that is needed so that the buffer will be full and the output data will be sent immediately. Utilizing this function classes within the OpenPipe library to not need to individually carry the concern of output buffering and how to circumvent it.

Figure 10.1: PHP function that helps bypass PHP output buffering that blocks the HTTP pipelining of data to the client browser

```
/**
 * Highly reusable output method which echos data NOW - by NOW
 * we mean in an intelligent way that takes into account
 * output buffering in PHP
 * as well as browser based deferred display of data (until
 * data is of x bytes) - Using this utility method one should
 * not have to worry about how
 * to immediately send data to an end client browser NOW
 * @param string $output the data to output NOW!
 * @param int|null $outputBufferSize the output buffer
 * currently in use -if a string is not of an output buffer
 * length it will be padded to meet the minimum buffer size -
 * If not provided this value will be looked up from the PHP
 * ini configuration value
 * @param string $paddingCharacter the character to pad output
 * with if the buffer is larger than the data to output
 */
public static function echoNow($output, $outputBufferSize=
    null, $paddingCharacter = ' '){

    //if the output buffer is null, then attempt to get it from
    php ini
    if($outputBufferSize === null){
        $outputBufferSize = @ini_get('output_buffering');
        if($outputBufferSize == 'Off') $outputBufferSize = 0;
    }

    //now that we know the buffer check to see how much we need
    to pad the string that is to be outputted
    $bufferSpace = $outputBufferSize - strlen($output);
    if($bufferSpace > 0){
        $output = $output.str_repeat($paddingCharacter,
            $bufferSpace);
    }

    //echo the string (with possible padding), then flush!
    echo $output;
    flush();
}
```

10.2 Dynamic JavaScript DOM Insertion

Inserting JavaScript into an existing DOM document needs to be done in a slightly different way than just appending the raw JavaScript data to the current HTML document being loaded. To do this in a reliable and cross browser way the boiler plate DOM function, 'createElement', is used to create a new script tag. Once the script tag is created the type and source code attributes are set independently using the JSON data that is sent with a piplet's JavaScript component. The code that accomplishes the JavaScript insertion can be found in figure ??

Figure 10.2: JavaScript code segment that allows for reliable cross browser insertion of dynamic JavaScript code into the DOM

```
var script = document.createElement('script');
script.type = jq_script.attr('type') || '';
script.src = jq_script.attr('src') || '';
$('body').append(script);
```

Chapter 11

Analysis

11.1 Data Collection Methodology

To collect data a simple but powerful script [see figure 11.1] was developed utilizing the Selenium framework to automate the systematic retrieval of performance timing data for OpenPipe enabled web pages. The web page selected for data collection and analysis is provided in the set of example for OpenPipe. The URL installed on a local system is 'HTTP://localhost/dev/openpipe/example/code'. Selenium was a perfect candidate for automated retrieval of data, because unlike other types of web request tools such as cURL or Wget which just send and receive raw HTTP data, Selenium drives a physical web browser through the utilizing of underlying browser API systems. This means that data is loaded and processed exactly the same as when an end user accesses a given website. This includes web requests, web response, loading and unloading of the DOM, and JavaScript execution. JavaScript execution was a crucial component in determining the load time for an OpenPipe web page, since all data is unpacked and loaded in browser via JavaScript after it has been received from the server.

Figure 11.1: A selenium script that retrieves performance and timing data from websites

```
#-----
# Script for recording performance timing of a given web page
# @author Sean Kenny <skenny214@gmail.com>
#-----
require 'pp'
require 'rubygems'
require 'csv'
require 'selenium-webdriver'

#get arguments
url = ARGV[0];
cycles = ARGV[1] || '5'
output_file = ARGV[2] || 'output.csv'

#open client driver for firefox
browser = Selenium::WebDriver.for :firefox

#open csv for writing output data
CSV.open(output_file, "w") do |csv|
  #for the amount of times the user wanted, get the page, get
  #the performance timing, and output to csv
  cycles.to_i.times do |i|
    browser.get url
    browser_timing = browser.execute_script("return window.
      performance.timing");
    openpipe_timing = browser.execute_script("return typeof(
      op) !== 'undefined' ? op.performance.timing.segments :
      null");

    # calculate wait time in two ways - if not openpipe then
    #just use reponse start
    if(openpipe_timing == nil)
      browser_timing['responseWaitTime'] = browser_timing['
        responseStart'] - browser_timing['requestStart']

    #if we have openpipe timing then the response wait time
    #is for first piece of actual content (assumed to be
    #pipelet of first priority)
    else
      browser_timing['responseWaitTime'] = openpipe_timing[1]
        - browser_timing['requestStart']
    end

    browser_timing['totalLoadWaitTime'] = browser_timing['
      loadEventEnd'] - browser_timing['requestStart']

    sorted_timing_values = []
    browser_timing.keys.sort.each do |key|
      sorted_timing_values << browser_timing[key]
    end

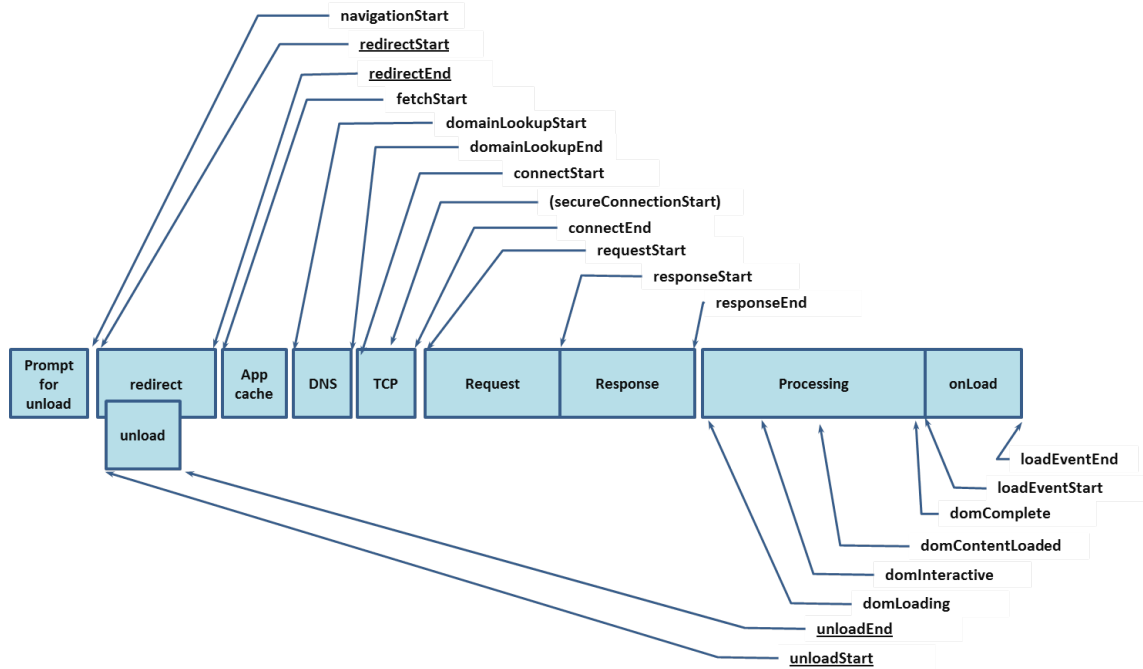
    csv << browser_timing.keys.sort if i==0
```

Utilizing Selenium to load a web page also allowed for controlled access to performance timing information recorded by web browsers during each web request. This performance timing information can be found in the DOM javascript element, ' window.performance.timing'. The data structure for window.performance.timing can be seen in figure 11.2. To make the meaning of the data found within the window.performance.timing data structure more clear a visualization timeline that illustrates when the performance timing events occur can be seen in figure 11.3.

Figure 11.2: DOM performance timing data made available via JavaScript [1]

```
interface PerformanceTiming {
    readonly attribute unsigned long long navigationStart;
    readonly attribute unsigned long long unloadEventStart;
    readonly attribute unsigned long long unloadEventEnd;
    readonly attribute unsigned long long redirectStart;
    readonly attribute unsigned long long redirectEnd;
    readonly attribute unsigned long long fetchStart;
    readonly attribute unsigned long long domainLookupStart;
    readonly attribute unsigned long long domainLookupEnd;
    readonly attribute unsigned long long connectStart;
    readonly attribute unsigned long long connectEnd;
    readonly attribute unsigned long long secureConnectionStart;
    ;
    readonly attribute unsigned long long requestStart;
    readonly attribute unsigned long long responseStart;
    readonly attribute unsigned long long responseEnd;
    readonly attribute unsigned long long domLoading;
    readonly attribute unsigned long long domInteractive;
    readonly attribute unsigned long long
        domContentLoadedEventStart;
    readonly attribute unsigned long long
        domContentLoadedEventEnd;
    readonly attribute unsigned long long domComplete;
    readonly attribute unsigned long long loadEventStart;
    readonly attribute unsigned long long loadEventEnd;
};
```


Figure 11.3: DOM performance timing data show as linear request [1]



11.2 Simulating Load

One important characteristic to consider when developing a web based library is how it performs under load. The dimension of load that was considered when performance testing OpenPipe was the number of concurrent connections the server was able to process, and how this affected overall performance with and without the OpenPipe library outputting data through an HTTP data pipeline.

To achieve an accurate simulation of load the freely available and open-source tool named, 'Siege'. Siege is an HTTP load testing and benchmarking utility. It was designed to let web developers measure their code under duress, to see how it will stand up to load on the internet [2]. OpenPipe was tested while siege was run at concurrency levels 10, 25, 50, and 100.

11.3 Results

Once the performance timing data was collected additional data points were calculated to clarify if any performance benefits could be found between a pipelined and non-pipelined version a webpage. This data was collected and laid out in a tabular format [see figure 11.4]. The columns in the presented table represent the following variables and formulas:

1. **Type** - Represents the type of external event that the server is issuing to complete the given web page request. Three types of web page requests were simulated:
 - (a) **Plain HTML** - The server does not connect to any external system. It simply renders and return HTML content. This HTML content also includes CSS and JavaScript.
 - (b) **Database** - The server creates a connection to a local database system and issues select queries over this connection to gather data. The data is not used in the display of the web page, but nevertheless this simulates the overhead necessary to connect to a local database system, loop through the data, and return a result.
 - (c) **Web Service** - The server create a connection to an external REST based web service. For the purposes of these tests the server connect to the twitter REST API and issues REST based query commands. The data is not used in the display of the web page, but nevertheless this simulates the overhead necessary to connect to an external REST based system, loop through the data, and return a result.
2. **Load** - The ammount of load that is being put on the web server during the testing time. Load was simulated using the command line tool named, 'Siege'. The number presented in this column represents the total number of concurrent connection being issues from Siege during the tests.
3. **Response** - The initial response time for a non piped page. $responseStart - requestStart$
4. **Response Piped** - The initial response time for a non piped page. This is calculated by recording the load time of the first piece of con-

tent within a pipelet (users sees something) and comparing that to the request start time. $firstPipeletLoadTimeEnd - requestStart$

5. **Total Time** and **Total Time Piped** - The total time it took to the load the page, starting at request start and ending at the final DOM loaded event. $loadEventEnd - requestStart$

Figure 11.4: Calculated response and load time in milliseconds. Data is based on timing data collected from automated browser runs via Selenium

Type	Load	Response	Response Piped	Total Time	Total Time Piped
<i>Plain HTML</i>	0	80.81632653	56.10204082	102.6122449	144.0408163
<i>Database</i>	0	149.0612245	74.14285714	166.5714286	127.6122449
<i>Web Service</i>	0	5144.959184	795.2244898	5164.77551	5103.959184
<i>Plain HTML</i>	10	141.9183673	90.51020408	187.4489796	172.3469388
<i>Database</i>	10	408.8367347	231.8979592	448.4081633	618.7142857
<i>Web Service</i>	10	5053.040816	767.5510204	5110.591837	5110.591837
<i>Plain HTML</i>	25	100	200	100	200
<i>Database</i>	25	100	200	100	200
<i>Web Service</i>	25	100	200	100	200
<i>Plain HTML</i>	50	100	200	100	200
<i>Database</i>	50	100	200	100	200
<i>Web Service</i>	50	100	200	100	200
<i>Plain HTML</i>	100	100	200	100	200
<i>Database</i>	100	100	200	100	200
<i>Web Service</i>	100	100	200	100	200

Chapter 12

Next Steps

12.1 Expanded Output API

Right now the OpenPipe system relies on an implicit layout output system that is integrated into a new or existing web framework. The API could be extended to provide a more explicit type of output system that may be useful in some scenarios that a developer may encounter. An example of an explicit API that could be developed can be found in figure 12.1

Figure 12.1: An explicit version of an OpenPipe output API

```
<?php
$layoutData = file_get_contents('layout.php');

$pipe = new OpenPipe_Output_Handle();
$pipe->sendLayout($layoutData);

$dbRecords = mysql_query('select * from facts');
$content = '';
foreach($dbRecords as $dbRecord){
    $content .= "<h1>User {$dbRecord['user_name']}
```

12.2 Minification

The OpenPipe output adapter could add another output step where JavaScript, CSS, and HTML code is minified. Minified code has all unnecessary characters removed from its content, without changing the functionality of the original program. Minification can play an important role in further optimizing the pipeline by reducing the overall size of individual pipelet components.

12.3 Addition of framework adapters

Out of the box OpenPipe allows for integration with the CodeIgniter MVC Framework. Utilizing the strategy design pattern it is possible to develop additional adapters that allow for integration with existing and future PHP frameworks. Some frameworks that would be next to integrate include:

1. Symfony
2. Zend Framework
3. CakePHP
4. Yii

12.4 Language agnostic Apache server extension

Currently the OpenPipe framework is limited to working with PHP based applications and websites. This is due to the fact that the entire server side implementation is built using the PHP language and is executed by the web server using additional modules like `mod_php`.

Web servers like apache provide application interfaces for developing server extensions. OpenPipe development could progress to include an embedded apache module that allows for integration with any web based applications utilizing any web scripting language. This would allow OpenPipe to integrate with tools such as Python, Ruby, and Perl.

Chapter 13

Code Listings

Listing 13.1: "OpenPipe/Runner.php"

```
1 <?php
2
3 require_once('Output/Util.php');
4 require_once('Adapter/Interface.php');
5 require_once('Pipelet/Factory.php');
6
7 /**
8  * A runner is the core object for any OpenPipe based
9  * application. A runner is responsible for gathering output
10  * from an OpenPipe_Adapter_Interface based
11  * adapter and returning to the end client browser as piped
12  * data objects. Before sending these piped based data
13  * objects this runner also ensures that the
14  * end client browser has been setup/instantiated
15  * appropriately by sending the CORE OpenPipe front end
16  * JavaScript libraries and the CORE HTML framework
17  * Once constructed calling this object run() method will
18  * kickoff the OpenPipe HTTP pipelining process
19  * @author Sean Kenny <skenny214@gmail.com>
20  * @package OpenPipe
21  * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
22  * State University (SCSU).
23  * @version 1.0.0
24  */
25 class OpenPipe_Runner {
26
27     /**
```

```

20  * The OpenPipe_Adapter_Interface object that is used by
    this OpenPipe_Runner to gather pipelets and load
    individual pipelet data
21  * @var OpenPipe_Adapter_Interface
22  */
23  protected $frameworkAdapter;
24
25  /**
26  * The OpenPipe_Output_Interface object that is used by this
    OpenPipe_Runner to send output data to the browser
27  * @var OpenPipe_Output_Interface
28  */
29  protected $output;
30
31
32
33  /**
34  * Constructs an OpenPipe_Runner object that communicated
    with the given OpenPipe_Adapter_Interface based object
35  * @param OpenPipe_Adapter_Interface $frameworkAdapter
36  * @param OpenPipe_Output_Interface $output
37  */
38  public function __construct(OpenPipe_Adapter_Interface
    $frameworkAdapter, OpenPipe_Output_Interface $output){
39      $this->frameworkAdapter = $frameworkAdapter;
40      $this->output = $output;
41  }
42
43
44  /**
45  * Is responsible for the ENTIRE OpenPipe HTTP pipelining
    lifecycle - handle all bootstrapping, base client
    library loading, output gathering, output transmission,
    Script, and shutdown
46  */
47  public function run(){
48      $this->bootstrap();
49      $this->output->preContent();
50
51      //ask the framework for the root output layer (the layout
        !). This contains the starting point for all pipelets
        to get recognized and loaded from
52      $layout = $this->frameworkAdapter->getOutput();
53      $this->output->content($layout);
54

```

```

55     $phase = 0;
56     $pipelets= OpenPipe_Pipelet_Factory::buildFromHtml(
57         $layout, $phase);
58     $pipeletsQueue = array();
59     while(!empty($pipelets)){
60
61         $currentPipelet = array_shift($pipelets);
62
63         $this->frameworkAdapter->getOutput($currentPipelet);
64         $this->output->content($currentPipelet);
65
66         //add pipelets contained within the current pipelet to
67         the the pipelet queue - the pipelet queue will get
68         loaded as part of the next phase
69         $pipeletsQueue = array_merge($pipeletsQueue,
70             OpenPipe_Pipelet_Factory::buildFromHtml(
71                 $currentPipelet->getOutput(), $phase+1));
72
73         //once the current pipelets have been completed. Check
74         the queue. If the queue is not empty then move batch
75         to the pipelets array for processing, and mark the
76         current pahse complete
77         if(empty($pipelets)){
78             $pipelets = $pipeletsQueue;
79             $pipeletsQueue = array();
80
81             $this->output->phaseComplete(++$phase);
82         }
83     }
84
85     $this->output->postContent();
86     $this->clean();
87 }
88
89 /**
90 * Performs bootstrapping of OpenPipe runner object and
91 calls the injected OpenPipe_Adapter_Interface bootstrap
92 () method at the very end

```



```

90  */
91  protected function bootstrap(){
92      $this->frameworkAdapter->bootstrap();
93      $this->output->bootstrap();
94  }
95
96  /**
97   * Performs Script of OpenPipe runner object and calls the
        injected OpenPipe_Adapter_Interface clean() method at
        the very end
98   */
99  protected function clean(){
100      $this->frameworkAdapter->clean();
101      $this->output->clean();
102  }
103
104
105
106 }

```

Listing 13.2: "OpenPipe/Pipelet/Interface.php"

```

1  <?php
2
3  /**
4   * An interface defining a pipelet. A pipelet is an atomic
        entity with an pipe based HTML layout. A pipelet is
        essentially a piece of content that is loaded
5   * in a priority based sequential fashion and outputted
        immediately to the end client browser, without having to
        wait for other pipelets or sub pipelets to be
6   * loaded as well. A pipelet's main purposed is to deliver
        content in modular packages that increase the, 'perceived
        ', load time of an HTML based PHP application.
7   * A pipelet at its core has an ID, phase, and output.
8   * - An id is derived either from the parent layout of parent
        pipelet. The id is used to identify and load content from
        a PHP OpenPipe adapter (OpenPipe_Adapter_Interface)
9   * - A phase is used to determine the timing and priority of a
        pipelet when it is received by an end client browser
10  * - The output is any data gathered from the OpenPipe adapter
        (utilizing the pipelet id). This data is subsequently
        piped to an end client browser
11  * @author Sean Kenny <skenny214@gmail.com>
12  * @package OpenPipe_Pipelet

```

```

13 * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
14 * @version 1.0.0
15 */
16 interface OpenPipe_Pipelet_Interface{
17
18     /**
19     * Sets the id of the pipelet (used to determine what
        content to gather from a Pipe Adapter)
20     * @param string $id a unique identifier for the pipelet
        that will signify importance to the client adapter and
        allow data to be looked up/generated accordingly
21     */
22     function setId($id);
23
24     /**
25     * Returns the current set pipelet id
26     */
27     function getId();
28
29
30     /**
31     * Sets the phase of the pipelet (used to determine loading
        priorities and sequences)
32     * @param int/string phase to set - Lower numbers are
        higher priority (1), than higher numbers (999)
33     */
34     function setPhase($phase);
35
36     /**
37     * Return the current set phase number
38     */
39     function getPhase();
40
41
42     /**
43     * Set the output that has been gathered for this pipelet
        from a Pipelet_Adapter
44     * @param string $output the output string that has been
        generated/gathered for this given pipelet
45     */
46     function setOutput($output);
47
48
49     /**

```

```

50  * Return the output that is currently set for the pipelet
51  */
52  function getOutput();
53
54  }

```

Listing 13.3: "OpenPipe/Pipelet/Abstract.php"

```

1  <?php
2
3  require_once('Interface.php');
4
5  /**
6   * Abstract implementation of the OpenPipe_Pipelet_Interface.
7   * Provided basic bindings for all methods defined in the
8   * interface.
9   * @author Sean Kenny <skenny214@gmail.com>
10  * @package OpenPipe_Pipelet
11  * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
12  * State University (SCSU).
13  * @version 1.0.0
14  */
15  abstract class OpenPipe_Pipelet_Abstract implements
16      OpenPipe_Pipelet_Interface {
17
18      /**
19       * The unique identifier for this pipelet - that
20       * distinguishes it from all others.
21       * @var string
22       */
23      protected $id;
24
25      /**
26       * The numbered phase of the pipelet to signify priority
27       * low-to-high
28       * @var string
29       */
30      protected $phase;
31
32      /**
33       * The output that is potentially gathered and piped as
34       * output for this pipelet
35       * @var string
36       */
37      protected $output;

```

```

32
33
34 /**
35  * Sets the id of the pipelet (used to determine what
      content to gather from a Pipe Adapter)
36  * @param string $id a unique identifier for the pipelet
      that will signify importance to the client adapter and
      allow data to be looked up/generated accordingly
37  */
38 public function setId($id){
39     $this->id = $id;
40 }
41
42 /**
43  * Returns the current set pipelet id
44  */
45 public function getId(){
46     return $this->id;
47 }
48
49
50 /**
51  * Sets the phase of the pipelet (used to determine loading
      priorities and sequences)
52  * @param int/string $phase to set - Lower numbers are
      higher priority (1), than higher numbers (999)
53  */
54 public function setPhase($phase){
55     $this->phase = $phase;
56 }
57
58 /**
59  * Return the current set phase number
60  */
61 public function getPhase(){
62     return $this->phase;
63 }
64
65
66 /**
67  * Set the output that has been gathered for this pipelet
      from a Pipelet_Adapter
68  * @param string $output the output string that has been
      generated/gathered for this given pipelet
69  */

```

```

70 public function setOutput($output){
71     $this->output = $output;
72 }
73
74 /**
75  * Return the output that is currently set for the pipelet
76  */
77 public function getOutput(){
78     return $this->output;
79 }
80
81
82 }

```

Listing 13.4: "OpenPipe/Pipelet/Base.php"

```

1 <?php
2
3 require_once('Abstract.php');
4
5 /**
6  * Provided a basic extension off of the
7   * OpenPipe_Pipelet_Abstract convenience implementation
8  * @author Sean Kenny <skenny214@gmail.com>
9  * @package OpenPipe_Pipelet
10 * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
11   * State University (SCSU).
12 * @version 1.0.0
13 */
14
15 class OpenPipe_Pipelet_Base extends OpenPipe_Pipelet_Abstract
16 {
17
18     /**
19     * Builds the object
20     * @param string $id the identifier for the pipelet
21     * @param int/string $phase the phase for the pipelet
22     * @return OpenPipe_Pipelet_Base new instance
23     */
24     public function __construct($id, $phase){
25         $this->setId($id);
26         $this->setPhase($phase);
27     }
28 }

```

Listing 13.5: "OpenPipe/Pipelet/Factory.php"

```

1 <?php
2
3 require_once('Base.php');
4
5 /**
6  * Generated pipelets using factory based methods
7  * @author Sean Kenny <skenny214@gmail.com>
8  * @package OpenPipe_Pipelet
9  * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
10   State University (SCSU).
11  * @version 1.0.0
12  */
13 class OpenPipe_Pipelet_Factory{
14
15     /**
16      * Extracts an array of pipelets from an given HTML document
17      * (represented as a string)
18      * @param string $html An html document represented via
19      * string
20      * @param int/string $phase The current phase of the pipelet
21      * loading process - This is assigned to any loaded
22      * pipelets extracted from the first html string parameter
23      * @return array all pipelets extracted from the HTML input
24      * and instantiated as OpenPipe_Pipelet_Base objects
25      */
26     public static function buildFromHtml($html, $phase){
27
28         //pipelet containers
29         $pipelets = array();
30         $pipeletGroups = array();
31
32         //setup regex to find pipelets - all pipelets need to
33         specify at least a pipelet-id attribute
34         preg_match_all('/<.*?pipelet-id.*?>/', $html, $matches,
35             PREG_SET_ORDER);
36
37         //for all matches extract the pipelet to its appropriate
38         group
39         foreach($matches as $match){
40             //reset match arrays
41             $pipeletIdMatch = array();
42             $pipeletPriorityMatch = array();
43

```

```

36      //extract pipelet attributes into corresponding match
        arrays
37      preg_match('/pipelet-id\=(\'.*?\'|".*?")/', $match[0],
        $pipeletIdMatch);
38      preg_match('/pipelet-priority\=(\'.*?\'|".*?")/',
        $match[0], $pipeletPriorityMatch);
39
40      //assign matches to local variables
41      $pipeletId = trim(@$pipeletIdMatch[1], '\'"');
42      $pipeletPriority = trim(@$pipeletPriorityMatch[1], '\'"
        ');
43      if(empty($pipeletPriority)) $pipeletPriority = 0;
44
45      //construct a pipelet based on extracted information,
        and place in proper group
46      if(!empty($pipeletId)){
47          $pipeletGroups[$pipeletPriority][$pipeletId] = new
            OpenPipe_Pipelet_Base($pipeletId, $phase);
48      }
49  }
50
51      //sort each groups array by pipelet id
52      foreach($pipeletGroups as $key => $pipeletGroup){
53          ksort($pipeletGroups[$key]);
54      }
55      //now that all groups are accounted for sort by priority
56      krsort($pipeletGroups);
57
58
59      //flatten all the segments to a single array
60      foreach($pipeletGroups as $pipeletGroup){
61          foreach($pipeletGroup as $pipelet){
62              $pipelets[] = $pipelet;
63          }
64      }
65
66      return $pipelets;
67  }
68
69 }

```

Listing 13.6: "OpenPipe/Output/Interface.php"

```

1 <?php
2
3 /**

```

```

4  * An interface defining the output mechanism for OpenPipe.
   * This abstraction allows for the implementing class to
   * handle individual pipelet output appropriately
5  * @author Sean Kenny <skenny214@gmail.com>
6  * @package OpenPipe_Pipelet
7  * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
   * State University (SCSU).
8  * @version 1.0.0
9  */
10 interface OpenPipe_Output_Interface {
11
12     /**
13      * Allow implementor to setup/output any data before the
        content phase begins
14     */
15     public function bootstrap();
16
17
18     /**
19      * Called immediately before any content is to be outputted
        via the associated content() method
20     */
21     public function preContent();
22
23
24     /**
25      * Called when content is ready for output - This content is
        already generated HTML string
26      * @param string $content html data
27     */
28     public function content($content);
29
30     /**
31      * Called when an output phase is complete -A phase
        represents a layer of data (each layer of data can
        contain n number of deeper layers)
32      * @param int $phase the number of the phase to mark
        complete
33     */
34     public function phaseComplete($phase);
35
36
37     /**
38      * Called immediately after all data has been sent for
        output

```



```

39  */
40  public function postContent();
41
42
43  /**
44   * Allows implementor to do any final cleanup/output - last
45     step in the output process
46  */
47  public function clean();
48  }

```

Listing 13.7: "OpenPipe/Output/Piped.php"

```

1  <?php
2
3  require_once('Interface.php');
4  require_once('Util.php');
5
6  /**
7   * Implementation of an OpenPipe output interface that sends
8     data via an HTTP pipeline.
9   * This is done by loading the openpipe.js client library and
10     associated libraries.
11   * The output handler handles extracting pipelet html data,
12     and transmitting it as packed JSON object -
13   * which will be unpacked by the client openpipe.js library
14   * @author Sean Kenny <skenny214@gmail.com>
15   * @package OpenPipe_Output
16   * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
17     State University (SCSU).
18   * @version 1.0.0
19  */
20  class OpenPipe_Output_Piped implements
21     OpenPipe_Output_Interface {
22
23
24     /**
25      * the web path where the client openpipejs library will
26        reside
27      * @var string the
28      */
29     protected $jsPath;
30
31
32     /**
33      * Builds an Piped output object

```

```

27  * @param string $jsPath the web path to the openpipejs
    client library
28  */
29  public function __construct($jsPath){
30      $this->jsPath = $jsPath;
31  }
32
33
34  /**
35  * Sends an initial string of output to force php and the
    browser to display piped output immediately
36  */
37  public function bootstrap(){
38      //set a 1024 newline - this forces soutput to the browser
    to start
39      echo str_repeat(" ",1024);
40      flush();
41  }
42
43
44  /**
45  * Outputs the framework for an HTTP Pipeline HTML document
    - this is essentially html and Javascript libraries -
    Note the html is unclosed (no ending body and html tags)
46  */
47  public function preContent(){
48      $header = "<!DOCTYPE HTML>\n<html><head>";
49      $header .= "<script type='text/javascript' src='{ $this->
    jsPath}/libs/jquery.js' ></script>";
50      $header .= "<script type='text/javascript' src='{ $this->
    jsPath}/libs/underscore.js' ></script>";
51      $header .= "<script type='text/javascript' src='{ $this->
    jsPath}/openpipe.js' ></script>";
52      $header .= "<script type='text/javascript' >op.init();</
    script>";
53      $header .= '</head><body><div pipelet-id="op-container
    "></div>';
54
55      OpenPipe_Output_Util::echoNow($header);
56  }
57
58
59
60  /**
61  * Extracts and outputs data in an openpipe.js friendly way

```

```

62  * @param string $content the content to be piped
        immediately - If CSS/JS is contained within the content
        this will be extracted and handled automatically
63  */
64  public function content($content){
65
66      if(is_string($content)){
67          $id = 'op-container';
68          $html = $content;
69      }else{
70          $id = $content->getId();
71          $html = $content->getOutput();
72      }
73
74      $css = array_merge(OpenPipe_Output_Util::extractLinkTags(
        $html), OpenPipe_Output_Util::extractStyleTags($html))
        ;
75      $js = OpenPipe_Output_Util::extractScriptTags($html);
76
77      $html = str_replace("'", "\\'", $html);
78      $css = json_encode($css);
79      $js = json_encode($js);
80
81
82      OpenPipe_Output_Util::echoJsNow("op.load({'id': '$id', '
        html': '$html', 'css': $css, 'scripts': $js});");
83  }
84
85
86  /**
87  * Handles a phase complete signal by sending the openpipejs
        phaseComplete command to the client browser
88  * @param int $phase the number of the phase that has been
        completed
89  */
90  public function phaseComplete($phase){
91      OpenPipe_Output_Util::echoJsNow("op.phaseComplete($phase)
        ;");
92  }
93
94
95  /**
96  * Outputs the closing framework elements for an HTTP
        Pipeline HTML document - sends shutdown (done) method
        for client library and close initially open body and

```

```

103         html tags
104     */
105     public function postContent(){
106         OpenPipe_Output_Util::echoJsNow('op.done();');
107         OpenPipe_Output_Util::echoNow('</body></html>');
108     }
109
110     /**
111     * This handler is always fresh and so clean clean
112     */
113     public function clean(){
114         //we're all clean!
115     }
116 }

```

Listing 13.8: "OpenPipe/Output/Standard.php"

```

1 <?php
2
3 require_once('Interface.php');
4 require_once('Util.php');
5
6 /**
7  * Implementation of an OpenPipe output interface that sends
8  * data as a standard HTML document
9  * Content pieces are used to construct a complete HTML
10 * document, placing CSS and JavaScript in proper
11 * placement, and inject each content piece within a pipelet
12 * place holder on the server side. It's
13 * important to note that no javascript is required to
14 * complete output on the client web browser while
15 * utilizing this output implementation
16 * @author Sean Kenny <skenny214@gmail.com>
17 * @package OpenPipe_Output
18 * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
19 * State University (SCSU).
20 * @version 1.0.0
21 */
22 class OpenPipe_Output_Standard implements
23     OpenPipe_Output_Interface {
24
25     /**

```

```

20  * linear array of style tags extracted from content and
    stored as string data
21  * @var array
22  */
23  protected $styles;
24
25  /**
26  * linear array of link tags extracted from content and
    stored as string data
27  * @var array
28  */
29  protected $links;
30
31  /**
32  * linear array of script tags extracted from content and
    stored as string data
33  * @var array
34  */
35  protected $scripts;
36
37  /**
38  * main html content stored as string and injected piece by
    piece as new content becomes available
39  * @var string
40  */
41  protected $content;
42
43
44  /**
45  * Setup all the variables that will be needed to generate
    proper output
46  */
47  public function bootstrap(){
48      $this->styles = array();
49      $this->links = array();
50      $this->scripts = array();
51      $this->content = '';
52  }
53
54
55  /**
56  * because standard output does not send any output until
    the end (clean method) - This method is not needed
57  */
58  public function preContent(){

```

```

59     //nothing to do for standard based output
60 }
61
62
63 /**
64 * takes content and builds an complete html document piece
65 by piece
66 * @param string/OpenPipe_Pipelet_Interface $content the
67 html content that will have data extracted and assigned
68 for final output
69 */
70 public function content($content){
71     if(is_string($content)){
72         $id = '';
73         $html = $content;
74     }else{
75         $id = $content->getId();
76         $html = $content->getOutput();
77     }
78
79     //get the link, style, and script tages in each content
80     section
81     $this->styles = array_merge($this->styles,
82         OpenPipe_Output_Util::extractStyleTags($html));
83     $this->links = array_merge($this->links,
84         OpenPipe_Output_Util::extractLinkTags($html));
85     $this->scripts = array_merge($this->scripts,
86         OpenPipe_Output_Util::extractScriptTags($html));
87
88     $this->injectHtml($id, $html);
89 }
90
91
92
93
94 /**
95 * because standard output does not send any output until
96 the end (clean method) - This method is not needed
97 * @param int phase not needed
98 */
99 public function phaseComplete($phase){
100     //nothing to do for standard based output
101 }
102
103
104 /**
105 * because standard output does not send any output until

```

```

    the end (clean method) - This method is not needed
96 */
97 public function postContent(){
98     //nothing to do for standard based output
99 }
100
101
102 /**
103  * Takes all of the gathered output and send the final html
    document as part of this last step
104 */
105 public function clean(){
106     $finalOutput = "<!DOCTYPE HTML>\n<html><head>";
107
108     //put the collected scripts before body close
109     foreach($this->links as $link){
110         $finalOutput .= $link;
111     }
112
113     //put the collected styles in the head;
114     foreach($this->styles as $style){
115         $finalOutput .= $style;
116     }
117
118     $finalOutput .= '</head><body>';
119     $finalOutput .= $this->content;
120
121
122     //put the collected scripts before body close
123     foreach($this->scripts as $script){
124         $finalOutput .= $script;
125     }
126
127     $finalOutput .= '</body></html>';
128
129     echo $finalOutput;
130 }
131
132
133
134 /**
135  * Attempts to inject the given html data into the currently
    recorded data - The point of injection is determined by
    the id provided
136  * @param string $pipeletId the identifier for the pipelet

```

```

137     that will have html content injected within it
138     * @param string $html the content that will be injected
139     into the current gathered output
140     */
141     protected function injectHtml($pipeletId, $html){
142         //if the content is currently empty, no injection needs
143         to take place. Just set as the content root
144         if($this->content == ''){
145             $this->content = $html;
146         }
147         //if we have content, find the injection point and
148         perform the string replacement with a regex
149         }else{
150             $this->content = preg_replace("/(<.*?pipelet-id=(?:\"
151             $pipeletId\"|'{$pipeletId}').*?>)/ms", "\\1 $html",
152             $this->content);
153     }
154 }
155 }
156 }

```

Listing 13.9: "OpenPipe/Output/Util.php"

```

1 <?php
2
3 /**
4  * Utility object which provides reusable output based
5  * services for HTTP Pipeline based systems
6  * @author Sean Kenny <skenny214@gmail.com>
7  * @package OpenPipe_Output
8  * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
9  * State University (SCSU).
10  * @version 1.0.0
11  */
12
13 class OpenPipe_Output_Util {
14
15     /**
16     * Given an html string extract the link information from
17     the raw data and return
18     * @param string $html the html string to extract script
19     tags from
20     * @return array strings containing link tags found within
21     the html string
22     */
23 }

```



```

17 public static function extractLinkTags(&$html){
18     preg_match_all('/<link.*?\>/ms', $html, $matches,
        PREG_SET_ORDER);
19     $html = preg_replace('/<link.*?\>/ms', '', $html);
20
21     $links = array();
22     foreach($matches as $match){
23         $links[] = $match[0];
24     }
25
26     return $links;
27 }
28
29
30 /**
31  * Given an html string extract the style information from
        the raw data and return
32  * @param string $html the html string to extract style tags
        from
33  * @return array strings containing style tags found within
        the html string
34  */
35 public static function extractStyleTags(&$html){
36     preg_match_all('/<style.*?>.*?\</style>/ms', $html,
        $matches, PREG_SET_ORDER);
37     $html = preg_replace('/<style.*?>.*?\</style>/ms', '',
        $html);
38
39     $styles = array();
40     foreach($matches as $match){
41         $styles[] = $match[0];
42     }
43
44     return $styles;
45 }
46
47
48 /**
49  * Given an html string extract the information from the
        raw data and return
50  * @param string $html the html string to extract script
        tags from
51  * @return array strings containing script tags found within
        the html string
52  */

```

```

53 public static function extractScriptTags(&$html){
54     preg_match_all('/<script.*?>.*?</script>/ms', $html,
        $matches, PREG_SET_ORDER);
55     $html = preg_replace('/<script.*?>.*?</script>/ms', '',
        $html);
56
57     $scripts = array();
58     foreach($matches as $match){
59         $scripts[] = $match[0];
60     }
61
62     return $scripts;
63 }
64
65
66
67
68 /**
69  * Outputs javascript data in piped format - Piped format
        implies minimized and able to be placed in a pipe
        JavaScript array
70  * @param string $output the output data (javascript) to be
        wrapped in a javascript tagged and echoed immediately
71  * @param boolean $wrapTags wrap the output in a script
        opening and closing tag
72  * @param int|null $outputBufferSize the size of the buffer
        currently in use - used to determine how much passing
        must be used for output to skip buffering
73  * @param string $paddingCharacter the character that will
        be used if padding must occur
74  */
75 public static function echoJsNow($output, $wrapTags=true,
        $outputBufferSize=null, $paddingCharacter=' '){
76     $output = str_replace("\n", '', $output);
77     if($wrapTags === true) $output = '<script type="text/
        javascript" >'.$output.'</script>';
78
79     self::echoNow($output, $outputBufferSize,
        $paddingCharacter);
80 }
81
82
83 /**
84  * Highly reusable output method which echos data NOW - by
        NOW we mean in an intelligent way that takes into

```

```

85      account output buffering in PHP
86      * as well as browser based deferred display of data (until
      data is of x bytes) - Using this utility method one
      should not have to worry about how
87      * to immediately send data to an end client browser NOW
88      * @param string $output the data to output NOW!
89      * @param int/null $outputBufferSize the output buffer
      currently in use -if a string is not of an output buffer
      length it will be padded to meet the minimum buffer
      size - If not provided this value will be looked up from
      the PHP ini configuration value
90      * @param string $paddingCharacter the character to pad
      output with if the buffer is larger than the data to
      output
91      */
92      public static function echoNow($output, $outputBufferSize=
      null, $paddingCharacter = ' '){
93
94          //if the output buffer is null, then attempt to get it
      from php ini
95          if($outputBufferSize === null){
96              $outputBufferSize = @ini_get('output_buffering');
97              if($outputBufferSize == 'Off') $outputBufferSize = 0;
98          }
99
100         //now that we know the buffer check to see how much we
      need to pad the string that is to be outputted
101         $bufferSpace = $outputBufferSize - strlen($output);
102         if($bufferSpace > 0){
103             $output = $output.str_repeat($paddingCharacter,
      $bufferSpace);
104         }
105
106         //echo the string (with possible padding), then flush!
107         echo $output;
108         flush();
109     }
110 }

```

Listing 13.10: "OpenPipe/Adapter/Interface.php"

```

1 <?php
2
3 /**

```

```

4  * An interface defining an adapter which bridges php based
    applications with OpenPipe. OpenPipe will call the adapter
    to load layouts and pipelets.
5  * In essence the adapter is responsible for making sure that
    the php based application is instantiated, bootstrapped,
    and run appropriately to obtain
6  * the request element (either layout or pipelet)
7  * @author Sean Kenny <skenny214@gmail.com>
8  * @package OpenPipe_Adapter
9  * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
10 * @version 1.0.0
11 */
12 interface OpenPipe_Adapter_Interface {
13
14     /**
15      * return output from the php application for immediate
        piping
16      * @param OpenPipe_Pipelet_Interface $pipelet a pipelet
        which supplies information on
17      * @return mixed implementors are free to return what they
        will
18      */
19     function getOutput(OpenPipe_Pipelet_Interface $pipelet =
        null);
20
21     /**
22      * called once during the initialization of an OpenPipe
        runner
23      */
24     function bootstrap();
25
26     /**
27      * called once during the shut down of an OpenPipe runner
28      */
29     function clean();
30 }

```

Listing 13.11: "OpenPipe/Adapter/Abstract.php"

```

1 <?php
2
3 require_once('Interface.php');
4
5 /**

```

```

6  * Represents an abstract OpenPipe adapter. As an abstract
   class it provides basic services for obtaining the layout
   (root object for pipelets), and generating
7  * output for pipelets that are requested. Any object which
   extends this class will implement the getLayout() and
   getContent() methods. This abstract class will
8  * handle the details in regards to buffering output and
   sending it back to the requesting object.
9  * @author Sean Kenny <skenny214@gmail.com>
10 * @package OpenPipe_Adapter
11 * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
   State University (SCSU).
12 * @version 1.0.0
13 */
14 abstract class OpenPipe_Adapter_Abstract implements
   OpenPipe_Adapter_Interface{
15
16     /**
17     * Returns output for the given pipelet - Output is web
       content (html, css, javascript) - If Pipelet is null
       then the layout is generated.
18     * @param OpenPipe_Pipelet_Interface|null $pipelet if not
       specified then the adapter will generate the pipelet
       layout by default
19     * @return string/OpenPipe_Pipelet_Interface given string
       output either generated for layout or a
       OpenPipe_Pipelet_Interface with output set
20     */
21     public function getOutput(OpenPipe_Pipelet_Interface
       $pipelet = null){
22
23         ob_start();
24
25         if($pipelet === null){
26             $this->getLayout();
27         }else{
28             $this->getContent($pipelet);
29
30         }
31
32         $output = ob_get_contents();
33         ob_end_clean();
34
35
36         if($pipelet !== null){

```

```

37         $pipelet->setOutput($output);
38         return $pipelet;
39     }else{
40         return $output;
41     }
42
43 }
44
45
46 /**
47  * Method should return the layout for the given web request
48  * @return string the root layout for all pipelets to be
49     derived from
50 */
51 abstract protected function getLayout();
52
53 /**
54  * Method should return the layout for the given web request
55  * @param OpenPipe_Pipelet_Interface $pipelet the pipelet to
56     get content from
57  * @return string the root layout for all pipelets to be
58     derived from
59 */
60 abstract protected function getContent(
61     OpenPipe_Pipelet_Interface $pipelet);
62
63
64 /**
65  * This abstract class does not provided any bootstrapping
66     logic
67 */
68 public function bootstrap(){ }
69
70 /**
71  * This abstract class does not provided any clean logic
72 */
73 public function clean(){ }
74
75 }

```

Listing 13.12: "OpenPipe/Adapter/Basic.php"

```

1 <?php
2
3 require_once('Abstract.php');
4

```

```

5  /**
6  * A basic adapter which provides an implementation of the
   * abstract adapter class. It simply loads layouts and
   * pipelets from known directories given during
7  * the constructor process
8  * @author Sean Kenny <skenny214@gmail.com>
9  * @package OpenPipe_Adapter
10 * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
   * State University (SCSU).
11 * @version 1.0.0
12 */
13 class OpenPipe_Adapter_Basic extends
   OpenPipe_Adapter_Abstract {
14
15     /**
16     * The full path to the layouts php files that will be
       loaded by this adapter
17     * @var string
18     */
19     public $layoutsPath;
20
21     /**
22     * The full path to the pipelets php files that will be
       loaded by this adapter
23     * @var string
24     */
25     public $pipeletsPath;
26
27     /**
28     * Constructs a new Basic pipe adapter
29     * @param string $layoutsPath the full path to the layouts
       php file that will be loaded by this adapter
30     * @param string $pipeletsPath the full path to the pipelet
       php files that will be loaded by this adapter
31     */
32     public function __construct($layoutsPath, $pipeletsPath){
33         $this->layoutsPath = $layoutsPath;
34         $this->pipeletsPath = $pipeletsPath;
35     }
36
37     /**
38     * loads a php layout via include()
39     * @param string $id the id of the layout to be used - An id
       is the filename without the php extension - For example
       default.php would be default

```

```

40  * @return void
41  */
42  protected function getLayout($id='default'){
43      include($this->layoutsPath.'/'.$id.'.php');
44  }
45
46  /**
47   * loads a php pipelet via include()
48   * @param OpenPipe_Pipelet_Interface $pipelet the pipelet to
49     load content for
50   * @return void
51   */
52  protected function getContent(OpenPipe_Pipelet_Interface
53     $pipelet){
54      include($this->pipeletsPath.'/'.$pipelet->getId().'php');
55  }

```

Listing 13.13: "OpenPipe/Adapter/Pvc/CodeIgniter.php"

```

1  <?php
2
3  require_once(dirname(__FILE__).'../Abstract.php');
4
5  //declare global variables that CodeIgniter 2 needs to
6     function. This will be called on before including code
7     igniter files
8  $BM; $CFG; $UNI;
9
10 /**
11  * A PMVC adapter which provides an implementation of for
12     CodeIgniter 2.x applications to take advantage of piped
13     output
14  * @author Sean Kenny <skenny214@gmail.com>
15  * @package OpenPipe_Adapter
16  * @license (c) 2011-2012 Sean Kenny, Southern Connecticut
17     State University (SCSU).
18  * @version 1.0.0
19  */
20 class OpenPipe_Adapter_Pvc_CodeIgniter extends
21     OpenPipe_Adapter_Abstract {
22
23     /**

```



```

19  * The root path of the currently active CodeIgniter
    application
20  * @var string
21  */
22  protected $appRootPath;
23
24
25  /**
26  * The file name within the $appRootPath that bootstraps and
    runs a CodeIgniter application
27  * @var string
28  */
29  protected $indexFileName;
30
31  /**
32  * Constructs a new CodeIgniter pipe adapter
33  * @param string $appRootPath the root path of the currently
    active CodeIgniter application
34  * @param string $indexFileName the file name within the
    $appRootPath that bootstraps and runs a CodeIgniter
    application
35  * @return OpenPipe_Adapter_Pvc_CodeIgniter new instance of
    this object
36  */
37  public function __construct($appRootPath, $indexFileName='
    index.pipe.php'){
38      $this->appRootPath = rtrim($appRootPath, '/');
39      $this->indexFileName = $indexFileName;
40  }
41
42  /**
43  * loads a php layout by starting the code igniter index
    file
44  */
45  protected function getLayout(){
46      global $BM, $CFG, $UNI;
47      include($this->appRootPath.'/' . $this->indexFileName);
48  }
49
50
51  /**
52  * loads a php pipelet via CodeIgniter controller - being
    sure to play nice with output class
53  * @param OpenPipe_Pipelet_Interface $pipelet the pipelet to
    be used.

```

```

54  */
55  protected function getContent(OpenPipe_Pipelet_Interface
56      $pipelet){
57      global $BM, $CFG, $UNI;
58
59      $CI = &get_instance();
60      $CI->output->set_output('');
61
62      call_user_func_array(array($CI, $pipelet->getId()), array
63          ());
64
65      $CI->output->_display();
66  }

```

Listing 13.14: "openpipe.js"

```

1  //      OpenPipe.js 1.0.0
2  //      (c) 2011-2012 Sean Kenny, Southern Connecticut State
3  //      University (SCSU).
4  //      OpenPipe is freely distributable under the MIT license
5
6  .
7
8  (function() {
9
10     //debug vars - for timing
11     var isDebug = (document.location.search.indexOf('debug=true
12         ') != -1);
13     var debugInitTime;
14     var debugDoneTime;
15     var debugLastSegmentLoadTime;
16
17     // Establish the root object, 'window' in the browser, or '
18     global' on the server.
19     var root = this;
20
21     // Save the previous value of the 'op' variable.
22     var previousOpenPipe = root.op;
23
24     var isFirstSegment = true;
25
26     var lastPhase = 0;
27     var phases = [];
28     var scripts = [];

```

```

25
26
27 var op = {};
28
29 //if this is debug then record performance statistics
30 if(isDebug === true){
31     op.performance = {};
32     op.performance.timing = {};
33     op.performance.timing.segments = [];
34 }
35
36 //init the OpenPipe client - hide pipelets initally (no
    FLQCs)
37 op.init = function(){
38     //record times for logging
39     if(isDebug === true){
40         debugInitTime = new Date().getTime();
41         debugLastSegmentLoadTime = debugInitTime;
42     }
43
44     $("*[pipelet-loading-indicator]").append('<div class="op-
        loading">loading</div>');
45     $("*[pipelet-auto-show='true']").hide();
46 };
47
48 //load a given segment object into the pipelined document
49 op.load = function(segment){
50     this.loadSegment(segment);
51
52     //record times and output to the log
53     if(isDebug === true){
54         var debugCurrentSegmentLoadTime = new Date().getTime();
55         op.performance.timing.segments.push(
            debugCurrentSegmentLoadTime);
56         console.log('SEGMENT "' + segment.id + '" LOAD TIME: ' + (
            debugCurrentSegmentLoadTime - debugLastSegmentLoadTime
        ));
57         console.log('TIME UNTIL SEGMENT: ' + (
            debugCurrentSegmentLoadTime - debugInitTime));
58         debugLastSegmentLoadTime = debugCurrentSegmentLoadTime;
59     }
60 };
61
62
63 //register a given phase from a segment

```

```

64 op.registerPhase = function(phase){
65     if(typeof(phase) == 'undefined') phase = 0;
66     if(typeof(phases[phase]) == 'undefined') phases[phase] =
        phase;
67 };
68
69 //mark a phase complete. A completed phase has its deferred
        JavaScript elemetns loaded (and consequently run)
70 op.phaseComplete = function(phase){
71     lastPhase = phase;
72     this.loadScripts(phase);
73 };
74
75
76 //mark the pipeline as completed. For all the phases mark
        complete if not already done (load all JavaScript for
        any phases where complete notification was not sent)
77 op.done = function(){
78     var that = this;
79     _.each(phases, function(phase){
80         if(phase > lastPhase) that.phaseComplete(phase);
81     });
82
83     if(isDebug === true){
84         debugDoneTime = new Date().getTime();
85         console.log('TOTAL LOAD TIME: '+(debugDoneTime-
            debugInitTime));
86     }
87
88 };
89
90
91
92 //load an pipeline segment. A segment contains an id (
        element to load on page), css, html and JavaScript. The
        id is found, css is loaded, html is loaded , and
        JavaScript is queued until the end of the segments phase
93 op.loadSegment = function(segment){
94     if(isFirstSegment === true){
95         isFirstSegment = false;
96         $("*[pipelet-loading-indicator]").hide();
97     }
98
99     this.registerPhase(segment.phase);
100

```

```

101     if(typeof(segment.css) != 'undefined') this.loadCss(
        segment.css);
102     if(typeof(segment.html) != 'undefined') this.loadHtml(
        segment.html, segment.id);
103     $("[pipelet-id='"+segment.id+"'").show();
104
105     if(typeof(segment.scripts) != 'undefined') this.
        pushScripts(segment.scripts, segment.phase);
106 };
107
108
109
110 //load html into the given #id'ed element by appending
111 op.loadHtml = function(html, id){
112     if(typeof(id) == 'undefined') id = 'content';
113
114     $("[pipelet-id='"+id+"'").append(html);
115 };
116
117 //load css into the head of the documents
118 op.loadCss = function(css){
119
120     var that = this;
121     _.each(css, function(css_item){
122         $('head').append(css_item);
123     });
124
125
126 };
127
128
129
130 //push a set of script blocks onto a phase of the pipeline
cycle. Once the phase is marked complete all the scripts
in that phase will be loaded (and executed)
131 op.pushScripts = function(scripts, phase){
132
133     var that = this;
134     _.each(scripts, function(script){
135         that.pushScript(script, phase);
136     });
137
138 };
139
140 //push a single script block onto a phase of the pipeline

```

```

        cycle. Once the phase is marked complete all the script
        in that pahse will be loaded (and execuated)
141 op.pushScript = function(script, phase){
142     if(typeof(phase) == 'undefined') phase=lastPhase+1;
143     if(typeof(scripts[phase]) == 'undefined') scripts[phase]
        = [];
144     scripts[phase].push(script);
145 };
146
147 //loads all the pusheds script for a given phase
148 op.loadScripts = function(phase){
149     var that = this;
150     _.each(scripts[phase], function(script_item){
151         jq_script = $(script_item);
152
153         //if this is an external javascript then we make a new
            dom object to house it from the string data
154         if(typeof(jq_script.src) != 'undefined'){
155             var script = document.createElement('script');
156             script.type = jq_script.attr('type') || '';
157             script.src = jq_script.attr('src') || '';
158             $('body').append(script);
159
160             //if this was just an internal javascript append to
                body and jquery will execute it
161         } else {
162             $('body').append(script_item);
163         }
164
165     });
166 };
167
168
169 //set the root open pipe object
170 root['op'] = op;
171
172
173 }).call(this);

```

Bibliography

- [1] Navigation Timing, Editor's Draft July 16th, 2012 [Online]. Available: <http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html>
- [2] Siege Home, January 30, 2012 [Online]. Available: <http://www.joedog.org/siege-home/>