# **Tabletop.js** (gives spreadsheets legs)

Tabletop.js takes a Google Spreadsheet and makes it easily accessible through JavaScript. With zero dependencies! If you've ever wanted to get JSON from a Google Spreadsheet without jumping through a thousand hoops, welcome home.

Tabletop.js easily integrates Google Spreadsheets with templating systems and anything else that is hip and cool. It will also help you make new friends and play jazz piano.

[![Build Status](https://travis-ci.org/jsoma/tabletop.svg?branch=master)](https://travis-ci.org/jsoma/tabletop)

### Like how easy?

Step One: make a Google Spreadsheet and "Publish to Web."

Step Two: Write a page that invokes Tabletop with the published URL Google gives you.

```
function init() {
  Tabletop.init( { key: 'https://docs.google.com/spreadsheets/d/0AmYzu_s7QHsmdDNZUzRlYldnWTZCLXdrMXlYQzVxSFE/pubhtml',
                   callback: function(data, tabletop) {
                       console.log(data)
                   },
                   simpleSheet: true } )
}
window.addEventListener('DOMContentLoaded', init)
```

Step Three: Enjoy your data!

```
[ { name: "Carrot", category: "Vegetable", healthiness: "Adequate" },
  { name: "Pork Shoulder", category: "Meat", healthiness: "Questionable" },
  { name: "Bubblegum", category: "Candy", healthiness: "Super High"} ]
```

Yes, it's that easy.

NOTE: If your share URL has a `/d/e` in it, try refreshing the page to see if it goes away. If it doesn't, [try this](#if-your-publish-to-web-url-doesnt-work).

# Getting Started

### 1) Publishing your Google Sheet

The first step is to get your data out into a form Tabletop can digest

Make a [Google Spreadsheet](http://drive.google.com). Give it some column headers, give it some content.

```
    Name            Category   Healthiness
    Carrot          Vegetable  Adequate
    Pork Shoulder   Meat       Questionable
    Bubblegum       Candy      Super High
```

Now go up to the `File` menu and pick `Publish to the web`. Fiddle with the options, then click `Start publishing`. A URL will appear, something like `https://docs.google.com/spreadsheets/d/e/2PACX-1vQ2qq5UByYNkhsujdrWlDXtpSUhh7ovl0Ak6pyY3sWZqEaWS2lJ0iuqcag8iDLsoTuZ4XTiaEBtbbi0/pubhtml` .

IGNORE THIS URL! You used to be able to use it, you can't anymore (you still need to do this step, though).

Now that you've published your sheet, you now need to share it, too.

1. Click the **Share** link in the upper right-hand corner
2. Click the very pale **Advanced** button
3. **Change...** access to "On - Anyone with a link"
4. Make sure **Access: Anyone** says **Can view**, since you don't want strangers editing your data
5. Click **Save**

Copy the **Link to Share**. Your URL should look something like `https://docs.google.com/spreadsheets/d/1Io6W5XitNvifEXER9ECTsbHhAjXsQLq6VEz7kSPDPiQ/edit?usp=sharing`. It should **not** have a `/d/e` in it.

### 2) Setting up Tabletop

Now you're going to feed your spreadsheet into Tabletop

Include the Tabletop JavaScript file in your HTML, then try the following, substituting your URL for `publicSpreadsheetUrl`

```
    <script
src='https://cdnjs.cloudflare.com/ajax/libs/tabletop.js/1.5.1/tabletop
.min.js'></script>
    <script type='text/javascript'>
      var publicSpreadsheetUrl =
'https://docs.google.com/spreadsheets/d/1sbyMINQHPsJctjAtMW0lCfLrcpMqo
GMOJj6AN-sNQrc/pubhtml';
```

```
      function init() {
        Tabletop.init( { key: publicSpreadsheetUrl,
                         callback: showInfo,
                         simpleSheet: true } )
      }

      function showInfo(data, tabletop) {
        alert('Successfully processed!')
        console.log(data);
      }

      window.addEventListener('DOMContentLoaded', init)
    </script>
```

After Tabletop reads your Sheet, it hops to the `showInfo` function with your data. Open up your console and check out the data it retrieved. All of those rows were turned right into objects! **See how easy that was?**

### 3) Honestly, that's it.

Check out the reference and the examples, but basically you're set. The only thing to think about right _now_ is if you want to deal with multiple sheets you can get rid of `simpleSheet: true` (more on that later).

You might also be interested in the publishing/republishing/publish-as-it-changes aspects of Google Spreadsheets, but you'll need to google that for any specifics.

# Reference

## Tabletop initialization

The simplest Tabletop initialization works like this:

```
    var tabletop = Tabletop.init({
      key: '1sbyMINQHPsJctjAtMW0lCfLrcpMqoGMOJj6AN-sNQrc',
      callback: showInfo
    })
```

You pass in either `key` as the actual spreadsheet key, or just the full published-spreadsheet URL.

`showInfo` is a function elsewhere in your code that gets called with your data.

## Tabletop initialization options

#### key

`key` is the key of the published spreadsheet or the URL of the published spreadsheet.

#### callback

`callback` is the callback for when the data has been successfully pulled. It will be passed an object containing the models found in the spreadsheet (worksheets => models), and the tabletop instance. Each of these models contains the rows on that worksheet (see Tabletop.Model). If simpleSheet is turned on it simply receives an array of rows of the first worksheet.

#### simpleSheet

`simpleSheet` can be true or false (default false). It assumes you have one table and you don't care what it's called, so it sends the callback an array of rows instead of a list of models. Peek at the examples for more info.

#### parseNumbers

`parseNumbers` can be true or false (default false). If true, Tabletop will automatically parse any numbers for you so they don't run around as strings.

#### orderby

`orderby` asks Google to sort the results by a column. You'll need to strip spaces and lowercase your column names, i.e. `{order: 'firstname'}` for a column called **First Name**. You'll want to use this when you only have a single sheet, though, otherwise it will try to sort by the same column on every single sheet.

#### reverse

`reverse` reverses the order if set to true.

#### postProcess

`postProcess` is a function that processes each row after it has been created. Use this to rename columns, compute attributes, etc.

For example:

````javascript
 postProcess: function(element) {
   // Combine first and last name into a new column
````

~~~
   element["full_name"] = element["first_name"] + " " +
element["last_name"];

   // Convert string date into Date date
   element["timestamp"] = Date.parse(element["displaydate"]);
 }
````

#### wanted

`wanted` is an array of sheets you'd like to pull. If you have 20
sheets in a public spreadsheet you might as well only pull what you
need to access. See the example in simple/multiple.html. Defaults to
all.

#### endpoint

`endpoint` is the protocol and domain you'd like to query for your
spreadsheet. Defaults to `https://spreadsheets.google.com`.

#### singleton

`singleton` assigned the instantiated Tabletop object to
Tabletop.singleton, implemented to simplify caching and proxying of
requests. Defaults to `false`.

#### simple_url

`simpleUrl`, if true, changes all requests to `KEY` and `KEY-
SHEET_ID`. Defaults to `false`.

#### proxy

`proxy` allows you to easily use spreadsheets not located on Google
Spreadsheet servers. Setting `proxy: "http://www.proxy.com"` is
equivalent to setting `{ simple_url: true, singleton: true, endpoint:
"http://www.proxy.com" }`.
[Flatware](https://github.com/jsoma/flatware) might provide better
documentation.

#### wait

`wait` prevents tabletop from pulling the Google spreadsheet until
you're ready. Used in the backbone.js example.

#### ~~query~~

~~`query` sends a [structured
query](https://developers.google.com/google-
apps/spreadsheets/#sending_a_structured_query_for_rows) along with the
~~~

spreadsheet request, so you can ask for rows with `age > 55` and the like. Right now it's passed with *every request*, though, so if you're using multiple tables you'll end up in Problem City. It should work great with `simpleSheet` situations, though.~~ Doesn't want to work at the moment.

#### debug

`debug` returns on debug mode, which gives you plenty of messaging about what's going on under the hood.

#### authkey

`authkey` is the authorization key for private sheet support.

#### parameterize

`parameterize` changes the src of all injected scripts. Instead of `src`, `src` is URI encoded and appended to `parameterize`, e.g. set it to `http://example.herokuapp.com/?url=`. Mostly for [gs-proxy](https://github.com/MinnPost/gs-proxy).

#### callbackContext

`callbackContext` sets the `this` for your callback. It's the tabletop object by default.

#### prettyColumnNames

`prettyColumnNames` can be true or false (default to true, unless `proxy` is enabled&dagger;). Since Google doesn't pass us exactly the same column names as in the header ('$ Processed' becomes 'processed'), it takes an extra request to correct them. If you don't want the extra request, you'll want to set it to `false`

> &dagger; prettyColumnNames doesn't work with [Flatware](https://github.com/jsoma/flatware), is why we disable it with a proxy by default

## Tabletop object attributes and methods

Once you're in the callback, you get the data **and** a `tabletop` object. That object is capable of all sorts of fun things.

#### .sheets()

`.sheets()` are the `Tabletop.Model`s that were populated, one per worksheet. You access a sheet by its name.

`.sheets(name)` is how you access a specific sheet. Say I have a worksheet called **Cats I Know**, I'll access it via `tabletop.sheets("Cats I Know")`

#### .modelNames

`.modelNames` are the names of the models [read: sheets] that Tabletop knows about. The sheet names do *not* reflect their ordering in the original spreadsheet.

#### .foundSheetNames

`.foundSheetNames` are the names of the sheets [read: models] that Tabletop knows about. Their order reflects the sheets' order in the original spreadsheet.

#### .data()

`.data()` returns the rows of the first model if you're in simpleSheet mode. It's the same as `.sheets()` otherwise. This is just a little sugar.

#### .fetch()

`.fetch()` manually initializes a data request to the Google Sheet.

#### .addWanted(name)

`.addWanted(name)` adds a sheet to the list that are updated with `.fetch`

## Tabletop.Model attributes and methods

Tabletop refers to sheets as **Models,** which have a few extra abilities compared to the sheets-as-plain-objects.

#### .name

`.name` is the name of the worksheet it came from (the tab at the bottom of the spreadsheet)

#### .columnNames

`.columnNames` gives you the names of the columns in that table

#### .originalColumns

`.originalColumns` gives you the names of the columns that Google sends on the first pass (numbers stripped, lowercase, etc)

#### .prettyColumns

`.prettyColumns` gives you the mapping between the column headers in the spreadsheet and the and the `columnNames`. Disabled by passing `prettyColumnNames: false` when initializing Tabletop.

#### .all()

`.all()` returns an array of the rows of the table, in the style of `[ { name: "Tom", age: 5}, { name: "Liz", age: 12 } ]`

#### .toArray()

`.toArray()` returns the rows of the table with numbered indices instead of named ones [["Tom", 5] , ["Liz", 12]]

## So what the hell do I do with this?

Imagine it's a read-only, JavaScript CMS that you can edit through Google Docs. It's like _Christmas_ up in here.

You can see examples of working with different systems in, yes, `/examples/`.

### Tabletop and any templating system (Handlebars etc)

Super easy. Just feed the models to your template and you're all set.

### Tabletop and Backbone.js

I've put together a `Backbone.tabletopSync` driver for Backbone collections and models. It's read-only, but you can't really complain if you're powering your Backbone app through Google Spreadsheets.

Source is, of course, in `/src`, and you can check it out in action in `examples/backbone/`

### Tabletop and AngularJS

[Ændrew Rininsland (@aendrew)](http://www.github.com/aendrew) at [The Times and Sunday Times](http://www.github.com/times) has created a module that makes using Tabletop with [AngularJS](http://www.angularjs.org) extremely easy. It also includes a loader for [angular-translate](https://angular-translate.github.io) that gives Tabletop the ability to provide i18n translation strings.

Please see [times/angular-tabletop](http://www.github.com/times/angular-tabletop) for more details.

## Caching/Proxying Google Spreadsheets

Yeah, Google Spreadsheets can sometimes be slow or sometimes be overwhelmed or *maybe* one day Google will just up and disappear on us. So Tabletop.js now supports fetching your data from elsewhere, using options like `endpoint` and `proxy`.

`proxy` is the fun one, in that it rewrites your requests to be simpler-looking and plays nicely with the app & example I put together.

### Using Flatware

If you don't mind running around with Heroku and AWS, [Flatware](https://github.com/jsoma/flatware) is an app I built that uploads the spreadsheet JSON response to S3.

### Using table-service

[table-service](https://github.com/martinburch/table-service) hosts it on your own server using a python script, and auto-updates thanks to a tiny script you add to your spreadsheet.

### Using gs-proxy

[gs-proxy](https://github.com/MinnPost/gs-proxy) is another option that also uses Heroku. You'll set `parameterize` to something like `http://example.herokuapp.com/?url=` and off you go!

### Using other caching

You can point `proxy` at anything you'd like as long as it has `KEY` and `KEY-SHEET_ID` files sitting in a directory. Feel free to host it on your own server! You can use `/caching/local.rb` if you want a pretty easy solution for generating the flat files.

# Notes

## Strange behavior

Empty tables are trouble. We can't get column names from them (c'mon, Google!), so don't be too confused when a table with 0 rows is coming back with an empty `.column_names` or your code starts throwing weird errors when processing the results.

Empty rows are trouble. If you have a row that's completely empty, Google doesn't return any rows after the empty row. As a result, you need to make sure every line in your spreadsheet has data in it.

## If you are having trouble

Turn on debugging by passing `debug: true` when you initialize
Tabletop. Check out the console, I try to keep my error messages
chatty and informative. Or just email me at
jonathan.soma@gmail.com, I'm happy
to help!

## Tabletop.js in the wild

The more examples the better, right? Feel free to fork or contact
me if you have a good example of something you've done.

A [contextual video
player](http://www.aljazeera.com/indepth/interactive/2012/04/201241071
56511888.html) with [popcorn.js](http://popcornjs.org) by
[@maboa](https://twitter.com/maboa)

The [WNYC mayoral tracker](http://project.wnyc.org/elections/mayor-
tracker/) uses Tabletop along with
[Backbone.js](http://backbonejs.org)

A [Facebook-esque timeline](http://builtbybalance.com/github-
timeline/) from [Balance Media](http://builtbybalance.com) (with a
[git repo](https://github.com/balancemedia/Timeline))

[Mapsheet](https://github.com/jsoma/mapsheet) creates super easy,
customizable maps.

## Other Options

If you aren't really feeling Tabletop, you should give
[Dataset](http://misoproject.com/dataset/) a shot. It's "a JavaScript
client-side data transformation and management library," which means
it does a hell of a lot more than our dear Tabletop.

## Credits

[Jonathan Soma](http://twitter.com/dangerscarf), who would rather be
cooking than coding. Inspired by the relentless demands of [John
Keefe](https://twitter.com/jkeefe) of WNYC.

Thanks to [Scott Seaward](https://github.com/plainview) for
implementing multi-instance Tabletop.

[Alan Palazzolo](https://github.com/zzolo) hooked the world up with
[gs-proxy](https://github.com/MinnPost/gs-proxy) and added support for
it into Tabletop via `parameterize`

[Max Ogden](https://github.com/maxogden) was kind enough to lend
Tabletop nodejs support as part of an [Open

News](http://www.mozillaopennews.org) [code sprint](http://www.mozillaopennews.org/codesprints.html)