CSVelte Documentation

Release v0.2

Luke Visinoni

Dec 12, 2016

Contents

CSVelte is a simple yet flexible CSV and tabular data library for PHP5.6+. It was originally written as a pseudo-port of Python's CSV module in early 2008 and was called PCU (PHP CSV Utilities). Unfortunately my time was very limited and after only version 0.3, PCU went dormant. Fast forward to eight years later and I come across my own library in an unrelated Google search. Surprisingly, PCU had gained and then lost, a somewhat respectable user base. So I revived the project, rewrote it from the ground up using solid object-oriented design principles, keeping only the most basic concepts, and renamed it CSVelte (pronounced just like the word svelte).

Note: Why the name CSVelte?

The library was originally called PHP CSV Utilities, which I never particularly liked. So when I revived the project, I decided that if I was going to release a new version, I wanted a new name. I wanted the name to reflect the library's goal of being simple and elegant and still have CSV (comma-separated values) in the name. The word "svelte" means "slender and elegant". So I just added a "C" to the beginning of it and a slender and elegant CSV library was born!

CHAPTER 1

User Documentation

CSVelte's user documentation is organized into three main sections, as outlined below. If you're new to CSVelte, I recommend that you begin here: *Getting Started*.

1.1 User's Guide

The user's guide is broken down into a series of chapters that walk you through CSVelte's features and concepts. Think of it like an instruction manual. It should be read more or less sequentially, as each chapter tends to build on knowledge from previous chapters. Throughout the user's guide you wlll find references and links to the *API Documentation*, which we'll get to next.

1.1.1 User's Guide

The user's guide is broken down into a series of chapters that walk you through CSVelte's features and concepts. Think of it like an instruction manual. It should be read more or less sequentially, as each chapter tends to build on knowledge from previous chapters. Throughout the user's guide you wlll find references and links to the *API Documentation*, which we'll get to next.

Important: From this point forward, for the sake of brevity, I will be leaving the `CSVelte` namespace prefix out when I mention a CSVelte class by name. For instance, rather than saying `CSVelte\IO\Stream`, I will just say `IO\Stream`.

Getting Started

If this is your first time using CSVelte, you've come to the right place. This "getting started" guide will introduce you to the library, as well as to the CSV format. It will then walk you through its installation and introduce you to its most basic concepts.

Introduction

CSVelte is a modern, object-oriented CSV library for PHP5.6+. Its goal is to take the typically tedious, error-prone task of processing and manipulating CSV data, and make it as simple and easy as possible.

A little history

CSVelte's story actually begins all the way back in 2008. I had just begun to feel confident as a PHP developer and wanted to start my own open source PHP library. I didn't want to bite off more than I could chew, so I chose something I felt would be relatively easy. Something I've been working with since the first time I wrote a line of code. I did a quick Google search and found that there really weren't any proper object-oriented CSV libraries for PHP (at that time), so I set to work writing PHP CSV Utilities (or PCU). Over the next six months or so, I pumped out version 0.1, then 0.2, and then at 0.3 things just stalled. I lost interest. Found other things to do. And PCU faded into obscurity.

Fast forward to eight years later and I'm in a similar position as I was in 2008. It's been a little while since I contributed anything open source. Not to mention I currently have an actual *need* for a CSV library, so I stripped PCU down to its core concepts, rewrote it from the ground up, and renamed it CSVelte (pronounced exactly like the word, svelte [1]).

Note: Why the name CSVelte?

The library was originally called PHP CSV Utilities, which I never particularly liked. So when I revived the project, I decided that if I was going to release a new version, I wanted a new name. I wanted the name to reflect the library's goal of being simple and elegant and still have CSV in the name. The word "svelte" means "slender and elegant". So I just added a "C" to the beginning of it and a slender and elegant CSV library was born!

Library scope

So far, the scope of this library has been limited to the basic reading and writing of CSV-formatted data. It also has some format-detection features and a few other goodies. In the not-too-distant future however, I intend to widen that scope considerably to encompass the aggregation, manipulation, and import/exportation of tabular data in general.

There has been a lot of work done in the last several years by various standardization bodies, organizations, and interested individuals, to improve the wild-west nature of the CSV format. Of particular note is the W3C's CSV on the Web Working Group [2] and the work they've put into what they're calling "CSVW", a series of specs and recommendations aimed at improving interoperability between CSV and other tabular data related formats on the web. It is my intention to implement much, if not all of their recommendations by the time CSVelte reaches v1.0.

What is CSV?

It's highly unlikely that you would even be here reading about this library if you weren't already familiar with CSV in some capacity. But what defines CSV as a format? Who invented it? What body governs its standardization? Where can one find a detailed specification that defines the format down to its most mundane detail? Unfortunately, I can't provide you with a satisfying answer to any of those questions. CSV is a very old format. It has been defined and redefined endlessly by any number of organizations and software products over the course of its over forty-year lifespan.

Note: There is an RFC (**RFC 4180**) that documents one very specific *flavor* of CSV, but few would dare call it the *definitive* CSV standard.

---

[1] (of a person) slender and elegant – Google.com definition for "svelte"

[2] The CSV on the Web Working Group is a W3C chartered group of individuals and organizations working towards improving CSV interoperability on the web

CSV as a format

Although CSV is an extremely widely-used format for importing/exporting data, its lack of a unified standard means CSV data out in the wild can vary substantially in its style and format. Exacerbating the problem, CSV (in all its flavors) lacks a standardized method for dictating metadata such as column type, character encoding, locale information such as language and date/time/currency formatting, etc. One can't even rely on a comma being the delimiter character within a CSV file and the name of the format is *comma*-separated values! This can make life very difficult for developers attempting to reliably output and/or consume CSV-formatted data.

CSV in General

For all the reasons I just mentioned, it isn't possible for me to define the CSV format in any specific way. I can only define its general properties. Basically, CSV is a human-readable data interchange format that represents tabular data. That is to say, it represents rows of fields where rows are separated by some form of line terminator character sequence (typically `\n`, `\r` or `\r\n`) and rows are separated by a character called the delimiter (generally this is a comma, but tabs, pipes and semi-colons are often used as well). The number of fields *should* be the same on every row, although this cannot be relied upon. The data may optionally contain a header row, dictating column header names, which should not be processed as a row of data, but rather as labels for each column *within* that data. Fields may contain the delimiter character and/or line breaks, but if they do, they should be enclosed by quotes. If a quoted field itself contains quotes, it is the general rule that it should be escaped by doubling it up. That is, replacing it with two consecutive quote characters.

Beyond that very general description, CSV files can vary substantially. Whitespace before or after a field is typically ignored, although there is no rule stating that it *must* (and in fact, **RFC 4180** specifies that it *must not*). Blank lines are also typically ignored. Sometimes it's acceptable to use a backslash rather than an additional double quote to escape quote characters. Also, quote characters are generally not allowed unless they are escaped and the field containing them is itself enclosed by quotes. Sometimes CSV files contain more than one header row. Sometimes they contain row titles as well as column titles. All of these little quirks are what make the CSV pseudo-format such a *pleasure* to work with (see sarcasm [1]). This is why I wrote CSVelt–to handle as much of this inconsistency and silliness as possible so that you don't have to.

Tip: For a much more reliable and well-informed history, as well as a more detailed and articulate description of the CSV psuedo-format and a wide selection of examples, I refer you to the Wikipedia entry for comma-separated values.

An example

Although I'm certain that you've seen CSV data before, I'll go ahead and show you an example to demonstrate, at the very least, the terminology mentioned above.

Table 1.1: Contacts

id	lastname	firstname	misc	email
1	Visinoni	Luke	A.K.A. "The CSV Master"	luke.visinoni@gmail.com
2	Jones	Davey	Loves the sea	djones@locker.io
3	Kelly	Marge	Marge, Margery, or Margo	margeincharge@mekelly.info
4	Smith	John		john.smith@yahoo.com
5	Doe	Jane	Been missing a while	janeydoeyes@example.com

The contacts table above may be represented in CSV format as follows. It's *delimited* by the comma character, *lines are terminated* by the (usually invisible, but included here for clarity) line feed character, *quoted* by the double-quote

[1] the use of irony to mock or to convey contempt – Google.com definition for "sarcasm"

character, and its quotes are escaped by doubling them up (two consecutive quote characters). It's perfectly acceptable to leave a field blank, as you can see for the "misc" field for "John Smith". Notice though, that a comma was included even though the field was blank. Also of note, fields are only quoted when they contain either the delimiter (comma), a line terminator character (`\n` in this case), or a quote character. This is probably the most common behavior, although technically, any field may be quoted without adversely affecting any potential consuming script/program.

```
id,lastname,firstname,misc,email
1,Visinoni,Luke,"A.K.A. ""The CSV Master""",luke.visinoni@gmail.com\n
2,Jones,Davey,Loves the sea,djones@locker.io\n
3,Kelly,Marge,"Marge, Margery, or Margo",margeincharge@mekelly.info\n
4,Smith,John,,john.smith@yahoo.com\n
5,Doe,Jane,Been missing a while,janeydoeyes@example.com\n
```

It isn't difficult to imagine how the preceeding CSV data could produce the table above. This is because, despite CSV's failings as a standard, it is meant to be human-readable as well as machine-readable. And it succeeds, for the most part, on both counts. Which is why it has stood the test of time and has been around for over forty years.

Will this be on the test?

Fortunately for you, CSVelte will abtract away a lot of these details and you won't have to deal with them. At least not directly. As long as you understand the basic characteristics that define CSV as a format (rows contain fields, fields contain data separated by a delimiter, etc.), you can pretty much forget the rest. In the vast majority of cases, CSVelte will provide sane defaults and/or automatically detect a CSV dataset's formatting parameters for you anyway. But we'll get to that later.

Installation

I recommend that you install CSVelte using Composer, PHP's de facto package manager. If you aren't using Composer, I highly recommend that you check it out. It makes dependency management ridiculously easy and it's used by virtually every modern PHP library and framework in use today. There are other ways to install the library (which I will outline below), but Composer is by far the cleanest and easiest. Not to mention, the most well-tested.

Requirements

As of v0.2, CSVelte requires PHP5.6+. It is my intention to maintain PHP5.6+ support at least until its end-of-life (until it no longer receives updates and security fixes).

Note: CSVelte does not currently require any PHP extensions, but internationalization and localization, as well as character transcoding are features that are on the /roadmap. These features (and probably others as well) will most likely require one or more of the mbstring, iconv, intl and possibly other extension(s) to be installed. So I can't promise the library won't require certain PHP extensions in the future.

Installation

With Composer

If you've never used Composer, you'll want to head over to getcomposer.org and follow the installation instructions on their download page first. Once Composer has been successfully installed, you may use the following command

to install CSVelte within your Composer package/project. First `cd` into your project directory and then issue this command.

```
$ composer require nozavroni/csvelte dev-master
```

Important: CSVelte is currently under heavy development. Once it reaches a stable version, it will be simply a matter of `composer require nozavroni/csvelte`, but for the time being you will need the `dev-master` flag or Composer will complain and refuse to install (or you can lower your minimum-stability setting, which will have the same effect for *all* your project's dependencies).

Direct Download

Danger: Unless you know what you're doing and/or you have a good reason not to, it's **highly** recommended that you install CSVelte using Composer. It isn't particularly difficult to install without Composer, it just allows me to ensure everybody has a consistent installation experience and as a result, you will find it much easier to get help (at least from me) if you install using Composer.

To install CSVelte manually, first download the latest version of CSVelte (currently v0.2) from GitHub. After extracting the contents of the zip or tarball, simply include the `src/autoload.php` file, which will add the `src` directory to PHP's include path [1] and register CSVelte's autoload function [2] for you (obviously you'll need to change `/path/to/csvelte` to wherever the `src` directory is on your system).

```
<?php
require_once "/path/to/csvelte/src/autoload.php";
```

Note: It *should* be as simple as that, but I can't promise it will be. Your mileage may vary. If you're unable to get CSVelte installed, try sending a message to the official mailing list. And if that fails, you can try e-mailing me directly at csvelte@phpcsv.com and I will do my best to get back to you as soon as possible (just keep in mind I'm a busy guy and this is free software).

First Steps with CSVelte

At this point, you should have CSVelte installed and ready to start working with some data! Before we begin though, it may be a good idea to get some terminology out of the way, just so that we're speaking the same language.

First, a few terms

Dataset Because CSVelte can read CSV data from a variety of different sources, it's not technically correct to reference CSV "files" specifically when talking about CSV data. For this reason, this documentation uses the term CSV "dataset" rather than CSV "file" to refer to the contents of any given CSV resource.

Flavor As I've mentioned already, CSV is not the most well-defined format there is. In fact, it's likely one of the worst. There is virtually no end to the different ways in which you might expect a CSV

[1] See include_path ini setting on php.net
[2] See spl_autoload_register function on php.net

dataset to be formatted. In order to bring some order to the chaos, CSVelte defines several attributes that together make up a CSV "Flavor". Attributes such as delimiter, quote character, line terminator, etc. If you and I can agree on a common "flavor" of CSV, we can at least be sure our CSV files are formatted consistently and are therefor compatible.

Taster Another unfortunate side-effect of CSV coming in so many flavors is that you can never really be sure which flavor you're going to get. Making matters worse, the CSV format doesn't natively support meta data of any kind (well, with the possible exception of an optional header row). The only way you can ever 100% reliably determine the flavor of a dataset is to open the file and look at its content yourself. That's what a taster does. It's simply an object that, given a dataset, will analyze (taste) a sample of it, and return a flavor object, each of its attributes set to the taster's educated best guess.

Stream Rather than attempt to write classes for each potential source or destination for CSV data, CSVelte instead relies on the power and flexibility of PHP's native streams functionality. A stream, according to php.net, "is a resource object which exhibits streamable behavior. That is, it can be read from or written to in a linear fashion". CSVelte provides a class called `IO\Stream` which provides an object-oriented interface to this functionality.

Resource In PHP, a resource is a special type of variable used to represent external objects. More specifically, a stream resource is a reference to a streamable data source, or even more specifically, to a specific position within that data source. For example, let's assume we're streaming a file from the local file system. A resource variable in this instance will point to a specific position within your file, changing as you read or seek through it.

I describe what a PHP resource is only so that you can understand the distinction between a PHP resource variable and a `IO\Resource` object within CSVelte. `IO\Resource` is a class used within CSVelte to represent a stream resource. It cannot be read from, it cannot be written to. It doesn't *do* anything. It simply wraps a native PHP stream resource, providing an object-oriented interface and some conveniences such as lazy-opening and the like. Within CSVelte, it's used wherever one would normally expect a native PHP stream resource.

Getting down to business

For the sake of simply writing some code using CSVelte, let's take some common CSV-related use cases and see how CSVelte fares against them.

Producing a two-dimensional array from a CSV dataset

During the initial research phase of writing this library, I did a Google search for "php csv" and a large portion of the results were various PHP message boards with users asking for an easy way to read a CSV file and produce a two-dimensional array containing its data. This is as good a place as any to start.

Let's assume our CSV file is located on the local file system at `/var/www/data/products.csv`. Our first step is going to be to create an `IO\Stream` object capable of reading our CSV file. To do that, we first need to instantiate a `IO\Resource` object using a valid stream URI. Every stream resource URI consists of a scheme followed by `://`, followed by a path or identifier. Since we're accessing a local file, we want the `file://` scheme. So we simply prepend our file path with `file://` to get our stream URI: `file:///var/www/data/products.csv`. Finally, we can use this URI to instantiate a `IO\Resource` object.

```
$resource = new IO\Resource('file:///var/www/data/products.csv');
```

Now that we have a resource object, we can use it to instantiate a stream object, which will give us all the I/O methods we need to read and write data to our local file.

```php
$resource = new IO\Resource('file:///var/www/data/products.csv');
$stream = new IO\Stream($resource);
// you can now ensure the stream object is readable by doing...
$stream->isReadable(); // should return true
```

A few shortcuts

Invoke the resource object

For the sake of brevity in these examples, I'm going to show you a couple shortcuts you can use to reduce the amount of code it takes to get a stream object. First off, once you've instantiated a `IO\Resource` object, simply invoke it as if it were a function and it will return a `IOStream` using your resource object.

```php
$resource = new IO\Resource('file:///var/www/products.csv');
// invoke a resource object as if it were a function to get a stream
$stream = $resource();
```

Skip the resource object

To create a stream object without first instantiating a `IO\Resource`, you can use the `IO\Stream::open()` method, which does it for you. Its signature is very similar to the `IO\Resource` class's constructor.

```php
// use the stream factory method to skip the resource object
$stream = IO\Stream::open('file:///var/www/products.csv');
```

Attention: For the sake of brevity, I will use the latter of these two techniques to create a stream object. But in your own code, you do what works for you.

The file stream wrapper

Although all stream URIs require a valid scheme to identify which stream wrapper is intended, `file` is a special case because it is the default stream wrapper or scheme. For this reason it is optional and may be omitted when constructing a stream URI. This means that our example URI could have just as easily been `/var/www/data/products.csv`. And in fact, from here on out, we will leave out the `file://` portion when we reference stream URIs for the local filesystem.

At this point, we need to instantiate a `Reader` object to read/parse CSV data from the stream object we just created. We already know that our CSV file is formatted using a comma as its delimiter, a line feed as its line terminator, and it has a header row. Let's create a flavor object with those attributes.

```php
$flavor = new Flavor([
    'delimiter' => ',',
    'lineTerminator' => "\n",
    'header' => true
]);
```

Now, using our stream and flavor objects, we can finally instantiate the reader and call `Reader::toArray()` to get our two-dimensional array. Let's put it all together.

```
// create a stream object to read from our local file...
$stream = IO\Stream::open('/var/www/data/products.csv');
if (!$stream->isReadable()) {
    die('Cannot read CSV file.');
}

// now create a flavor object using our known flavor attributes...
$flavor = new Flavor([
    'delimiter' => ',',
    'lineTerminator' => "\n",
    'header' => true
]);

// now we can go ahead and instantiate our reader
$reader = new Reader($stream, $flavor);
// and we have our two-dimensional array!
$array = $reader->toArray();
```

Note: **Why do we need a Reader object** if we already have `IO\Stream`? Doesn't the `IO\Stream` object *read* data from its underlying stream?

Yes it does. But the `IO\Stream` class is designed to be stupid (at least as it relates to CSV data). It only knows how to read bytes from a stream resource. Once the data's been read, its job is done. The `Reader` object takes over at that point, taking plain text data being read to it by `IO\Stream` and applying semantic meaning to it. These are two entirely different kinds of "reading".

What if I don't know the CSV flavor?

The previous example looks simple enough, but what if we *didn't* know anything about our CSV data? What if we *didn't* know ahead of time what the delimiter and line terminator characters are? No big deal! Simply instantiate your reader the exact same way, only this time, omit the flavor parameter. In the absense of an explicit flavor, the reader will use the `Taster` class internally to automatically determine these attributes for us (in other words, it will "taste" the CSV data and tell us its "flavor").

```
$stream = IO\Stream::open('/var/www/data/products.csv');
$reader = new Reader($stream);
$array = $reader->toArray();
```

In the vast majority of cases, the reader will be able to deduce the CSV flavor on its own and this will work just fine. However, if a flavor cannot be determined, an `Exception\TasterException` will be thrown. You can use this to recover from such an error.

```
try {
    $stream = IO\Stream::open('/var/www/data/products.csv');
    $reader = new Reader($stream);
    $array = $reader->toArray();
} catch (Exception\TasterException $e) {
    // this is an extreme action, in your own script you would handle this
    // a bit more gracefully, depending on the situation...
    die("Flavor could not be determined");
}
```

Producing CSV data from a two-dimensional array

Well, I can't in good conscience show you how to convert a CSV file to a PHP array and then not show you how to convert it back! Fortunately it's pretty trivial. Let's assume we have a two-dimensional array containing the following data:

1	Muhammed MacIntyre	3	35	Nunavut	Storage & Organization
2	Barry French	293	68.02	Nunavut	Appliances
3	Barry French	293	2.99	Nunavut	Binders and Binder Accessories
4	Clay Rozendal	483	3.99	Nunavut	Telephones and Communication
5	Carlos Soltero	515	5.94	Nunavut	Appliances
6	Carlos Soltero	515	4.95	Nunavut	Office Furnishings
7	Carl Jackson	613	7.72	Nunavut	Binders and Binder Accessories
8	Carl Jackson	613	6.22	Nunavut	Storage & Organization
9	Monica Federle	643	35	Nunavut	Storage & Organization
10	Dorothy Badders	678	8.33	Nunavut	Paper

Again, our first task is going to be creating an `IO\Stream` object. Only this time, we'll want to prepare it for writing by passing the correct access mode string as the second parameter to `IO\Stream::open()`. We want to create a new file on the local file system at `/var/www/data/inventory.csv` so we'll want to use "w" to open our stream in write mode [1].

```
$stream = IO\Stream::open('/var/www/data/inventory.csv', 'w');
```

Just as with our input stream and its `IO\Stream::isReadable()` method, we can call `IO\Stream::isWritable()` to make sure that our stream is indeed, writable.

```
$stream = IO\Stream::open($resource);
// you can now ensure the stream object is writable by doing...
$stream->isWritable(); // should return true
```

Now that we have an output stream object to write our data for us, we can instantiate our `Writer` object. If you have a specific flavor object, you can pass that to the writer as well. Otherwise it will use the default (outlined by **RFC 4180** [2]). Let's put it all together.

```
<?php
// we'll assume this variable contains our CSV data in an array...
$csv_array = some_func_that_returns_array();

// create stream in write mode...
$stream = IO\Stream::open('/var/www/data/inventory.csv', 'w');
if (!$stream->isWritable()) {
    die('Cannot write to CSV file');
}

// change the flavor a little...
$flavor = new Flavor([
    'delimiter' => "\t",
    'lineTerminator' => "\n",
    'quoteStyle' => Flavor::QUOTE_ALL
]);

// create a writer...
```

[1] File access mode strings are a short (typically 1-3 characters) string containing very concise instructions about how a file or stream should be opened. See fopen file modes for a more detailed explanation.

[2] **RFC 4180** was written in 2005 by Yakov Shafranovich in an attempt to formalize Microsoft Excel's particular flavor of CSV as the official CSV standard

```
$writer = new Writer($stream, $flavor);
// now write our array and we're done!
$writer->writeRows($csv_array);
```

There's more than one way to skin a cat

The two examples provided thus far offer solutions to arguably the two most common use cases involving CSV (for PHP anyway). So you may be asking yourself, "Shouldn't there be quicker, easier ways to do this?". And you'd be right. CSVelte provides shorter, simpler solutions to both these use cases. So why did I show you these verbose solutions rather than the simple ones? Because it's important that you see the entire interface (in all its power and flexibility) before I show you the facades and factory methods that abstract away all that flexibility for brevity and ease of use. For simple tasks like these, it makes no sense to waste keystrokes on instantiating a resource and then a stream and then a reader. But there are vastly more complex problems that CSVelte aims to solve and for them, all this composition suddenly becomes an asset.

In the next section we will explore streams and resources in detail, investigating all the ways we can use them to manipulate, read, and write CSV and tabular data.

Hint: There are methods on the `CSVelte` class that can provide solutions to both these use cases using a single line of code. I refer you to CSVelte's *Facade methods* to find out more.

CSV Streams

Rather than provide you with a whole arsenal of input and output classes for every conceivable source or destination for CSV data, CSVelte takes advantage of the power and flexibility of PHP's native streams functionality, allowing you to instantiate an `IO\Stream` object using any valid stream URI, open stream resource, `SplFileObject`, PHP string, or any object that implements a `__toString()` method.

What is a stream?

> Streams were introduced with PHP 4.3.0 as a way of generalizing file, network, data compression, and other operations which share a common set of functions and uses. In its simplest definition, a stream is a resource object which exhibits streamable behavior. That is, it can be read from or written to in a linear fashion, and may be able to fseek() to an arbitrary locations within the stream.
>
> —php.net definition of streams [1]

Streams provide a unified interface for reading and writing to just about any conceivable source, whether they be local files, HTTP resources, stdin/stdout, the options are virtually endless. CSVelte takes advantage of this amazing flexibility and power by means of its `IO\Stream` class, which simply wraps PHP's native streams functions. Later on you will learn how to create CSV reader and writer objects, both of which delegate I/O functionality entirely to `IO\Stream`.

Warning: Be careful not to confuse PHP streams with `IO\Stream`. These are two separate things. `IO\Stream` is a class defined by the CSVelte library, while PHP streams are a native feature of the PHP language itself. `IO\Stream` was written as an object-oriented API to PHP's native streams.

[1] Succinct definition of PHP streams pulled from PHP's documentation at php.net.

The `IO\Stream` class

`IO\Stream` is a very simple, yet flexible class that allows you to manipulate data from just about any conceivable source, so long as it's supported by PHP's native stream system (although it is possible to write your own custom stream wrappers, that is a topic for another day). The class supports read, write, and seek operations so long as the underlying stream it represents supports these operations. For example, you can perform read operations on an HTTP stream, but you cannot perform write operations on it. The HTTP protocol simply doesn't allow write operations (if it did, the entire internet would descend into a cesspool containing nothing but ads for male enhancement drugs, every nook and cranny completely defaced with drawings of "peepees" and "weewees", not to mention lolcats).

Because you can never really know until runtime whether a particular stream is readable, writable, and/or seekable, `IO\Stream` provides methods that will tell you. You can call `isReadable()`, `isWritable()`, or `isSeekable()` to determine whether those operations are supported on your stream object.

```
<?php
$stream = Stream::open('http://www.example.com/data.csv');
echo $stream->isReadable(); // outputs "true"
echo $stream->isWritable(); // outputs "false"
echo $stream->isSeekable(); // outputs "true" (seekable only for data that has␣
↪already been read)
```

Note: Unless you intend to extend the `IO\Stream` class or you need advanced streams-related functionality/features, you honestly don't really need to know all that much about how it works. At least in regards to its API. All you really need to know (for now) is that it provides a common interface for `Reader`, `Writer` and a few other classes to work with and that those classes delegate all actual I/O functionality to this one class.

Create a stream using an URI

PHP natively offers a multitude of possible stream wrappers [2]. You can stream data using the local file system, FTP, SSL or HTTP, just to name a few. Each stream wrapper works a little differently, so you'll need to consult PHP's streams documentation if you intend to use a stream wrapper not covered here.

Local filesystem

To instantiate an `IO\Stream` object using a local file, simply pass a valid file name (including its path) to the `IO\Stream::open` method (file name may optionally be preceeded with `file://` but it is not required because "file" is the default stream wrapper). You may also optionally pass a file access mode string [3] as a second parameter to tell `IO\Stream` how you intend to use the stream. `IO\Stream` respects the rules specified by each of PHP's available access mode characters, so its behavior should be familiar if you've ever worked with PHP's `fopen` function.

```
<?php
// create a new local file stream object, and prepare it
// for binary-safe reading (plus writing)
$stream = IO\Stream::open('file:///var/www/data.csv', 'r+b');
// or...
// create a new local file stream object, placing the file pointer at the
// end of the file and preparing to append the file
$stream = IO\Stream::open('./data.csv', 'a');
```

[2] PHP defines stream wrappers as "additional code which tells the stream how to handle specific protocols/encodings". See PHP streams documentation for a more complete description.

[3] File access mode strings are a short (typically 1-3 characters) string containing very concise instructions about how a file or stream should be opened. See fopen file modes for a more detailed explanation.

HTTP

Streaming CSV data over HTTP is made trivial with `IO\Stream`. Simply pass in the fully qualified URI to the CSV file and you're all set!

```
<?php
$stream = IO\Stream::open('http://www.example.com/data/products.csv');
```

PHP

The PHP stream wrapper provides access to various miscellaneous I/O streams such as standard input and standard output [4]. You could use this stream wrapper from within a PHP CLI script to stream CSV data directly from the user.

```
<?php
$stream = IO\Stream::open('php://stdin');
```

For more detailed documentation regarding PHP's available stream wrappers and their respective options and parameters, I refer you to the PHP streams documentation at php.net.

Create a stream with additional stream context

Each of PHP's native stream wrappers (HTTP, file, FTP, etc.) has a list of optional stream context options and parameters that can be set to change a stream's context. For instance, the *http* stream wrapper allows you to specify such things as request method, headers, timeout, etc. `IO\Stream` allows you to pass in these parameters as the third argument to its constructor.

To demonstrate how this works, let's assume we have a script called `download_data.php` on a website called example.com. To make the script work, you must send it an HTTP POST request containing query, type, and format attributes. Our query is "active" and our type is "users". The format we want is, obviously, CSV. So, let's take a look at how we might use `IO\Stream` to stream the resulting CSV data.

```
<?php
$stream = IO\Stream::open('http://www.example.com/download_data.php', 'r', [
    'http' => [
        'method'  => 'POST',
        'header'  => 'Content-type: application/x-www-form-urlencoded',
        'content' => 'type=users&query=active&format=csv'
    ]
]);
```

This example is pretty straight-forward, but the point is made. Context parameters can make our `IO\Stream` objects *extremely* flexible and powerful if used correctly. Unfortunately, beyond this brief introduction, stream context parameters are outside the scope of this documentation. If you'd like to learn more about them, please check out the PHP documentation regarding stream context options and parameters (although to be honest, their documentation isn't much better).

[4] Standard input and standard output are preconnected I/O channels, input typically being a data stream going into a program from the user and output being the stream where a program writes its output. See standard streams Wikipedia page for more on stdin/stdout.

The `streamize` factory

CSVelte provides a namespaced stream factory function called `CSVelte\streamize()` for convenience. While the `IO\Resource` and `IO\Stream` classes require very specific instantiation parameters, this function does its best to convert anything you pass to it into a stream object. Let's take a look at a few examples.

Tip: `streamize` is a namespaced function. This means that in order to use it you need to use its fully-qualified name (`CSVelte\streamize`) unless you first import it into your current namespace with a "use" statement.

```
// import the streamize function into your namespace
use function CSVelte\streamize;
// now you can use streamize without its fully-qualified name
$stream = streamize($var);
```

Create a stream from a standard PHP string

Often times you may end up with a PHP string containing CSV data. In this case you can use the `CSVelte\streamize()` factory to get an `IO\Stream` object representing your string. You can then pass the stream object to a `Reader` object to parse its CSV data or to a `Taster` to ascertain what flavor of CSV it contains.

```
<?php
$csv_string = some_func_that_returns_csv_string();
$stream = CSVelte\streamize($csv_string);
```

There is nothing magic about how this works. CSVelte simply places your string into memory using the "php" stream wrapper [5]. It can then be read from just like any other stream resource.

Using an open `SplFileObject` to create a stream

Although CSVelte cannot work with the `SplFileObject` class directly, `CSVelte\streamize()` *can* convert it to a valid `IO\Stream` object, which it understands perfectly.

```
<?php
$file = new \SplFileObject('./files/data.csv', 'r+b');
$stream = CSVelte\streamize($file);
```

Warning: The `SplFileObject` class does not have any way to access its underlying stream resource, so although `CSVelte\streamize()` can accept an `SplFileObject`, it's pretty limited in that it will always open the file in `r+b` (binary-safe read + write) mode, regardless of what mode was used to open the `SplFileObject`. As a result, the internal file pointer will be moved to the beginning of the file.

Create a stream from an existing stream resource

In PHP, a stream is represented by a special variable called a resource. These variables are used throughout PHP to represent references to external resources (a database for instance). Because CSVelte makes such extensive use of PHP's native streams, I have implemented a `IO\Resource` class that represents a stream resource. This allows me

[5] See php://temp on php.net for more on storing temporary data using the "PHP" stream wrapper

to instantiate a stream resource and pass it around without ever actually opening it (until the time it is actually needed– this is called lazy-loading or in this case lazy-opening). It also allows you to instantiate it using an already-open stream resource object as shown below. Just invoke it as if it were a function to get an `IO\Stream` object.

Note: Unfortunately, because PHP7 has reserved the word "resource" for future use, I will need to change the name of `IO\Resource` to something else in my next release (most likely something like `IO\StreamResource` or `IO\Handle`).

If you already have a stream resource that you've opened using `fopen`, you can pass that resource directly to the `IO\Resource` constructor to create a `IO\Resource` object. Then simply invoke it as if it were a function to get a Stream object.

```
<?php
$stream_resource = @fopen('http://www.example.com/data/example.csv', 'r');
if (false === $stream_resource) {
    die("Could not read from stream URI.");
}
$res = new IO\Resource($stream_resource);

// to get a stream object, simply invoke it as if it were a function...
$stream = $res();
echo $stream->getUri(); // prints "http://www.example.com/data/example.csv"
```

Flavors of CSV

How does CSVelte address the extremely loose nature of CSV as a format? It allows the developer define "flavors" of CSV, or in other words, classes or objects representing a particular set of CSV formatting attributes. Flavors can either be defined at runtime by instantiating a `Flavor` class, passing an associative array of CSV formatting attributes to its constructor, or they can be defined at compile time, by extending the `Flavor` class and setting its attributes internally. CSVelte ships with `Flavor` classes representing several of the most commonly-used CSV flavors, but we'll get to that in a minute. First let's go over the various attributes that, together, define a CSV flavor.

Flavor Attributes

header Specifies whether to treat the first row of the dataset as a header row. If `true`, the first row will be ignored by the `Reader` class when iterating over a dataset. Defaults to `null`

delimiter Specifies a single character to be used as the field separator. Defaults to `,`. Other common values are `\t`, and `|`.

lineTerminator Specifies a character or sequence of characters used to terminate each row. Defaults to `\r\n`. Other common values are `\n` and `\r`.

quoteChar Specifies a single character to be used for quoting fields. Defaults to `"`. Other common values are `'` and `` ` ``.

doubleQuote Specifies how to handle quote characters that fall within a quoted string. If set to `true`, two consecutive `quoteChar` characters will be treated as one. Defaults to `true`.

escapeChar Specifies a single character to be used for escaping the delimiter character within an unquoted field or a quote within a quoted field. Defaults to `null` as it is mutually exclusive to `doubleQuote`.

quoteStyle Specifies the types of fields that should be enclosed with `quoteChar`. Value must be one of the following class constants. Defaults to Flavor::QUOTE_MINIMAL.

QUOTE_NONE No fields should be quoted, regardless of data type or contents.

QUOTE_MINIMAL Only fields containing `quoteChar`, `lineTerminator` or `delimiter` should be quoted.

QUOTE_NONNUMERIC Only fields containing non-numeric data should be quoted.

QUOTE_ALL All fields should be quoted, regardless of data type or contents.

Defining a flavor at runtime

```php
<?php
// instantiate a new flavor object, defining its attributes on-the-fly
$flavor = new Flavor([
    'delimiter' => ",",
    'quoteChar' => '"',
    'doubleQuote' => true,
    'quoteStyle' => Flavor::QUOTE_MINIMAL,
    'lineTerminator' => "\n",
]);
```

Tip: To avoid any possibility of producing CSV data written half with commas and half with tabs (or other such nonsense), the `CSVelte\Flavor` class's attributes are immutable. Once it's been instantiated, its attributes cannot be altered. If you find yourself needing to alter a flavor object, just make a copy of it instead, specifying which attributes you'd like changed in the copy.

```php
<?php
$flavor = new Flavor([
    'delimiter' => ",",
    'quoteChar' =>'"',
    'doubleQuote' => true,
    'lineTerminator' => "\r\n"
]);
// cannot do this!! CSVelte will throw an exception
$flavor->quoteStyle = Flavor::QUOTE_NONNUMERIC;

// do this instead...
$newflavor = $flavor->copy([
    'quoteStyle' => Flavor::QUOTE_NONNUMERIC
]);
```

Common Flavors

Although the range of CSV flavors out *in the wild* is virtually limitless, there are definitely certain combinations of these attributes that are most common. The first of them I'll mention, and the only one with an RFC (**RFC 4180**), is the flavor that Microsoft Excel uses when exporting spreadsheets as CSV data. This is the flavor you'll get when you instantiate a `Flavor` object with no arguments. In addition to the default `Flavor` class, CSVelte provides four concrete classes representing common flavors of CSV.

`Flavor\Excel` This is just basically an alias for `Flavor`. It's included simply for clarity and consistency.

`Flavor\ExcelTab` Exactly the same as `Excel`, except with tabs rather than commas as the delimiter.

Flavor\Unix A common flavor of CSV used by non-Microsoft software. Uses Unix-style line endings (LF), uses backslash as the `escapeChar`, and quotes all non-numeric fields.

Flavor\UnixTab Exactly the same as `Unix`, except with tabs rather than commas as the delimiter.

These class work exactly the same way that `Flavor` does, except that they are preset to a different set of attributes. And just as you can override attributes using the default flavor class, so you can with these.

```
<?php
$excelPipe = new Flavor\Excel([
    'delimiter' => '|'
]);
$excelPipeQuoteAll = $excelPipe->copy([
    'quoteStyle' => Flavor::QUOTE_ALL
]);
```

Defining your own common flavors

If there is a particular flavor of CSV you find yourself using all the time, try extending `Flavor`. Any attributes you don't override in your class will remain their default value.

```
<?php
// my custom flavor uses semi-colons rather than commas to delimit fields
// it also uses old mac-style line endings, doubles up quote characters to escape␣
↪them,
// and quotes all fields no matter what!
class MyCustomFlavor extends Flavor
{
    public $delimiter = ';';
    public $lineTerminator = "\r";
    public $escapeChar = null;
    public $doubleQuote = true;
    public $quoteStyle = self::QUOTE_ALL;
}
```

But what do I do with it?

As I've explained, the `Flavor` class allows you to define a particular set of formatting attributes for CSV. But what then? Knowing a particular set of formatting attributes for CSV does you no good without some data to apply it to. This is where the reader and writer classes come in. And I promise, we will get to them very soon.

Auto-detecting CSV Flavor

If you know in advance what *Flavors of CSV* you're working with, the `Flavor` class is going to work great for you. But what if you don't? Does the CSV format have some way of telling the developer what flavor of CSV it's written in? Unfortunately, it doesn't. But CSVelte does. Any time you read CSV data using the `Reader` class, it will attempt to determine the flavor of its provided dataset automatically. The upshot being that in the overwhelming majority of cases, you can point a `Reader` object at some CSV data and it will just work.

How does it work?

It's actually magic. I figured out how to do magic.

Hint: Is it really magic?

Yes.

Using flavor auto-detection

Using the auto-detect feature is so easy, you won't even know you're using it. Any time you instantiate a `Reader` object without explicitly providing a flavor object, the library will analyze a sample of the data you're trying to read and build a flavor object *for* you, behind the scenes.

```
<?php
// flavor will be quietly inferred from a sample of "products.csv"
$reader = new Reader(IO\Stream::open("./files/products.csv"));
```

Note: There will be a more detailed explanation of this when we get to the section on *Reading CSV Data*. But for now, all you need to know is that CSVelte will *always* try to figure out the CSV flavor on its own unless you explicitly provide one.

Analyzing CSV data manually

Behind the scenes, CSVelte's auto-detect feature ultimately boils down to a single method of the `Taster` class. To manually run the CSV analyzer (flavor taster), you must instantiate a `Taster` object, passing it a readable stream object.

```
<?php
// create an input stream object that points to a CSV file
$csv = IO\Stream::open('./data/products.csv');

// now, using that stream object, instantiate your taster
$taster = new Taster($csv);

// finally, you can "lick" the CSV data to discern its particular "flavor"
$flavor = $taster->lick();
```

This will work for the overwhelming majority of datasets, but if your data is too uniform or your sample too small, the taster object will issue an exception. The exception's message will contain a short explanation of why the taster failed to produce a flavor object, along with an error code. This allows your script to recover from such a failure and rather than display some arcane error page, perhaps prompt your end-user to provide this information or failing that, just proceeding with a sane default. Let's see what that might look like.

```
<?php
// this time we wrap our tasting code in a try/catch
// block for more graceful error recovery
try {
    $csv = IO\Stream::open('./data/products.csv');
    $taster = new Taster($csv);
    $flavor = $taster->lick();
} catch (Exception\TasterException $e) {
    // log exception or something...
    my_exception_log_function($e);
    // flavor could not be determined, so lets use a sane default...
```

```
    $flavor = new CSVelte\Flavor([
        'lineTerminator' => PHP_EOL
    ]);
}
// proceed with data processing...
$reader = new CSVelte\Reader($csv, $flavor);
```

Reading CSV Data

Instantiating a `Reader` object

Before we can instantiate a `Reader` object, we must first instantiate a readable `IO\Stream` object. Let's assume we want to read a local CSV file located at `/var/www/inventory.csv`, and formatted according to the default flavor.

```
// first instantiate a readable stream object...
$stream = IO\Stream::open("/var/www/inventory.csv");
// then pass it to the reader
$reader = new Reader($stream);
```

We could also have used CSVelte's reader factory method to do the same thing.

```
$reader = CSVelte::reader("/var/www/inventory.csv");
```

Tip: Any class that implements the `Contract\Readable` interface can be read by the `Reader` object. This means you can write your own custom "readable" class if you're so inclined. You aren't in any way restricted to just the stream class provided by CSVelte.

Specifying CSV Flavor

If you know in advance what flavor of CSV you're working with, you can pass a `Flavor` object, or an associative array of flavor attributes as the second parameter to the reader's constructor [1].

```
// create readable stream
$in = IO\Stream::open("./data/purchases.csv");

// explicitly pass a flavor object to the reader
$reader = new Reader($in, new Flavor([
    'delimiter' => '|',
    'lineTerminator' => "\n",
    'quoteStyle' => self::QUOTE_ALL,
    'escapeChar' => '//',
    'doubleQuote' => false,
    'header' => true
]));

// or...

// pass an associative array of flavor attributes
```

[1] Any function or method that accepts a `Flavor` object will also accept an associative array of flavor attributes. The two are often interchangeable.

```
// the reader will convert it to a flavor object internally
$reader = new Reader($in, [
    'delimiter' => '|',
    'lineTerminator' => "\n",
    'quoteStyle' => self::QUOTE_ALL,
    'escapeChar' => '//',
    'doubleQuote' => false,
    'header' => true
]);
```

Taking the Pepsi challenge

Omitting the flavor parameter when instantiating a reader object tells CSVelte you want it to automatically detect the CSV flavor. It will use the `Taster` class to analyze a sample of your CSV dataset and provide its best guess as to what its flavor is. This applies whether you instantiate the reader manually or you use the factory method. All this happens behind the scenes and is completely transparent unless something goes wrong, in which case you can expect an `Exception\TasterException`.

Iterating using foreach

`Reader` implements PHP's built-in `Iterator` interface [2], allowing the use of foreach to loop over each row in the dataset. At each iteration, the key will refer to the current line number, while the value will contain a `Table\Row` object.

```
<?php
foreach (CSVelte::reader('./data/inventory.csv') as $line_no => $row) {
    do_something_with($row, $line_no);
}
```

Filtering/skipping certain rows

Although you could loop over every row in a CSV file, and place if/elseif/else branches directly inside the body of your foreach loop, like the following:

```
<?php
$reader = new Reader(IO\Stream::open('./data/products.csv'));
foreach ($reader as $line_no => $row) {
    if (isset($row[2]) && strlen($row[2]) > 10) {
        continue;
    }
    if (isset($row[5]) && (int) $row[5] <= 1000) {
        continue;
    }
    if (empty($row[8])) {
        continue;
    } elseif (isset($row[8]) && $row[8] == 'false') {
        continue;
    }
    // now we can do something with $row
    do_something_with($row);
}
```

[2] see Iterator interface at http://php.net/manual/en/class.iterator.php

This approach feels cluttered. A much cleaner, and clearer way to do this would be to filter out these rows using anonymous functions as filters. The reader object can accept any number of `Callables` [3] to filter out these rows instead. Let's see how this might look.

```
<?php
$reader = CSVelte::reader('./data/products.csv');
$reader->addFilter(function($row) {
    return ($row[2] < 10);
})->addFilter(function($row) {
    return ($row[5] > 1000);
})->addFilter(function($row) {
    return (!empty($row[8]) && $row[8] != 'false');
});
// now we can simply loop over our filtered reader and our unwanted rows
// will be filtered out for us automatically
foreach ($reader->filter() as $line_no => $row) {
    do_something_with($row);
}
```

Warning: As I've mentioned several times, CSVelte is still in its infancy. Its API and many other things about it are not yet stable. Several features can't even be called complete yet. Reader filtering is one such incomplete feature. There is currently no way to remove or alter filters once they've been added to the reader. If you need to change the filters you've added to the reader in any way, you will need to completely reinstantiate the reader from scratch. In the future there will be ways to remove filters after they've been added. In fact the reader filter feature(s) will likely change quite a bit before CSVelte reaches stability at version 1.0 so use them (and CSVelte in general) at your own risk until then.

Working with rows

When looping through CSV data using `Reader` and `foreach`, you will have access to a `Table\Row` object at each iteration. You can use this object to access the row's fields in various ways as well as to loop through its fields using `foreach` just as you did with the reader object.

```
<?php
$reader = new Reader(IO\Stream::open('./data/products.csv'));
foreach ($reader as $line_no => $row) {
    foreach ($row as $col_no => $field) {
        // now do something with $field
    }
}
```

Row indexing

By default, rows will be indexed numerically, starting at zero. This means that in order to work with a particular column's value within a row, you will need to know what its numeric index will be. Let's assume we're working with the following data:

[3] see Callable type-hinting at http://php.net/manual/en/language.types.callable.php

Table 1.2: ./data/great-albums.csv

0	1	2	3
Lateralus	Tool	2001	Volcano Entertainment
Wish You Were Here	Pink Floyd	1975	Columbia
The Fragile	Nine Inch Nails	1999	Interscope
Mezzanine	Massive Attack	1998	Virgin
Panopticon	ISIS	2004	Ipecac

The table above will represent our CSV data. The first row represents the index number for each column. So, let's take a look at how we might interact with such a dataset using `Reader` and `Table\Row`.

```php
<?php
$reader = new Reader(IO\Stream::open('./data/great-albums.csv'));
foreach ($reader as $line_no => $row) {
    // for the first row, this will print:
    // "One of my favorite albums is Lateralus by Tool."
    printf("One of my favorite albums is %s by %s.\n", $row[0], $row[1]);
}
```

Indexing with the column headers

If your CSV data contains a header row, you can use column header values as your row indexes (rather than the numerical indexing shown above). Let's use the same dataset from before, only this time we'll add a header row.

Table 1.3: ./data/great-albums.csv

Album	Artist	Release Year	Label
Lateralus	Tool	2001	Volcano Entertainment
Wish You Were Here	Pink Floyd	1975	Columbia
The Fragile	Nine Inch Nails	1999	Interscope
Mezzanine	Massive Attack	1998	Virgin
Panopticon	ISIS	2004	Ipecac

In order to be able to use column header values rather than numeric indexes, you must first ensure that your `Flavor` object has its header attribute set to true. This will tell the reader that the first row in the dataset should be considered the header row, rather than treated as data.

```php
<?php
$flavor = new Flavor([
    'header' => true
]);
$reader = new Reader(IO\Stream::open('./data/great-albums.csv'), $flavor);
foreach ($reader as $line_no => $row) {
    // now we can use column headers rather than numeric indexes
    $album = $row['Album'];
    $artist = $row['Artist'];
    // or, if you like, you can still use the numerical indexes as well
    $year = $row[2];
    $label = $row[3];
}
```

Attention: You must remember to use the exact spelling and capitalization that the header row uses. "Album" is not the same as "album". If you use the latter, it will trigger an exception. You don't want that. In the future, I will likely relax this to allow any capitalization but for now, you must remember to use the header value exactly as it appears in the data.

Writing CSV Data

The `Writer` class

The `Writer` class is the main workhorse for writing CSV data to any number of output sources. Instantiating a writer is very similar to instantiating a `Reader` object. You simply instantiate any class that implements the `Contracts\Writable` interface [1] and use that to instantiate a writer object.

Let's assume we want to write a CSV file called `./data/products.csv`. We would need to instantiate a `IO\Stream` object pointing to that file, making sure to supply the "w" access mode to open the file in write mode (you could also use the "a" mode if you want to append a stream rather than write a new one [2]). Let's see how that looks:

```
// we use "w" access mode string to open stream in write mode
$out = IO\Stream::open('./data/products.csv', 'w');
$writer = new Writer($out);
```

You could do the same thing using CSVelte's writer factory method.

```
$writer = CSVelte::writer('./data/products.csv');
```

Setting the flavor

If you want to use a specific flavor of CSV (rather than the standard `Flavor` class), you can do so by passing a `Flavor` object (or an associative array of flavor attributes) as the second parameter to the writer's constructor and the writer will write CSV according to your specified flavor. See *Flavors of CSV* for more on flavors and formatting.

```
$out = IO\Stream::open('./data/products.csv', 'w');
$flavor = new Flavor(['delimiter' => "\t"]);
$writer = new Writer($out, $flavor);
```

As I mentioned before, it is also acceptable to pass an associative array to the writer class rather than an `Flavor` object to override the default flavor's attributes. Here, we will override the standard delimiter, which is a comma, and use a tab character instead.

```
$out = IO\Stream::open('./data/products.csv', 'w');
$writer = new Writer($out, ['delimiter' => "\t"]);
```

We can shave off even *more* keystrokes by using CSVelte's writer factory method to generate our writer for us. As long as you don't need some custom stream output or something, this is the quickest and easiest way and it works just fine. Again, you can pass either a `Flavor` object *or* an associative array of flavor attributes as the second parameter.

[1] CSVelte only ships with one class that implements the `Contract\Writable` interface, and that is `IO\Stream` – see *CSV Streams* for more about that class

[2] See the fopen file modes section on php.net for more possible stream/file access modes.

```php
$writer = CSVelte::writer('./data/products.csv', new Flavor\ExcelTab);

// or...

$writer = CSVelte::writer('./data/products.csv', ['delimiter' => "\t"]);
```

Writing a single row

Once you've instantiated a `Writer` object, you can use the `Writer::writeRow()` method to write CSV line-by-line. You simply pass it an array or traversable (just be sure it contains the correct number of fields in the correct order [3]).

```php
<?php
$out = IO\Stream::open('./data/products.csv', 'w');
$writer = new Writer($out);
// you can pass an array...
$writer->writeRow(['one', 2, 'three', 'fore']);
// or any traversable object, so long as it contains the correct number of fields...
$writer->writeRow(new ArrayIterator(['five', 'sicks', '7 "ate" 9', 10]));
```

Depending on the `Flavor` object you use, this should output something along the lines of:

```
one,2,three,fore
five,sicks,"7 ""ate"" 9",10
```

Writing multiple rows

If you have a two-dimensional array or any other traversable tabular data [4], you can pass it to the `Writer::writeRows()` method to write multiple rows at once.

```php
<?php
$out = IO\Stream::open('./data/albums.csv', 'w');
$writer = new Writer($out);
$writer->writeRows([
    ['Lateralus', 'Tool', 2001, 'Volcano Entertainment'],
    ['Wish You Were Here', 'Pink Floyd', 1975, 'Columbia'],
    ['The Fragile', 'Nine Inch Nails', 1999, 'Interscope']
]);
```

Depending on your `Flavor` attributes, this should output something along the lines of:

```
Lateralus,Tool,2001,Volcano Entertainment
Wish You Were Here,Pink Floyd,1975,Columbia
The Fragile,Nine Inch Nails,1999,Interscope
```

[3] Every row in a CSV dataset should contain the same number of fields in the same order. For full description of CSV format, see "*What is CSV?*"

[4] Tabular data, in this context, refers to any traversable two-dimensional data structure. Each set of traversables must contain the same number of fields, in the same order or an exception will be thrown

Setting the header row

CSV files allow an optional header row to designate labels for each column within the data. If present, it should always be the first row in the data. You can go about writing your header row one of two ways. You can do it implicitly, by simply making sure the first row you write is your header row, like so:

```
$out = IO\Stream::open('./data/albums.csv', 'w');
$writer = new Writer($out);
$writer->writeRows([
    ['Album', 'Artist', 'Year', 'Label'],
    ['Lateralus', 'Tool', 2001, 'Volcano Entertainment'],
    ['Wish You Were Here', 'Pink Floyd', 1975, 'Columbia'],
    ['The Fragile', 'Nine Inch Nails', 1999, 'Interscope']
]);
```

But if you prefer to be explicit, like I do, you may use the `Writer::setHeaderRow()` method. Just be sure to call it before writing any other rows to your output.

```
$out = IO\Stream::open('./data/albums.csv');
$writer = new Writer($out);
$writer->setHeaderRow(['Album', 'Artist', 'Year', 'Label']);
$writer->writeRows([
    ['Lateralus', 'Tool', 2001, 'Volcano Entertainment'],
    ['Wish You Were Here', 'Pink Floyd', 1975, 'Columbia'],
    ['The Fragile', 'Nine Inch Nails', 1999, 'Interscope']
]);
```

This does the exact same thing as the first approach did, only it's more explicit and more clear to any programmer who comes along later, what you are trying to do.

Danger: You must be careful not to call `Writer::setHeaderRow()` after data has already been written to the output source. That is to say, after any calls to `Writer::writeRow()` or `Writer::writeRows()`. This will trigger an `Exception\WriterException`.

Using reader and writer together

The reader and writer classes are very useful by themselves, but when you combine them, you can really start to see the power and usability of CSVelte. Let's take a look at a few ways you can use `Reader` and `Writer` together to accomplish common tasks.

Reformatting by changing flavor

As I mentioned before, `Writer::writeRows()` accepts either an array of arrays or any tabular data structure. Instances of the `Reader` class, by design, fall within the second category. This means that you can instantiate a reader object and pass it to `Writer::writeRows()` as a means to either filter out certain rows, change its flavor (formatting), or both. Let's take a look at a few examples.

```
<?php
// create our reader object, allowing it to automatically determine CSV flavor
$reader = CSVelte::reader("./data/albums.csv");

// now create a writer object, passing it an explicit flavor we want to reformat to
```

```php
$writer = CSVelte::writer("./data/albums.tsv", new Flavor\ExcelTab());

// now you can simply pass the reader object to writeRows to get a tab-delimited file
$writer->writeRows($reader);
```

Filtering out unwanted rows

As demonstrated in *Reading CSV Data*, you can use the `Reader::addFilter()` method to attach any number of anonymous functions to your reader to filter out unwanted rows. You can then iterate your filtered reader using the `Reader::filter()` method. Again, because `Writer::writeRows()` can accept any traversable tabular data structure, you can pass the return value of `Reader::filter()` to `Writer::writeRows()` to write a new CSV file, less your unwanted rows.

```php
// create our reader object
$reader = CSVelte::reader("./data/albums.csv");
// this will filter out all but 90's albums
$reader->addFilter(function($row) {
    return ($row['Year'] >= 1990 && $row['Year'] < 2000);
});

// now create a writer object, pointing to a new "90s-albums.csv" file
$writer = CSVelte::writer("./data/90s-albums.csv");

// now you can simply pass the reader object to writeRows to get a CSV
// file with only 90's albums from the original CSV file
$writer->writeRows($reader->filter());
```

Facades and Factories

My goal with CSVelte is to provide a simple, yet powerful and flexible object-oriented interface for CSV data processing and manipulation. Sometimes though, in order to provide the level of flexibility I desire, simplicity and ease of use suffer. But, being the hard-headed fella that I am, rather than give up any of that power or flexibility, I provide you with facades and factory methods which instead abstract away that flexibility in favor of ease of use.

Note: If you're wondering why I showed you these *last*, it's because I wanted you to learn how to instantiate readers and writers manually before resorting to factory/facade methods exclusively. Using these methods eliminates the possibility for you to use a custom `IO\Stream` object, effectively eliminating a *huge* chunk of functionality just to save a few keystrokes. Don't get me wrong, there's nothing wrong with shortcuts, just don't overdo it.

Factory methods

CSVelte provides, via the `CSVelte` class, several methods for quickly and easily generating common objects. Let's take a look at them.

static `reader` (*$uri*[, *$flavor* = *null*])

Parameters

- **`$uri`** (*`string`*) – A fully-qualified stream URI or the path to a local file.
- **`$flavor`** (*`array|Flavor`*) – An explicit flavor object or an array of flavor attributes to pass to the reader.

Returns Reader object for specified stream URI or file

Throws `Exception\IOException`

Reader factory method. Provides a shortcut for creating a `Reader` object.

```
<?php
foreach (CSVelte::reader("./data/inventory.csv") as $line_no => $row)
↪{
    do_something_with($row);
}
```

static writer (*$uri*[, *$flavor* = *null*])

Parameters

- **$uri** (*string*) – A fully-qualified stream URI or the path to a local file.
- **$flavor** (*array|Flavor*) – An explicit flavor object or an array of flavor attributes to pass to the reader.

Returns

Writer object for specified stream URI or file

Writer factory method. Provides a shortcut for creating a `Writer` object.

```
<?php
$data = some_func_that_returns_tabular_data();
CSVelte::writer("./data/reports.csv", [
    'delimiter' => "\t",
    'lineTerminator' => "\n"
])->writeRows($data);
```

Facade methods

The `CSVelte` class also provides the following facade methods [1].

static export (*$uri*[, *$data*[, *$flavor* = *null*]])

Parameters

- **$uri** (*string*) – A fully-qualified stream URI or the path to a local file.
- **$data** (*mixed*) – Anything that can be passed to Writer::writeRows()
- **$flavor** (*array|Flavor*) – An explicit flavor object or an array of flavor attributes to pass to the reader.

Returns

The number of rows written to the output stream.

Writer facade method. Provides a shortcut for exporting tabular data to a stream or local file.

```
<?php
$data = some_func_that_returns_tabular_data();
CSVelte::export("./data/reports.csv", $data, [
    'delimiter' => "\t",
```

[1] A facade, in programming, is the abstraction of a complex and/or verbose interface into a more concise, simpler one. See "facade design pattern".

```
    'lineTerminator' => "\n"
]);
```

Hint: Although there isn't currently a `CSVelte::import()` method (to produce a two-dimensional array from a CSV dataset), you may combine the `CSVelte::reader()` and `CSVelte::toArray()` methods to approximate this functionality.

```
CSVelte::reader("./data/products.csv")->toArray();
```

Tip: It's a trade-off, like almost any design decision in programming. And it's one you'll have to make when you go to write your own code using CSVelte. The question you need to ask yourself is, "Do I need power and flexibility, or do I just need to get shit done?"

1.2 API Documentation

The API documentation is an exhaustive listing of every class, method, property, exception, and function defined by CSVelte as well as definitions for each of them. This portion of the documentation is generated automatically by a piece of software called ApiGen. It's definitely the most comprehensive of the three documentation sections, but not the easiest to read. It is recommended that you read the *User's Guide* to learn and the *API Documentation* as a reference.

1.2.1 API Documentation

Danger: Unfortunately, the software I use to auto-generate my API docs relies on a library that isn't fully compatible with PHP5.6. It chokes when you attempt to import namespaced functions. I am working on either forking the library and fixing it myself or finding another solution. But in the mean time, my API docs will not be available. Feel free to e-mail me directly if you have any questions that you would normally consult the API docs for. Thank you for your patience.

1.3 Tutorials

While the user's guide and API docs focus on explaining and defining CSVelte's features and classes (respectively), the tutorials focus on its actual use. Each tutorial addresses a specific use case, providing a somewhat opinionated solution and walking you through it step by step. Each tutorial can be downloaded in its entirety, including working source code and instructions.

1.3.1 Tutorials

Caution: *I'm sorry!* I haven't had time to write any tutorials yet. I'm only one man and I only have so much time. Tutorials will be added as I find the time to write them. Thank you for your patience!

While the user's guide and API docs focus on explaining and defining CSVelte's features and classes (respectively), the tutorials focus on its actual use. Each tutorial addresses a specific use case, providing a somewhat opinionated solution and walking you through it step by step. Each tutorial can be downloaded in its entirety, including working source code and instructions.

Import/Export to Database

Symbols

R