# 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

# List of Figures

# 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:

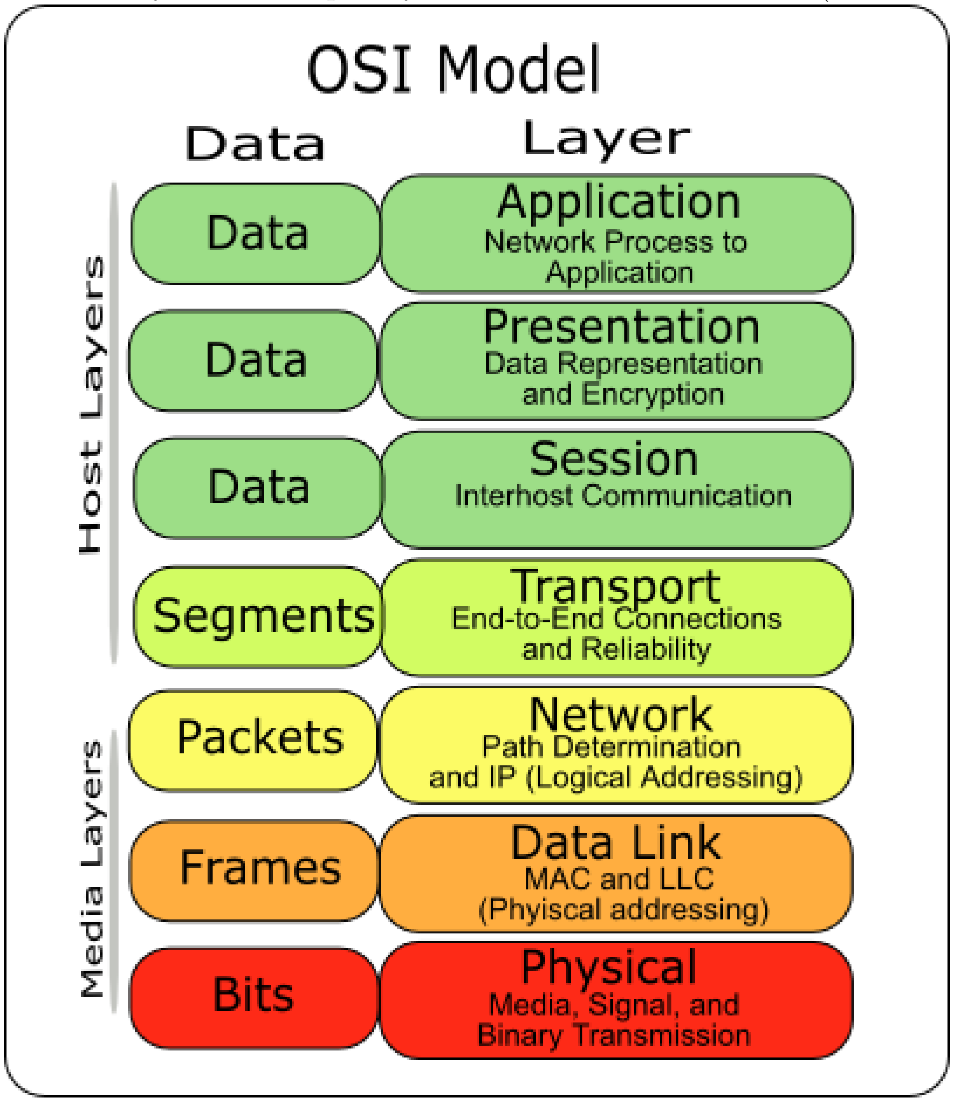
- Use a content delivery network
- Add an Expires header (HTTP 1.1)
- Gzip components
- Put CSS at the top of the page
- Avoid CSS expressions

- 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 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).

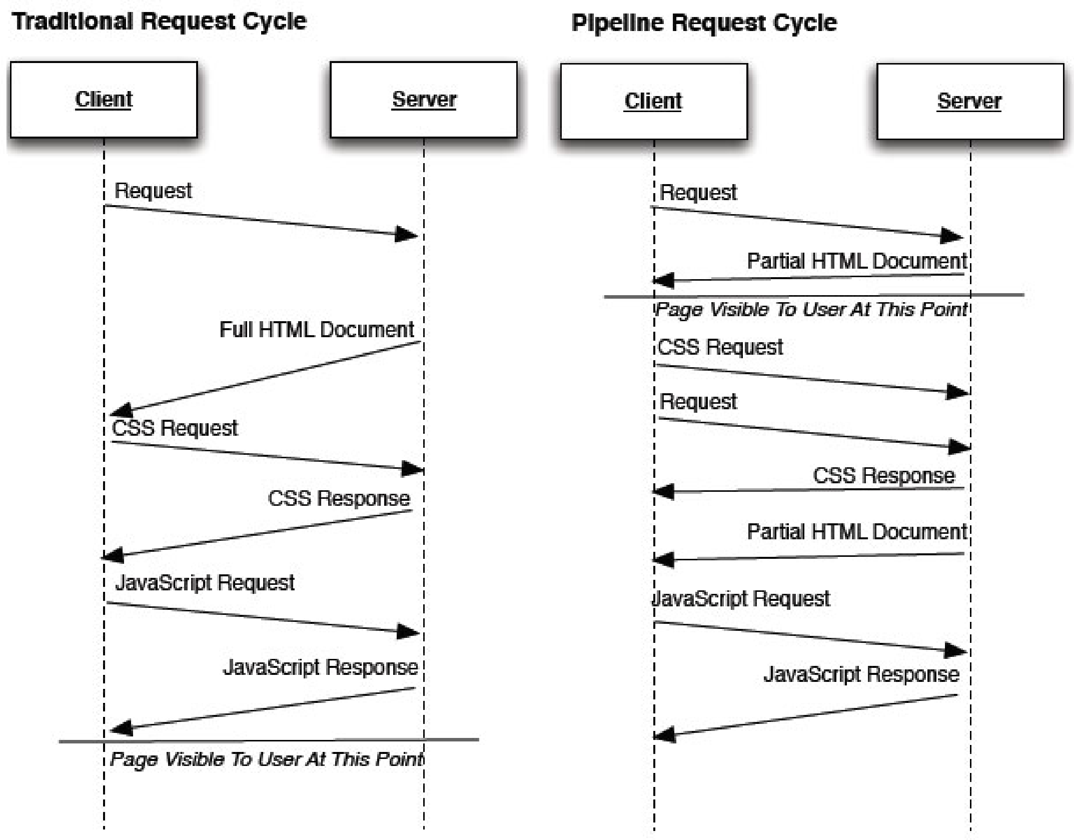

OSI Model
Data
Layer
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Packets
Network
Path Determination and IP (Logical Addressing)
Frames
Data Link
MAC and LLC (Phyiscal addressing)
Bits
Physical
Media, Signal, and Binary Transmission

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 1.2] 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 Tools

## 2.1 Git

This application makes use of Git to store the current codebase, examples, and documentation. 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 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 statitistics make PHP a very good option to build and develop new web technologies and frameworks which will then be available to a large communitiry of software developers.

## 2.3 phpDocumentor

phpDocumentor is a tool with which it is possible to generate documentation from your PHP source code. With this documentation you can provide your consumers with more information regarding the functionality embedded within your source and not only what is usable to them from your user interface. phpDocumentor is heavily inspired by the JavaDoc tool available with the Java SDK. All code documentation is producted through the user of the phpDocumentor tool.

## 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

## 2.5 Chrome

All client side testing and analysis of OpenPipe was performed using the Chrome web browser. Chrome supplies a very advanced toolset for profiling HTTP requests, viewing and editing HTML, and debugging JavaScript. Chrome 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.

# 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.

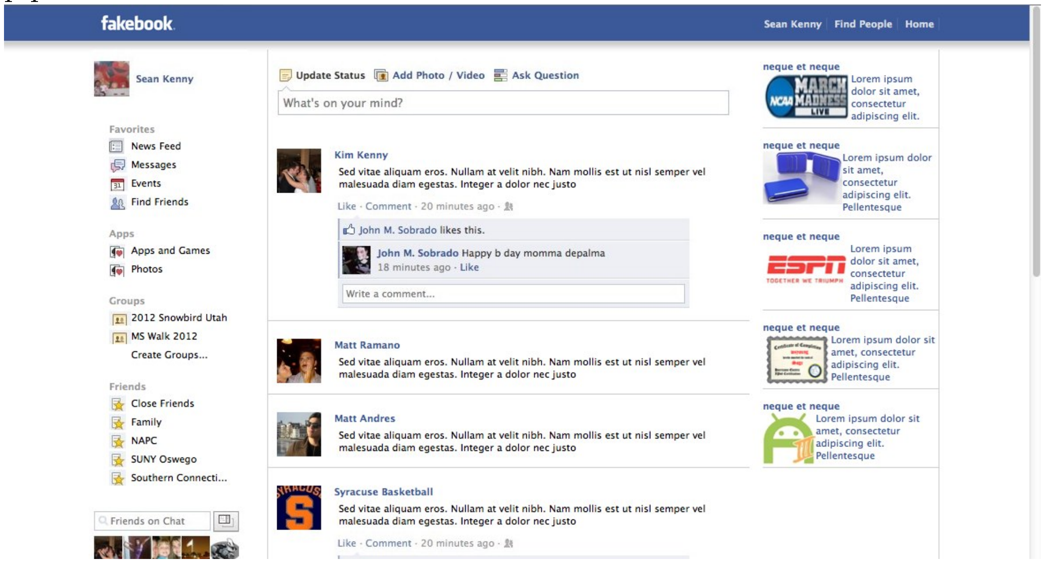
(d) Friends - groupings of friends the user is related to.

(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



Kim Kenny
Sed vitae aliquam eros. Nullam at velit nibh. Nam mollis est ut nisl semper vel malesuada diam egestas. Integer a dolor nec justo

John M. Sobrado likes this.

John M. Sobrado Happy b day momma depalma
18 minutes ago · Like

Matt Ramano
Sed vitae aliquam eros. Nullam at velit nibh. Nam mollis est ut nisl semper vel malesuada diam egestas. Integer a dolor nec justo

Matt Andres
Sed vitae aliquam eros. Nullam at velit nibh. Nam mollis est ut nisl semper vel malesuada diam egestas. Integer a dolor nec justo

Syracuse Basketball
Sed vitae aliquam eros. Nullam at velit nibh. Nam mollis est ut nisl semper vel malesuada diam egestas. Integer a dolor nec justo

neque et neque
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

neque et neque
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque

neque et neque
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque

neque et neque
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque

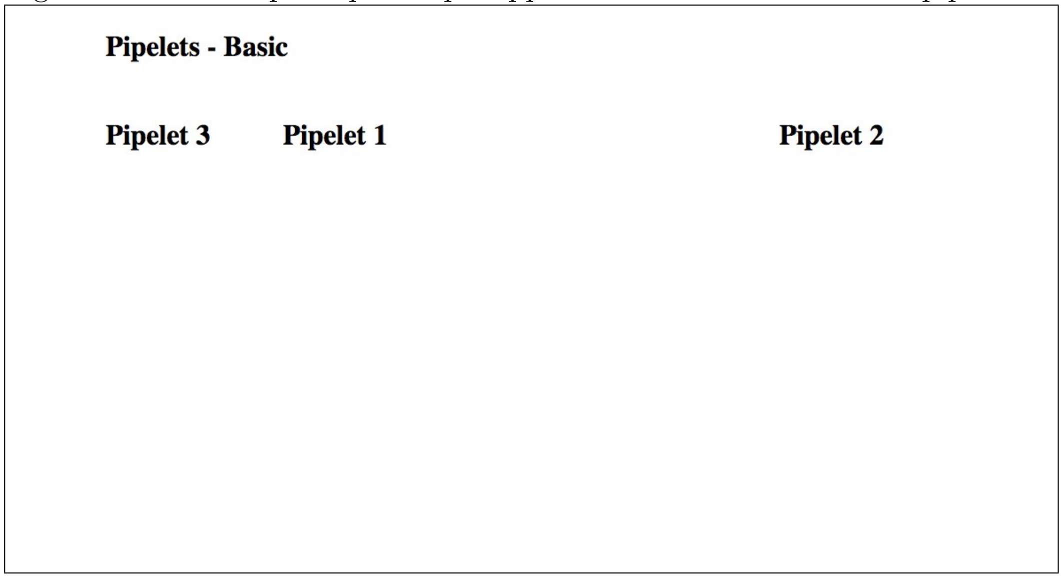
neque et neque
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque

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 simples series of installation steps illustrated in the provided sample application. The main index file ends of looking like figure 3.3.

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 interfaces 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
   ```

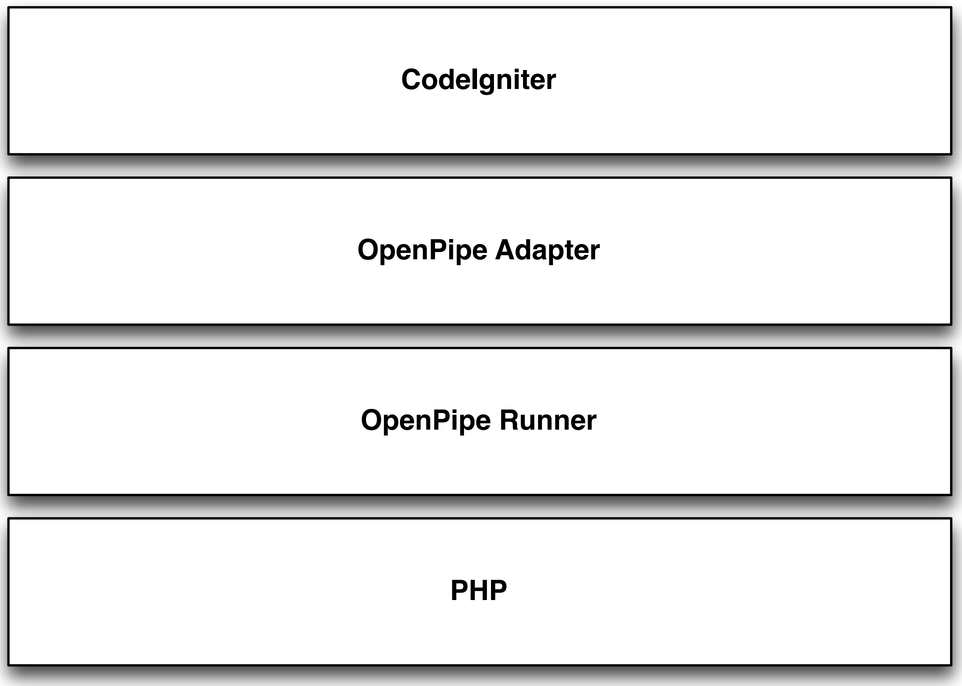
3. **OpenPipe_Adapter** - A pluggable interface which retrieves and returns pipelets components from the underlying architecture. An adapter can be used to interface preexisting PHP application Frameworks with OpenPipe, or whole new OpenPipe centric application frameworks.

   ```
   //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.

Figure 4.1: Server side tecnology stack with OpenPipe components



## 4.2 Client components

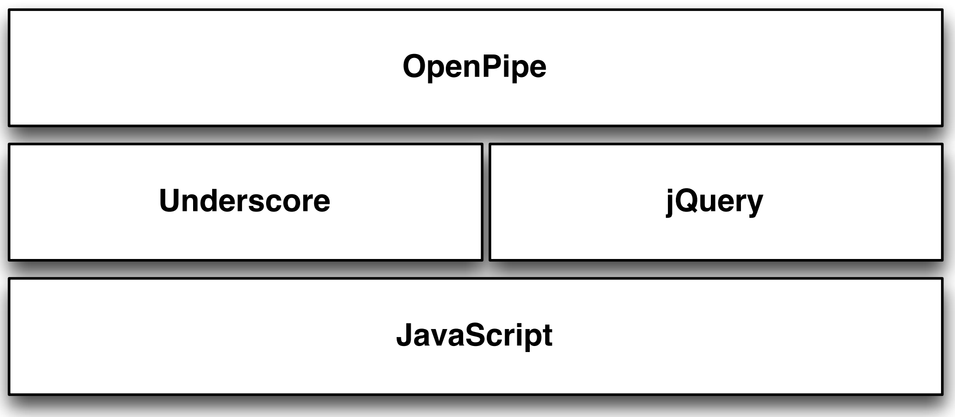
The client is composed of three main layers:

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. Underscore.js s a utility-belt library for JavaScript that provides functional programming support. 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 pipelined HTTP request 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

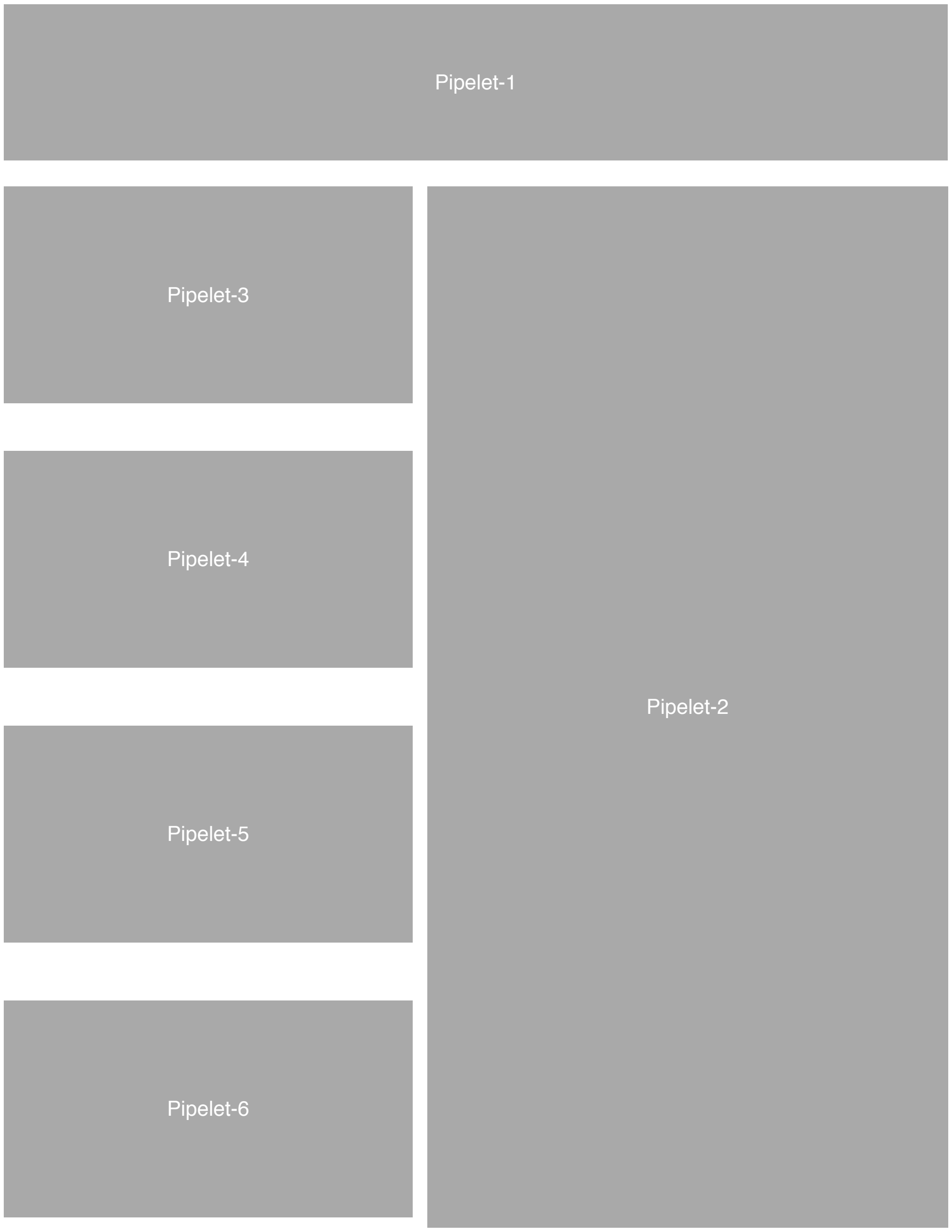
Every pipelined HTTP request contains at least one initial pipelet. This initial pipelet is known as the root pipelet, and is the source for extraction and retrieval for all other pipelets. The root pipelet is special because it defines the overall layout of the page, from which all pipelets will be 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 piplet 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

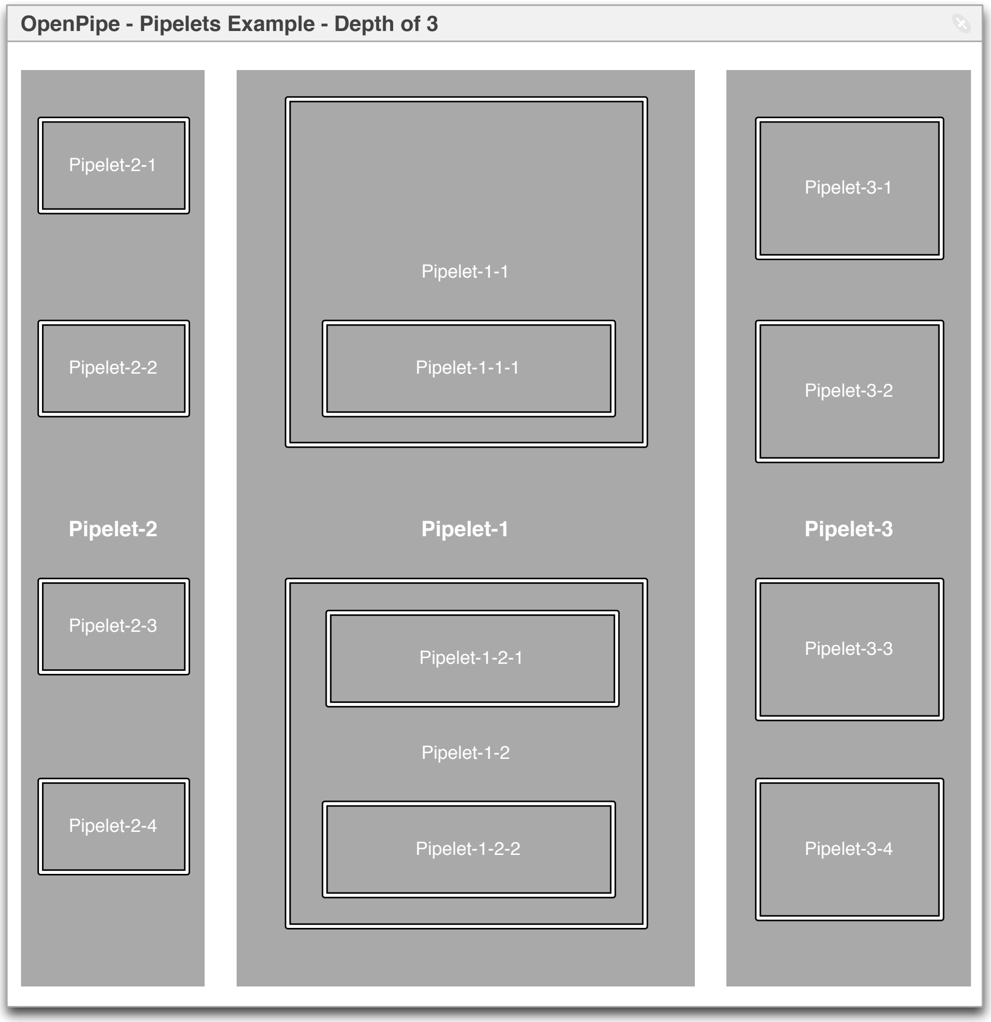


## 5.2 Nested Pipelets

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 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 Open-Pipe 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>
```

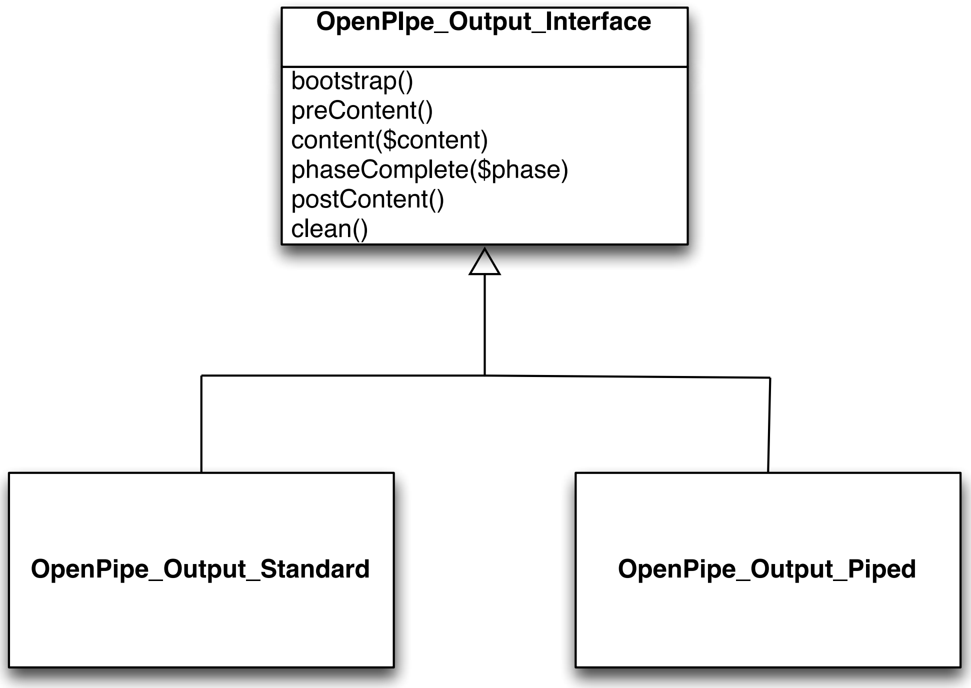
# Chapter 6

# 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:

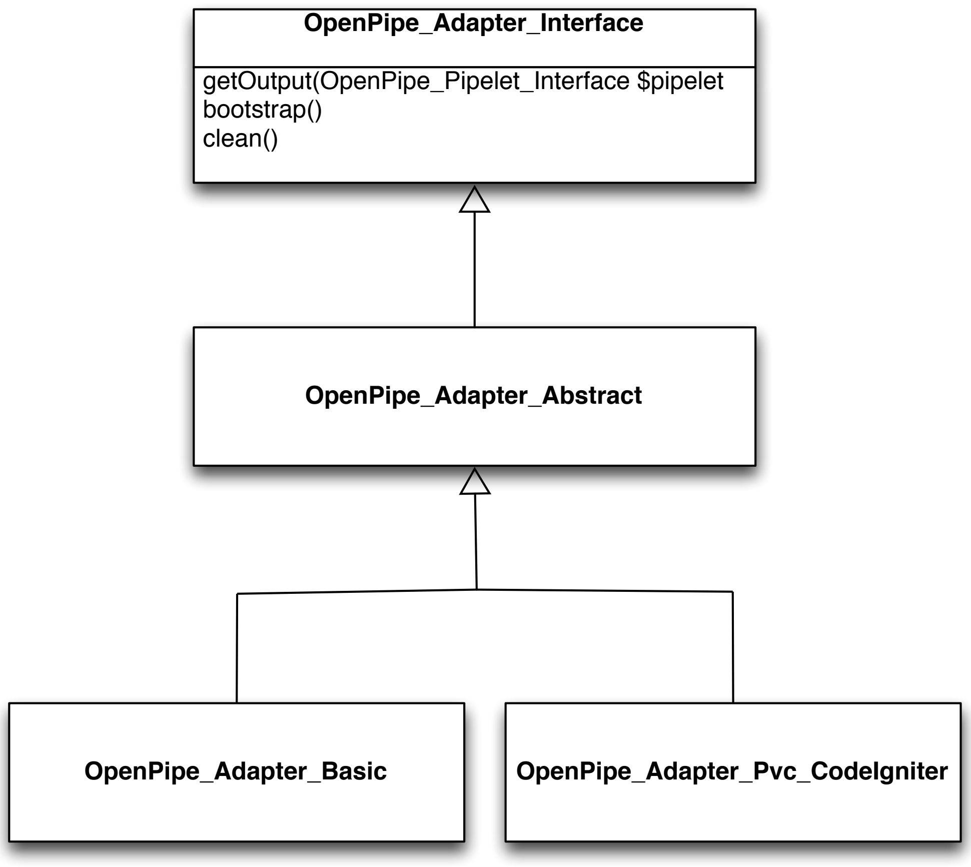
1. **OpenPipe_Output_Piped** - Implementation of an OpenPipe output interface that sends data by way of an HTTP pipeline. This is done by loading the openpipe.js client library and associated libraries. The output handler handles extracting pipelet HTML data, and transmitting it as packed JSON object, 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 not 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


OpenPipe_Adapter_Interface
getOutput(OpenPipe_Pipelet_Interface $pipelet
bootstrap()
clean()
OpenPipe_Adapter_Abstract
OpenPipe_Adapter_Basic
OpenPipe_Adapter_Pvc_CodeIgniter

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 retrieved by the framework adapter. This process continues until all output is retrieved.

# Chapter 7

# Design Patterns

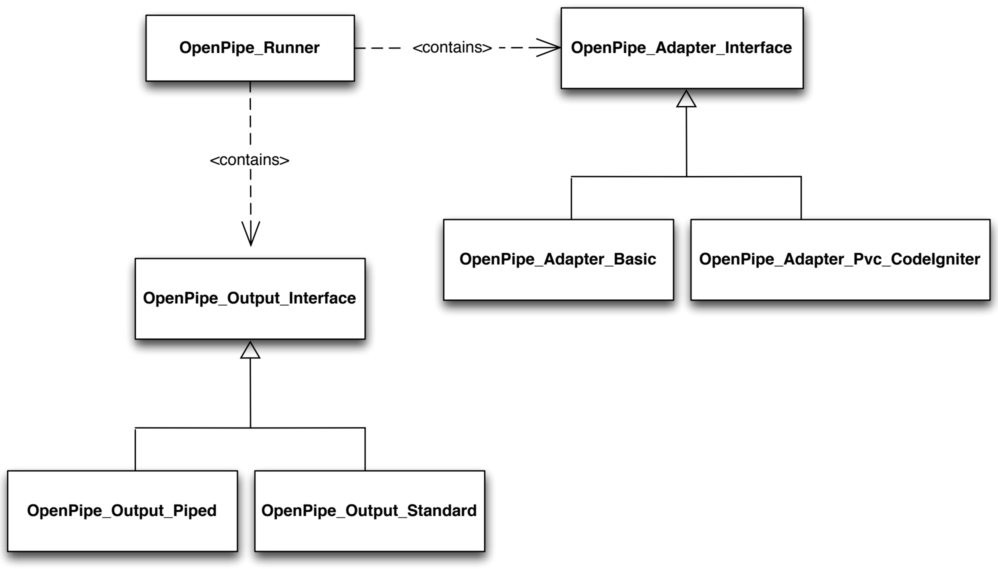
The OpenPipe library utilizes design patterns where appropriate to make the available components easier to comprehend from a conceptial level, and also easier to extend. The patterns explained below were chosen to provide maximum flexibility to the underlying system when integrating with PHP web application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

## 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 utilzed 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 OpenPiper_Runner class when it is instantiated [see figure 7.2].

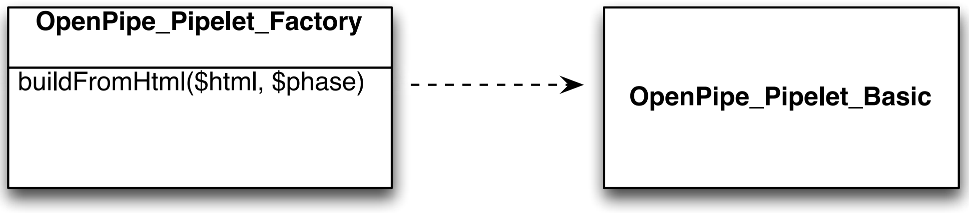
Figure 7.2: Instantiation and running of an OpenPipe_Runner object

```
//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.

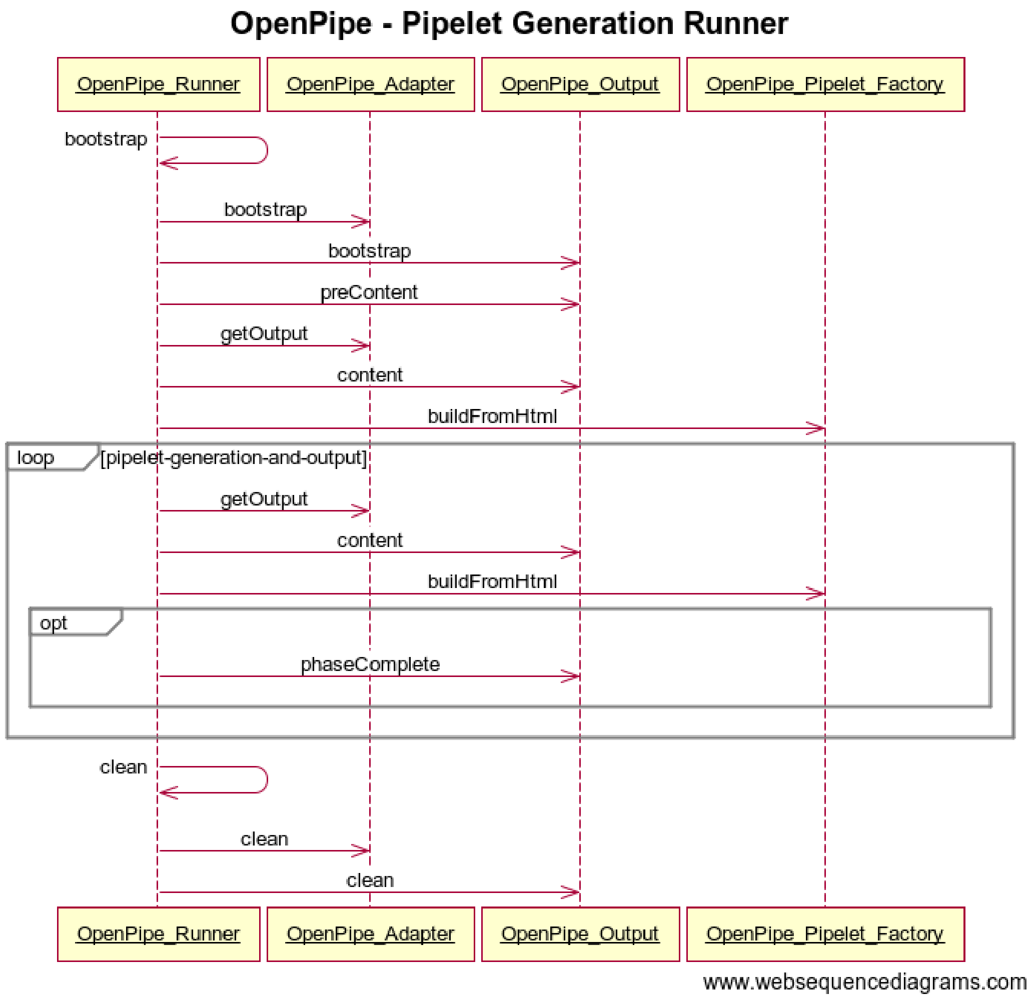
Once the OpenPipe_Runner class has received information from the OpenPipe_Adapter its passes any renderable content to a class that implements the OpenPipe_Output interface. The OpenPipe_Output class handles sending data to the client, and removes any special concerns for how this data is transmitted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

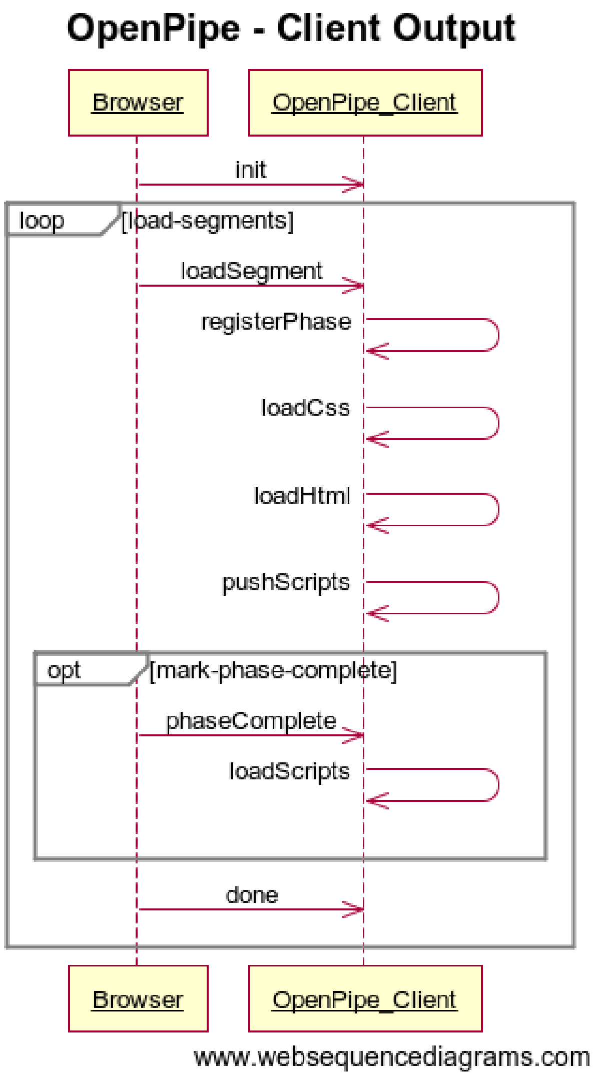


Figure 8.3: OpenPipe client side pipelet load calls

```
<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

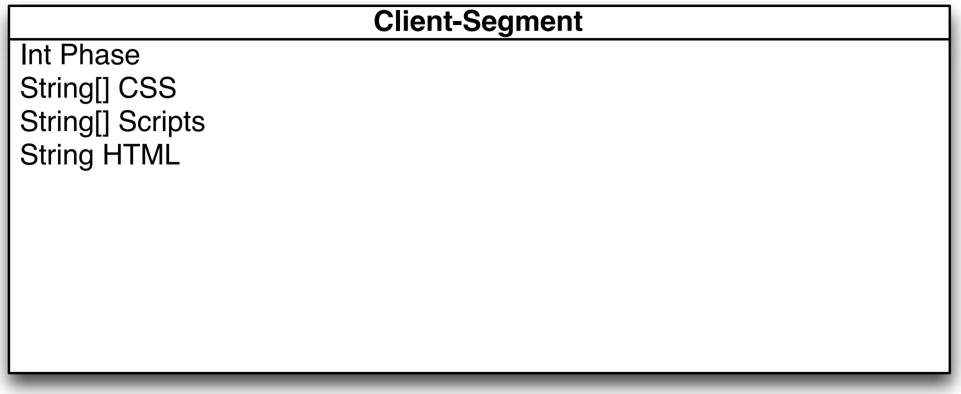
# Data Objects

## 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();
}
```

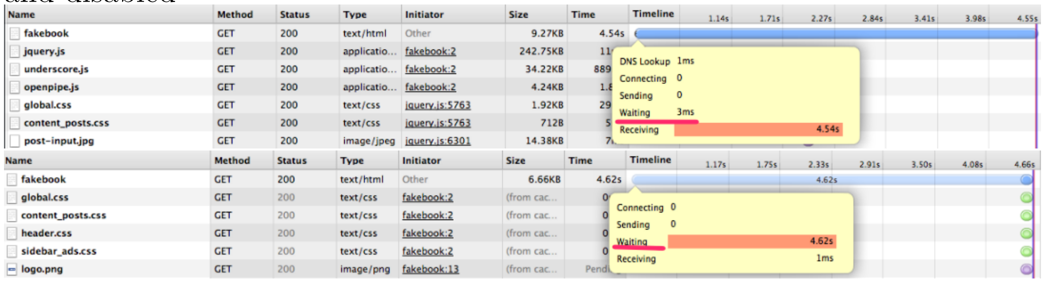
# Chapter 11

# Analysis

## 11.1 Initial Wait Time

Initial wait time was dramatically decreased when when utilizing OpenPipe. In fact since data is returned as soon as the initial page layout is ready. This results to just 3ms for data to be received by the browser. [figures-below]. This is much improved over the wait time for a non piped version of the same page

Figure 11.1: Initial wait times compared in Chrome with OpenPipe enabled and disabled



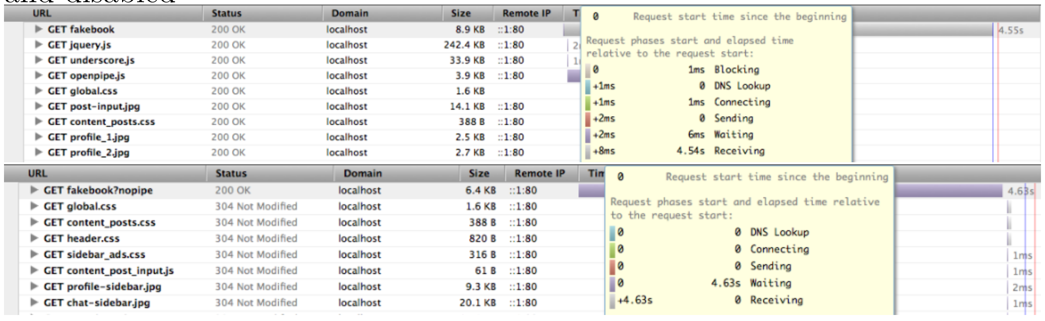
| Name | Method | Status | Type | Initiator | Size | Time | Timeline 1.14s 1.71s 2.27s 2.84s 3.41s 3.98s 4.55s |
|---|---|---|---|---|---|---|---|
| fakebook | GET | 200 | text/html | Other | 9.27KB | 4.54s | |
| jquery.js | GET | 200 | applicatio... | fakebook:2 | 242.75KB | 11 | |
| underscore.js | GET | 200 | applicatio... | fakebook:2 | 34.22KB | 889 | |
| openpipe.js | GET | 200 | applicatio... | fakebook:2 | 4.24KB | 1.8 | |
| global.css | GET | 200 | text/css | jquery.js:5763 | 1.92KB | 29 | |
| content_posts.css | GET | 200 | text/css | jquery.js:5763 | 712B | 5 | |
| post-input.jpg | GET | 200 | image/jpeg | jquery.js:6301 | 14.38KB | 7 | |

| | |
|---|---|
| DNS Lookup | 1ms |
| Connecting | 0 |
| Sending | 0 |
| Waiting | 3ms |
| Receiving | 4.54s |

| Name | Method | Status | Type | Initiator | Size | Time | Timeline 1.17s 1.75s 2.33s 2.91s 3.50s 4.08s 4.66s |
|---|---|---|---|---|---|---|---|
| fakebook | GET | 200 | text/html | Other | 6.66KB | 4.62s | 4.62s |
| global.css | GET | 200 | text/css | fakebook:2 | (from cac... | 0 | |
| content_posts.css | GET | 200 | text/css | fakebook:2 | (from cac... | 0 | |
| header.css | GET | 200 | text/css | fakebook:2 | (from cac... | 0 | |
| sidebar_ads.css | GET | 200 | text/css | fakebook:2 | (from cac... | 0 | |
| logo.png | GET | 200 | image/png | fakebook:13 | (from cac... | Pendi | |

| | |
|---|---|
| Connecting | 0 |
| Sending | 0 |
| Waiting | 4.62s |
| Receiving | 1ms |

Figure 11.2: Initial wait times compared in Firefox with OpenPipe enabled and disabled



| URL | Status | Domain | Size | Remote IP |
|---|---|---|---|---|
| GET fakebook | 200 OK | localhost | 8.9 KB | ::1:80 |
| GET jquery.js | 200 OK | localhost | 242.4 KB | ::1:80 |
| GET underscore.js | 200 OK | localhost | 33.9 KB | ::1:80 |
| GET openpipe.js | 200 OK | localhost | 3.9 KB | ::1:80 |
| GET global.css | 200 OK | localhost | 1.6 KB | |
| GET post-input.jpg | 200 OK | localhost | 14.1 KB | ::1:80 |
| GET content_posts.css | 200 OK | localhost | 388 B | ::1:80 |
| GET profile_1.jpg | 200 OK | localhost | 2.5 KB | ::1:80 |
| GET profile_2.jpg | 200 OK | localhost | 2.7 KB | ::1:80 |

```
0        Request start time since the beginning

Request phases start and elapsed time
relative to the request start:
0            1ms  Blocking
+1ms           0  DNS Lookup
+1ms         1ms  Connecting
+2ms           0  Sending
+2ms         6ms  Waiting
+8ms       4.54s  Receiving
```

4.55s

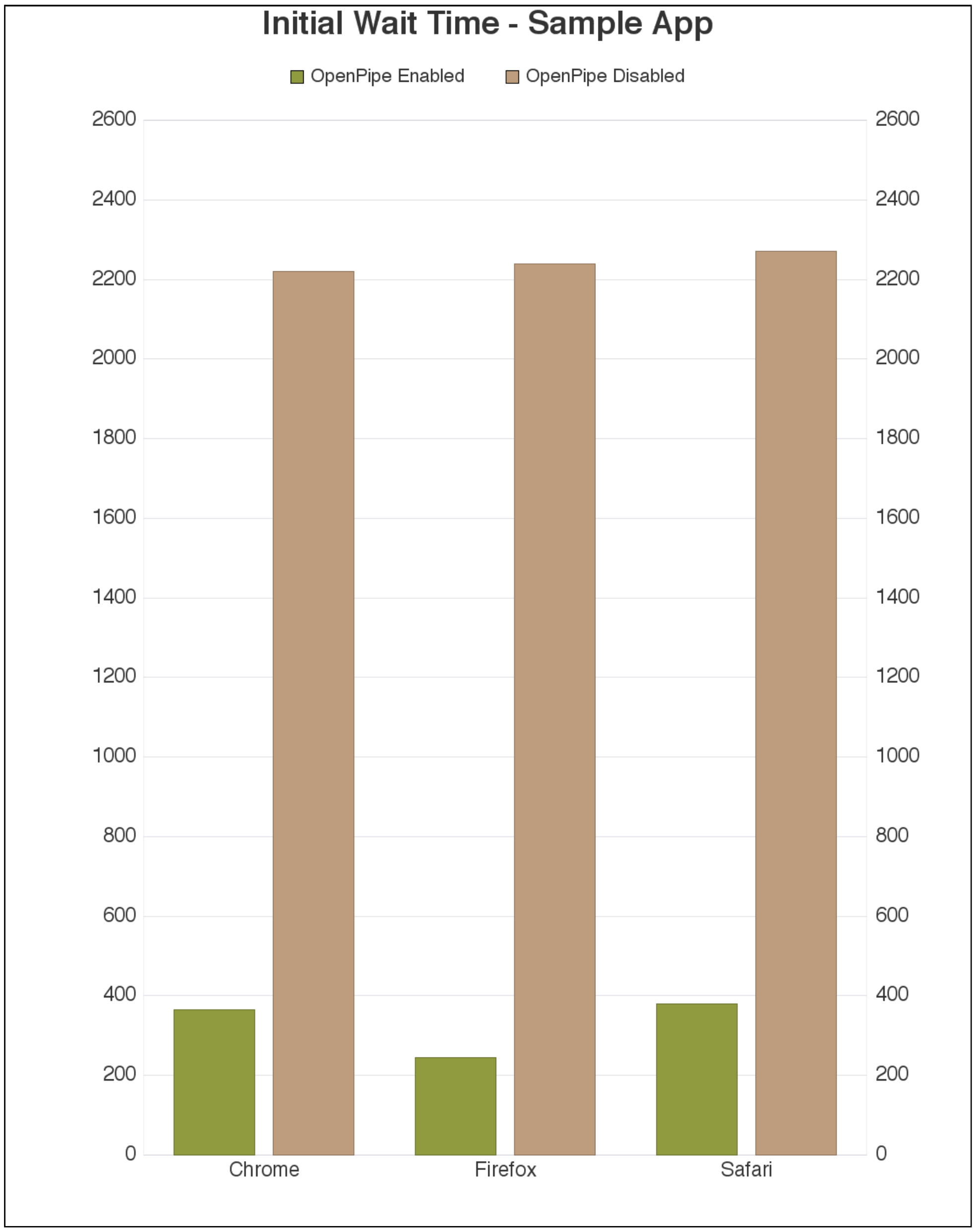
| URL | Status | Domain | Size | Remote IP |
|---|---|---|---|---|
| GET fakebook?nopipe | 200 OK | localhost | 6.4 KB | ::1:80 |
| GET global.css | 304 Not Modified | localhost | 1.6 KB | ::1:80 |
| GET content_posts.css | 304 Not Modified | localhost | 388 B | ::1:80 |
| GET header.css | 304 Not Modified | localhost | 820 B | ::1:80 |
| GET sidebar_ads.css | 304 Not Modified | localhost | 316 B | ::1:80 |
| GET content_post_input.js | 304 Not Modified | localhost | 61 B | ::1:80 |
| GET profile-sidebar.jpg | 304 Not Modified | localhost | 9.3 KB | ::1:80 |
| GET chat-sidebar.jpg | 304 Not Modified | localhost | 20.1 KB | ::1:80 |

```
0        Request start time since the beginning

Request phases start and elapsed time relative
to the request start:
0                 0  DNS Lookup
0                 0  Connecting
0                 0  Sending
0             4.63s  Waiting
+4.63s            0  Receiving
```

4.63s

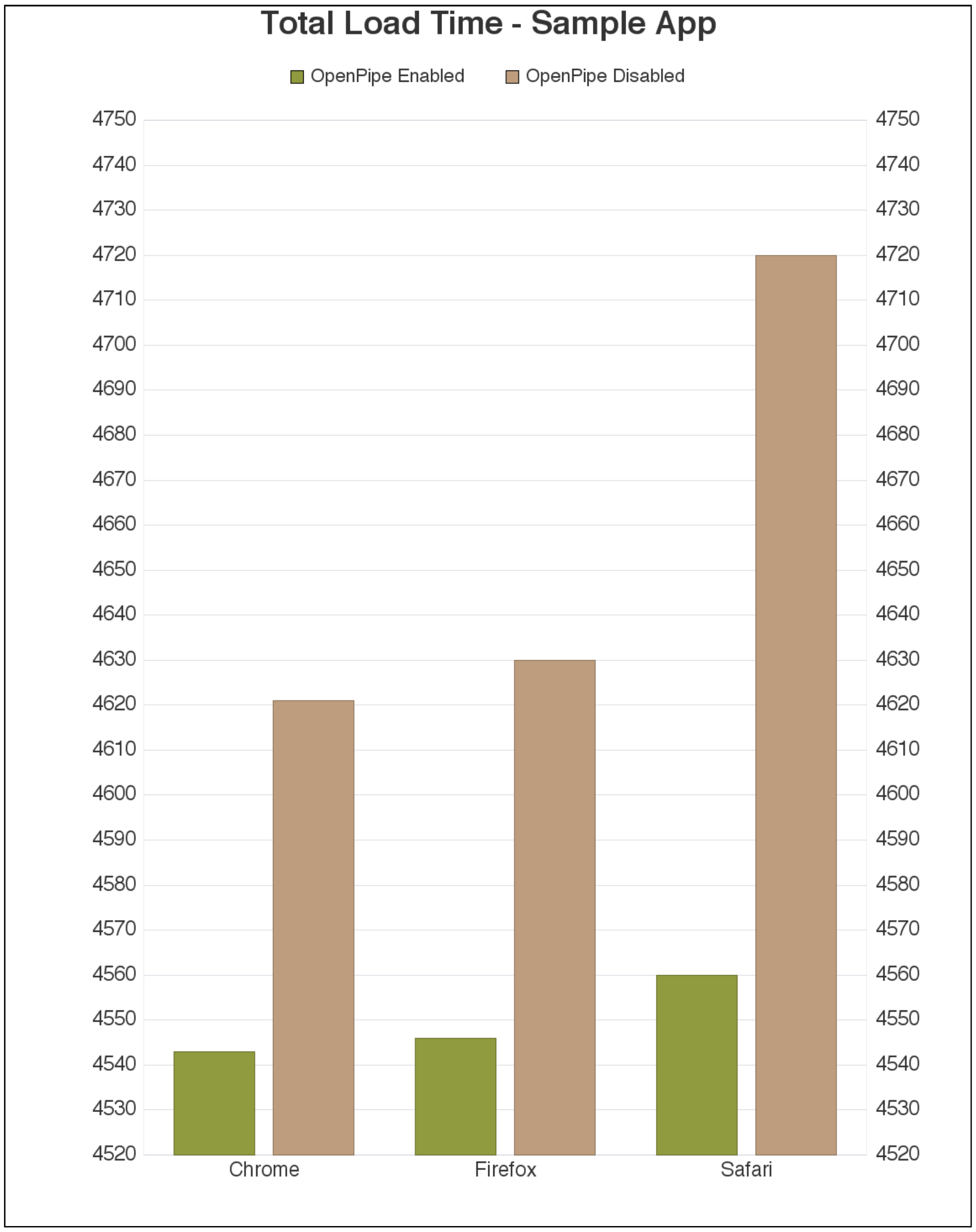
Figure 11.3: Initial wait times for all browsers with OpenPipe disabled and enabled



## 11.2 Total Load Time

Total load time remained similar between piped and non-piped output. This signifies that the additional overhead introduced in using OpenPipe server and client libraries remains relatively insignificant.

Figure 11.4: Initial wait times for all browsers with OpenPipe disabled and enabled



# Chapter 12

# Next Steps

## 12.1 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.

## 12.2 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 developer 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

## 12.3 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"

```php
<?php

require_once('Output/Util.php');
require_once('Adapter/Interface.php');
require_once('Pipelet/Factory.php');

/**
* A runner is the  core object for any OpenPipe based
    application. A runner is responsible for gathering output
    from an OpenPipe_Adapter_Interface based
* adapter and returning to the end client browser as piped
    data objects. Before sending these piped based data
    objects this runner also ensures that the
* end client browser has been setup/instantiated
    appropriately by sending the CORE OpenPipe front end
    JavaScript libraries and the CORE HTML framework
* Once constructed calling this object run() method will
    kickoff the OpenPipe HTTP pipelining process
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Runner {

  /**
```

```
 * The OpenPipe_Adapter_Interface object that is used by
    this OpenPipe_Runner to gather pipelets and load
    individual pipelet data
 * @var OpenPipe_Adapter_Interface
 */
protected $frameworkAdapter;

/**
 * The OpenPipe_Output_Interface object that is used by this
     OpenPipe_Runner to send output data to the browser
 * @var OpenPipe_Output_Interface
 */
protected $output;



/**
 * Constructs an OpenPipe_Runner object that communicated
    with the given OpenPipe_Adapter_Interface based object
 * @param OpenPipe_Adapter_Interface $frameworkAdapter
 * @param OpenPipe_Output_Interface $output
 */
public function __construct(OpenPipe_Adapter_Interface
    $frameworkAdapter, OpenPipe_Output_Interface $output){
  $this->frameworkAdapter = $frameworkAdapter;
  $this->output = $output;
}


/**
 * Is responsible for the ENTIRE OpenPipe HTTP pipelining
    lifecycle - handle all bootstrapping, base client
    library loading, output gathering, output transmission,
    Script, and shutdown
 */
public function run(){
  $this->bootstrap();
  $this->output->preContent();

  //ask the framework for the root output layer (the layout
     !). This contains the starting point for all pipelets
     to get recognized and loaded from
  $layout = $this->frameworkAdapter->getOutput();
  $this->output->content($layout);

```

```
    $phase = 0;
    $pipelets= OpenPipe_Pipelet_Factory::buildFromHtml(
       $layout, $phase);
    $pipeletsQueue = array();

    while(!empty($pipelets)){

      $currentPipelet =  array_shift($pipelets);

      $this->frameworkAdapter->getOutput($currentPipelet);
      $this->output->content($currentPipelet);


      //add piplets contained within the current pipelet to
          the the pipelet queue - the pipelet queue will get
          loaded as part of the next phase
      $pipeletsQueue = array_merge($pipeletsQueue,
          OpenPipe_Pipelet_Factory::buildFromHtml(
          $currentPipelet->getOutput(), $phase+1));

      //once the current pipelets have been completed. Check
          the queue. If the queue is not empty then move batch
           to the pipelets array for processing, and mark the
          current pahse complete
      if(empty($pipelets)){
        $pipelets = $pipeletsQueue;
        $pipeletsQueue = array();

        $this->output->phaseComplete(++$phase);

      }

    }

    $this->output->postContent();
    $this->clean();
  }




  /**
  * Performs bootstrapping of OpenPipe runner object and
      calls the injected OpenPipe_Adapter_Interface bootstrap
      () method at the very end
  */
  protected function bootstrap(){
    $this->frameworkAdapter->bootstrap();
    $this->output->bootstrap();
  }

  /**
  * Performs Script of OpenPipe runner object and calls the
      injected OpenPipe_Adapter_Interface clean() method at
      the very end
  */
  protected function clean(){
    $this->frameworkAdapter->clean();
    $this->output->clean();
  }



}
```

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

```
<?php

/**
* 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
* 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
* 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.
* A pipelet at its core has an ID, phase, and output.
* - 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)
* - A phase is used to determine the timing and priority of a
      pipelet when it is received by an end client browser
* - The output is any data gathered from the OpenPipe adapter
      (utilizing the pipelet id). This data is subsequently
    piped to an end client browser
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Pipelet
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
interface OpenPipe_Pipelet_Interface{

  /**
  * Sets the id of the pipelet (used to determine what
     content to gather from a Pipe Adapter)
  * @param string $id a unique identifier for the pipelet
     that will signify importance to the client adapter and
     allow data to be looked up/generated accordingly
  */
  function setId($id);

  /**
  * Returns the current set pipelet id
  */
  function getId();


  /**
  * Sets the phase of the pipelet (used to determine loading
     priorities and sequences)
  * @param  int|string phase to set - Lower numbers are
     higher priority (1), than higher numbers (999)
  */
  function setPhase($phase);

  /**
  * Return the current set phase number
  */
  function getPhase();


  /**
  * Set the output that has been gathered for this pipelet
     from a Pipelet_Adapter
  * @param string $output the output string that has been
     generated/gathered for this given pipelet
  */
  function setOutput($output);


  /**
  * Return the output that is currently set for the pipelet
  */
  function getOutput();

}
```

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

```
<?php

require_once('Interface.php');

/**
* Abstract implementation of the OpenPipe_Pipelet_Interface.
   Provided basic bindings for all methods defined in the
   interface.
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Pipelet
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
   State University (SCSU).
* @version 1.0.0
*/
abstract class OpenPipe_Pipelet_Abstract implements
   OpenPipe_Pipelet_Interface {

  /**
     * The unique identifier for this pipelet - that
        distinguishes it from all others.
     * @var string
     */
  protected $id;

  /**
     * The numbered phase of the pipelet to signify priority
        low-to-high
     * @var string
     */
  protected $phase;


  /**
     * The output that is potentially gathered and piped as
        output for this pipelet
     * @var string
     */
  protected $output;


  /**
  * Sets the id of the pipelet (used to determine what
     content to gather from a Pipe Adapter)
  * @param string $id a unique identifier for the pipelet
     that will signify importance to the client adapter and
     allow data to be looked up/generated accordingly
  */
  public function setId($id){
    $this->id = $id;
  }

  /**
  * Returns the current set pipelet id
  */
  public function getId(){
    return $this->id;
  }


  /**
  * Sets the phase of the pipelet (used to determine loading
     priorities and sequences)
  * @param int|string $phase to set - Lower numbers are
     higher priority (1), than higher numbers (999)
  */
  public function setPhase($phase){
    $this->phase = $phase;
  }

  /**
  * Return the current set phase number
  */
  public function getPhase(){
    return $this->phase;
  }


  /**
  * Set the output that has been gathered for this pipelet
     from a Pipelet_Adapter
  * @param string $output the output string that has been
     generated/gathered for this given pipelet
  */
  public function setOutput($output){
    $this->output = $output;
  }

  /**
  * Return the output that is currently set for the pipelet
  */
  public function getOutput(){
    return $this->output;
  }


}
```

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

```
<?php

require_once('Abstract.php');

/**
* Provided a basic extension off of the
    OpenPipe_Pipelet_Abstract convenience implementation
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Pipelet
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Pipelet_Base extends OpenPipe_Pipelet_Abstract
    {


  /**
  * Builds the object
  * @param string $id the identifier for the pipelet
  * @param int|string $phase the phase for the pipelet
  * @return OpenPipe_Pipelet_Base new instance
  */
  public function __construct($id, $phase){
    $this->setId($id);
    $this->setPhase($phase);
  }
}
```

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

```
<?php

require_once('Base.php');

/**
* Generated pipelets using factory based methods
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Pipelet
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Pipelet_Factory{


  /**
  * Extracts an array of pipelets from an given HTML document
      (represented as s string)
  * @param string $html An html document represented via
      string
  * @param int|string $phase The current phase of the pipelet
      loading process - This is assigned to any loaded
      pipelets extracted from the first html string parameter
  * @return array all pipelets extracted from the HTML input
      and instantiated as OpenPipe_Pipelet_Base objects
  */
  public static function buildFromHtml($html, $phase){

    //pipelet containers
    $pipelets = array();
    $pipeletGroups = array();

    //setup regex to find pipelets - all pipelets need to
        specify at least a pipelet-id attribute
    preg_match_all('/<.*?pipelet-id.*?>/', $html, $matches,
        PREG_SET_ORDER);

    //for all matches extract the pipelet to its appropriate
        group
    foreach($matches as $match){
      //reset match arrays
      $pipeletIdMatch = array();
      $pipeletPriorityMatch = array();

      //extract pipelet attributes into corresponding match
          arrays
      preg_match('/pipelet-id\=(\'.*?\'|".*?")/', $match[0],
          $pipeletIdMatch);
      preg_match('/pipelet-priority\=(\'.*?\'|".*?")/',
          $match[0], $pipeletPriorityMatch);

      //assign matches to local variables
      $pipeletId = trim(@$pipeletIdMatch[1], '\'"');
      $pipeletPriority = trim(@$pipeletPriorityMatch[1], '\'"
          ');;
      if(empty($pipeletPriority)) $pipeletPriority = 0;

      //construct a pipelet based on extracted information,
          and place in proper group
      if(!empty($pipeletId)){
        $pipeletGroups[$pipeletPriority][$pipeletId] = new
            OpenPipe_Pipelet_Base($pipeletId, $phase);
      }
    }

    //sort each groups array by pipelet id
    foreach($pipeletGroups as $key => $pipeletGroup){
      ksort($pipeletGroups[$key]);
    }
    //now that all groups are accounted for sort by priority
    krsort($pipeletGroups);

    //flatten all the segments to a single array
    foreach($pipeletGroups as $pipeletGroup){
      foreach($pipeletGroup as $pipelet){
        $pipelets[] = $pipelet;
      }
    }

    return $pipelets;
  }

}
```

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

```
<?php

/**
```

```
* An interface defining the output mechanism for OpenPipe.
    This abstraction allows for the implementing class to
    handle individual pipelet output appropriately
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Pipelet
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
interface OpenPipe_Output_Interface {

  /**
  * Allow implementor to setup/output any data before the
      content phase begins
  */
  public function bootstrap();


  /**
  * Called immediately before any content is to be outputted
      via the associated content() method
  */
  public function preContent();


  /**
  * Called when content is ready for output - This content is
       already generated HTML string
  * @param string $content html data
  */
  public function content($content);

  /**
  * 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)
  * @param int $phase the number of the phase to mark
      complete
  */
  public function phaseComplete($phase);


  /**
  * Called immediately after all data has been sent for
      output
  */
  public function postContent();


  /**
  * Allows implementor to do any final cleanup/output - last
     step in the output process
  */
  public function clean();

}
```

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

```
<?php

require_once('Interface.php');
require_once('Util.php');

/**
* Implementation of an OpenPipe output interface that sends
   data via 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
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Output
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
   State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Output_Piped implements
   OpenPipe_Output_Interface {

  /**
  * the web path where the client openpipejs library will
     reside
  * @var string the
  */
  protected $jsPath;


  /**
  * Builds an Piped output object
  * @param string $jsPath the web path to the openpipejs client library
  */
  public function __construct($jsPath){
    $this->jsPath = $jsPath;
  }


  /**
  * Sends an initial string of output to force php and the browser to display piped output immediately
  */
  public function bootstrap(){
    //set a 1024 newline - this forces soutput to the browser to start
    echo str_repeat(" ",1024);
    flush();
  }


  /**
  * 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)
  */
  public function preContent(){
    $header  = "<!DOCTYPE HTML>\n<html><head>";
    $header .= "<script type='text/javascript' src='{$this->jsPath}/libs/jquery.js' ></script>";
    $header .= "<script type='text/javascript' src='{$this->jsPath}/libs/underscore.js'></script>";
    $header .= "<script type='text/javascript' src='{$this->jsPath}/openpipe.js'></script>";
    $header .= "<script type='text/javascript' >op.init();</script>";
    $header .= '</head><body><div pipelet-id="op-container"></div>';

    OpenPipe_Output_Util::echoNow($header);
  }



  /**
  * Extracts and outputs data in an openpipe.js friendly way
```

```
  * @param string $content the content to be piped
      immediately - If CSS/JS is contained within the content
      this will be extracted and handled automatically
  */
  public function content($content){

    if(is_string($content)){
      $id = 'op-container';
      $html = $content;
    }else{
      $id = $content->getId();
      $html = $content->getOutput();
    }

    $css = array_merge(OpenPipe_Output_Util::extractLinkTags(
        $html), OpenPipe_Output_Util::extractStyleTags($html))
        ;
    $js = OpenPipe_Output_Util::extractScriptTags($html);

    $html = str_replace("'", "\\'", $html);
    $css = json_encode($css);
    $js = json_encode($js);


    OpenPipe_Output_Util::echoJsNow("op.load({'id': '$id', '
        html': '$html', 'css': $css, 'scripts': $js});");
  }


  /**
  * Handles a phase complete signal by sending the openpipejs
       phaseComplete command to the client browser
  * @param int $phase the number of the phase that has been
      completed
  */
  public function phaseComplete($phase){
    OpenPipe_Output_Util::echoJsNow("op.phaseComplete($phase)
        ;");
  }


  /**
  * Outputs the closing framework elements for an HTTP
      Pipeline HTML document - sends shutdown (done) method
      for client library and close initially open body and
      html tags
  */
  public function postContent(){
    OpenPipe_Output_Util::echoJsNow('op.done();');
    OpenPipe_Output_Util::echoNow('</body></html>');
  }


  /**
  * This handler is always fresh and so clean clean
  */
  public function clean(){
    //we're all clean!
  }



}
```

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

```
<?php

require_once('Interface.php');
require_once('Util.php');

/**
* 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
* utilizing this output implementation
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Output
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Output_Standard implements
    OpenPipe_Output_Interface {

  /**
  * linear array of style tags extracted from content and
      stored as string data
  * @var array
  */
  protected $styles;

  /**
  * linear array of link tags extracted from content and
      stored as string data
  * @var array
  */
  protected $links;

  /**
  * linear array of script tags extracted from content and
      stored as string data
  * @var array
  */
  protected $scripts;

  /**
  * main html content stored as string and injected piece by
      piece as new content becomes available
  * @var string
  */
  protected $content;


  /**
  * Setup all the variables that will be needed to generate
      proper output
  */
  public function bootstrap(){
    $this->styles = array();
    $this->links = array();
    $this->scripts = array();
    $this->content = '';
  }


  /**
  * because standard output does not send any output until
      the end (clean method) - This method is not needed
  */
  public function preContent(){
    //nothing to do for standard based output
  }


  /**
  * takes content and builds an complete html document piece
     by piece
  * @param string|OpenPipe_Pipelet_Interface $content the
     html content that will have data extracted and assigned
     for final output
  */
  public function content($content){
    if(is_string($content)){
      $id = '';
      $html = $content;
    }else{
      $id = $content->getId();
      $html = $content->getOutput();
    }

    //get the link, style, and script tages in each content
       section
    $this->styles = array_merge($this->styles,
       OpenPipe_Output_Util::extractStyleTags($html));
    $this->links = array_merge($this->links,
       OpenPipe_Output_Util::extractLinkTags($html));
    $this->scripts = array_merge($this->scripts,
       OpenPipe_Output_Util::extractScriptTags($html));

    $this->injectHtml($id, $html);
  }


  /**
  * because standard output does not send any output until
     the end (clean method) - This method is not needed
  * @param int phase not needed
  */
  public function phaseComplete($phase){
    //nothing to do for standard based output
  }


  /**
  * because standard output does not send any output until
```

```
    the end (clean method) - This method is not needed
  */
  public function postContent(){
    //nothing to do for standard based output
  }


  /**
  * Takes all of the gathered output and send the final html
     document as part of this last step
  */
  public function clean(){
    $finalOutput = "<!DOCTYPE HTML>\n<html><head>";

    //put the collected scripts before body close
    foreach($this->links as $link){
      $finalOutput .= $link;
    }

    //put the collected styles in the head;
    foreach($this->styles as $style){
      $finalOutput .= $style;
    }

    $finalOutput .= '</head><body>';
    $finalOutput .= $this->content;


    //put the collected scripts before body close
    foreach($this->scripts as $script){
      $finalOutput .= $script;
    }

    $finalOutput .= '</body></html>';

    echo $finalOutput;
  }



  /**
  * Attempts to inject the given html data into the currently
      recorded data - The point of injection is determined by
      the id provided
  * @param string $pipeletId the identifier for the pipelet
```

```
      that will have html content injected within it
   * @param string $html the content that will be injected
      into the current gathered output
   */
   protected function injectHtml($pipeletId, $html){

     //if the content is currently empty, no injection needs
        to take place. Just set as the content root
     if($this->content == ''){
       $this->content = $html;

     //if we have content, find the injection point and
        perform the string replacement with a regex
     }else{
       $this->content = preg_replace("/(<.*?pipelet-id=(?:\"
          $pipeletId\"|'$pipeletId').*?>)/ms", "\\1 $html",
          $this->content);
     }
   }


}
```

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

```
<?php

/**
* Utility object which provides reusable output based
   services for HTTP Pipeline based systems
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Output
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
   State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Output_Util  {

   /**
   * Given an html string extract the link information from
      the raw data and return
   * @param string $html the html string to extract script
      tags from
   * @return array strings containing link tags found within
      the html string
   */
```

```
  public static function extractLinkTags(&$html){
    error_log('test');
    preg_match_all('/<link.*?\/>/ms', $html, $matches,
        PREG_SET_ORDER);
    $html = preg_replace('/<link.*?\/>/ms', '', $html);

    $links = array();
    foreach($matches as $match){
      $links[] = $match[0];
    }

    return $links;
  }


  /**
  * Given an html string extract the style information from
      the raw data and return
  * @param string $html the html string to extract style tags
       from
  * @return array strings containing style tags found within
      the html string
  */
  public static function extractStyleTags(&$html){
    preg_match_all('/<style.*?>.*?<\/style>/ms', $html,
        $matches, PREG_SET_ORDER);
    $html = preg_replace('/<style.*?>.*<\/style>/ms', '',
        $html);

    $styles = array();
    foreach($matches as $match){
      $styles[] = $match[0];
    }

    return $styles;
  }


  /**
  * Given an html string extract the  information from the
      raw data and return
  * @param string $html the html string to extract script
      tags from
  * @return array strings containing script tags found within
       the html string
  */
  public static function extractScriptTags(&$html){
    preg_match_all('/<script.*?>.*?<\/script>/ms', $html,
        $matches, PREG_SET_ORDER);
    $html = preg_replace('/<script.*?>.*<\/script>/ms', '',
        $html);

    $scripts = array();
    foreach($matches as $match){
      $scripts[] = $match[0];
    }

    return $scripts;
  }




  /**
  * Outputs javascript data in piped format - Piped format
      implies minimized and able to be placed in a pipe
      JavaScript array
  * @param string $output the output data (javascript) to be
      wrapped in a javascript tagged and echoed immediately
  * @param boolean $wrapTags wrap the output in a script
      opening and closing tag
  * @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
  * @param string $paddingCharacter the character that will
      be used if padding must occur
  */
  public static function echoJsNow($output, $wrapTags=true,
      $outputBufferSize=null, $paddingCharacter=' '){
    $output = str_replace("\n", '', $output);
    if($wrapTags === true) $output = '<script type="text/
        javascript">'.$output.'</script>';

    self::echoNow($output, $outputBufferSize,
        $paddingCharacter);
  }


  /**
  * 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();
  }

}
```

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

```
<?php

/**
```

```
* An interface defining an adapter which bridges php based
    applications with OpenPipe. OpenPipe will call the adapter
     to load layouts and pipelets.
* In essence the adapter is responsible for making sure that
    the php based application is instantiated, bootstrapped,
    and run appropriately to obtain
* the request element (either layout or pipelet)
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Adapter
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
interface OpenPipe_Adapter_Interface {

  /**
  * return output from the php application for immediate
     piping
  * @param OpenPipe_Pipelet_Interface $pipelet a pipelet
     which supplies information on
  * @return mixed implementors are free to return what they
     will
  */
  function getOutput(OpenPipe_Pipelet_Interface $pipelet =
     null);

  /**
  * called once during the initialization of an OpenPipe
     runner
  */
  function bootstrap();

  /**
  * called once during the shut down of an OpenPipe runner
  */
  function clean();
}
```

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

```
<?php

require_once('Interface.php');

/**
```

```
* Represents an abstract OpenPipe adapter. As an abstract
    class it provides basic services for obtaining the layout
    (root object for pipelets), and generating
* output for pipelets that are requested. Any object which
    extends this class will implement the getLayout() and
    getContent() methods. This abtract class will
* handle the details in regards to buffering output and
    sending it back to the requesting object.
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Adapter
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
abstract class OpenPipe_Adapter_Abstract implements
    OpenPipe_Adapter_Interface{

  /**
  * Returns output for the given pipelet - Output is web
      content (html, css, javascript) - If Pipelet is null
      then the layout is generated.
  * @param OpenPipe_Pipelet_Interface|null $pipelet if not
      specified then the adapter will generate the pipelet
      layout by default
  * @return string|OpenPipe_Pipelet_Interface given string
      output either generated for layout or a
      OpenPipe_Pipelet_Interface with output set
  */
  public function getOutput(OpenPipe_Pipelet_Interface
      $pipelet = null){

    ob_start();

    if($pipelet === null){
      $this->getLayout();
    }else{
      $this->getContent($pipelet);

    }

    $output = ob_get_contents();
    ob_end_clean();


    if($pipelet !== null){
      $pipelet->setOutput($output);
      return $pipelet;
    }else{
      return $output;
    }

  }


  /**
  * Method should return the layout for the given web request
  * @return string the root layout for all pipelets to be
      derived from
  */
  abstract protected function getLayout();

  /**
  * Method should return the layout for the given web request
  * @param OpenPipe_Pipelet_Interface $pipelet the pipelet to
      get content from
  * @return string the root layout for all pipelets to be
      derived from
  */
  abstract protected function getContent(
      OpenPipe_Pipelet_Interface $pipelet);


  /**
  * This abstract class does not provided any bootstrapping
      logic
  */
  public function bootstrap(){ }

  /**
  * This abstract class does not provided any clean logic
  */
  public function clean(){ }

}
```

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

```
<?php

require_once('Abstract.php');

```

```
/**
* A basic adapter which provides an implementation of the
    abstract adapter class. It simply loads layouts and
    piplets from known directories given during
* the constructor process
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Adapter
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Adapter_Basic extends
    OpenPipe_Adapter_Abstract {

    /**
     * The full path to the layouts php files that will be
         loaded by this adapter
     * @var string
     */
  public $layoutsPath;

  /**
     * The full path to the pipelets php files that will be
         loaded by this adapter
     * @var string
     */
  public $pipeletsPath;

  /**
  * Constructs a new Basic pipe adapter
  * @param string $layoutsPath the full path to the layouts
      php file that will be loaded by this adapter
  * @param string $pipeletsPath the full path to the pipelet
      php files that will be loaded by this adapter
  */
  public function __construct($layoutsPath, $pipeletsPath){
    $this->layoutsPath = $layoutsPath;
    $this->pipeletsPath = $pipeletsPath;
  }

  /**
  * loads a php layout via include()
  * @param string $id the id of the layout to be used - An id
      is the filename without the php extension - For example
      default.php would be default
  * @return void
  */
  protected function getLayout($id='default'){
    include($this->layoutsPath.'/'.$id.'.php');
  }

  /**
  * loads a php pipelet via include()
  * @param OpenPipe_Pipelet_Interface $pipelet the pipelet to
      load content for
  * @return void
  */
  protected function getContent(OpenPipe_Pipelet_Interface
     $pipelet){
    include($this->pipeletsPath.'/'.$pipelet->getId().'.php')
       ;
  }

}
```

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

```
<?php

require_once(dirname(__FILE__).'/../Abstract.php');

//declare global variables that CodeIgniter 2 needs to
    function. This will be called on before including code
    igniter files
$BM; $CFG; $UNI;


/**
* A PMVC adapter which provides an implementation of for
    CodeIgniter 2.x applications to take advantage of piped
    output
* @author Sean Kenny <skenny214@gmail.com>
* @package OpenPipe_Adapter
* @license (c) 2011-2012 Sean Kenny, Southern Connecticut
    State University (SCSU).
* @version 1.0.0
*/
class OpenPipe_Adapter_Pvc_CodeIgniter extends
    OpenPipe_Adapter_Abstract {

  /**
```

```
 * The root path of the currently active CodeIgniter
    application
 * @var string
 */
protected $appRootPath;


/**
 * The file name within the $appRootPath that bootstraps and
     runs a CodeIgniter application
 * @var string
 */
protected $indexFileName;

/**
 * Constructs a new CodeIgniter pipe adapter
 * @param string $appRootPath the root path of the currently
     active CodeIgniter application
 * @param string $indexFileName the file name within the
    $appRootPath that bootstraps and runs a CodeIgniter
    application
 * @return OpenPipe_Adapter_Pvc_CodeIgniter new instance of
    this object
 */
public function __construct($appRootPath, $indexFileName='
    index.pipe.php'){
  $this->appRootPath = rtrim($appRootPath, '/');
  $this->indexFileName = $indexFileName;
}

/**
 * loads a php layout by starting the code igniter index
    file
 */
protected function getLayout(){
  global $BM, $CFG, $UNI;
  include($this->appRootPath.'/'.$this->indexFileName);

}

/**
 * loads a php pipelet via CodeIgniter controller - being
    sure to play nice with output class
 * @param OpenPipe_Pipelet_Interface $pipelet the pipelet to
     be used.
*/
protected function getContent(OpenPipe_Pipelet_Interface
    $pipelet){
  global $BM, $CFG, $UNI;

  $CI = &get_instance();
  $CI->output->set_output('');

  call_user_func_array(array($CI, $pipelet->getId()), array
     ());

  $CI->output->_display();
}

}
```

Listing 13.14: "openpipe.js"

```
//      OpenPipe.js 1.0.0
//      (c) 2011-2012 Sean Kenny, Southern Connecticut State
   University (SCSU).
//      OpenPipe is freely distributable under the MIT license
   .

(function() {

  //debug vars - for timing
  var isDebug = (document.location.search.indexOf('debug=true
     ') != -1);
  var debugInitTime;
  var debugDoneTime;
  var debugLastSegmentLoadTime;


  // Establish the root object, `window` in the browser, or `
     global` on the server.
  var root = this;

  // Save the previous value of the `op` variable.
  var previousOpenPipe = root.op;

  var isFirstSegment = true;

  var lastPhase = 0;
  var phases = [];
  var scripts = [];


  var op = {};

  //init the OpenPipe client - hide pipelets initally (no
      FLOCs)
  op.init = function(){
    //record times for logging
    if(isDebug === true){
      debugInitTime = new Date().getTime();
      debugLastSegmentLoadTime = debugInitTime;
    }

    $("*[pipelet-loading-indicator]").append('<div class="op-
        loading">loading</div>');
    $("*[pipelet-auto-show='true']").hide();
  };

  //load a given segment object into the piplined document
  op.load = function(segment){
    this.loadSegment(segment);

    //record times and output to the log
    if(isDebug === true){
      var debugCurrentSegmentLoadTime = new Date().getTime();
      console.log('SEGMENT "'+segment.id+'" LOAD TIME: '+(
          debugCurrentSegmentLoadTime-debugLastSegmentLoadTime
         ));
      console.log('TIME UNTIL SEGMENT: '+(
         debugCurrentSegmentLoadTime-debugInitTime));
      debugLastSegmentLoadTime = debugCurrentSegmentLoadTime;
    }
  };


  //register a given phase from a segment
  op.registerPhase = function(phase){
    if(typeof(phase) == 'undefined') phase = 0;
    if(typeof(phases[phase]) == 'undefined') phases[phase] =
        phase;
  };

  //mark a phase complete. A completed phase has its deferred
       JavaScript elemetns loaded (and consequently run)
  op.phaseComplete = function(phase){
    lastPhase = phase;
    this.loadScripts(phase);
  };


  //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)
  op.done = function(){
    var that = this;
    _.each(phases, function(phase){
      if(phase > lastPhase) that.phaseComplete(phase);
    });

    if(isDebug === true){
      debugDoneTime = new Date().getTime();
      console.log('TOTAL LOAD TIME: '+(debugDoneTime-
          debugInitTime));
    }

  };



  //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
  op.loadSegment = function(segment){
    if(isFirstSegment === true){
      isFirstSegment = false;
      $("*[pipelet-loading-indicator]").hide();
    }

    this.registerPhase(segment.phase);

    if(typeof(segment.css) != 'undefined') this.loadCss(
        segment.css);
    if(typeof(segment.html) != 'undefined') this.loadHtml(
        segment.html, segment.id);
    $("*[pipelet-id='"+segment.id+"']").show();

    if(typeof(segment.scripts) != 'undefined') this.
        pushScripts(segment.scripts, segment.phase);
  };
```

```



//load html into the given #id'ed element by appending
op.loadHtml = function(html, id){
  if(typeof(id) == 'undefined') id = 'content';

  $("*[pipelet-id='"+id+"']").append(html);
};

//load css into the head of the documents
op.loadCss = function(css){

  var that = this;
  _.each(css, function(css_item){
    $('head').append(css_item);
  });


};



//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)
op.pushScripts = function(scripts, phase){

  var that = this;
  _.each(scripts, function(script){
    that.pushScript(script, phase);
  });

};

//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 execueted)
op.pushScript = function(script, phase){
  if(typeof(phase) == 'undefined') phase=lastPhase+1;
  if(typeof(scripts[phase]) == 'undefined') scripts[phase]
      = [];
  scripts[phase].push(script);
};

```

```
  //loads all the pusheds script for a given phase
  op.loadScripts = function(phase){
    var that = this;
    _.each(scripts[phase], function(script_item){
      jq_script = $(script_item);

      //if this is an external javascript then we make a new
          dom object to house it from the string data
      if(typeof(jq_script.src) != 'undefined'){
        var script = document.createElement('script');
        script.type = jq_script.attr('type') || '';
        script.src = jq_script.attr('src') || '';
        $('body').append(script);

      //if this was just an internal javascript append to
          body and jquery will execute it
      } else {
        $('body').append(script_item);
      }

    });
  };


  //set the root open pipe object
  root['op'] = op;


}).call(this);
```