Problem Set 8: CS50 Shuttle

due by noon on Thu 11/10

Per the directions at this document's end, submitting this problem set involves submitting source code via `submit50` as well as filling out a Web-based form, which may take a few minutes, so best not to wait until the very last minute, lest you spend a late day unnecessarily.

Be sure that your code is thoroughly commented
to such an extent that lines' functionality is apparent from comments alone.

Goals.

- Prepare you for final projects.
- Introduce you to JavaScript and third-party APIs.
- Teach you how to teach yourself new languages.

Recommended Reading.

- `http://www.w3schools.com/js/`
- `https://developer.mozilla.org/en/JavaScript/Guide`
- `http://code.google.com/apis/maps/documentation/javascript/tutorial.html`
- `http://code.google.com/apis/earth/documentation/`

NOTICE.

For this problem set, you are welcome and encouraged to consult "outside resources," including books, the Web, strangers, and friends, as you teach yourself more about HTML, CSS, and JavaScript, so long as your work overall is ultimately your own. In other words, there remains a line, even if not precisely defined, between learning from others and presenting the work of others as your own.

You may adopt or adapt snippets of code written by others (whether found in some book, online, or elsewhere), so long as you cite (in the form of CSS, HTML, or JavaScript comments) the origins thereof.

And you may learn from your classmates, so long as moments of counsel do not devolve into "show me your code" or "write this for me." You may not, to be clear, examine the source code of classmates. If in doubt as to the appropriateness of some discussion, contact the staff.

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed in writing by the course's instructor. Collaboration in the completion of problem sets is not permitted unless otherwise stated by some problem set's specification.

Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student or soliciting the work of another individual. Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available solutions to problem sets to individuals who take or may take this course in the future. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the course's instructor.

You may turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on problem sets or your own final project. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is *Admonish*, *Probation*, *Requirement to Withdraw*, or *Recommendation to Dismiss*, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.

Grades.

Your work on this problem set will be evaluated along four axes primarily.

Scope. To what extent does your code implement the features required by our specification?
Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (*i.e.*, clearly, efficiently, elegantly, and/or logically)?
Style. To what extent is your code readable (*i.e.*, commented and indented with variables aptly named)?

All students, whether taking the course Pass/Fail or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a passing grade (*i.e.*, Pass or A to D–) unless granted an exception in writing by the course's instructor.

Getting Started.

☐ OMG, last problem set.

☐ If you haven't already, consider downloading and installing **Firefox** and **Firebug** (in that order) on your own computer (outside of the CS50 Appliance), both of which are available on the course's website under **Software**. Once installed, Firebug will appear as a bug icon in Firefox's top-right corner. Anytime you'd like to print some debugging information to your browser's window, a la `printf` in C, you can include a line like

```
console.log("hello, world");
```

in your JavaScript code. So long as Firebug's window is open, you'll see that text in its console. Be sure to remove any such lines before submitting your work. Alternatively, you can include a line like

```
alert("hello, world");
```

in your JavaScript code instead, but the pop-up that results tends to be more annoying, especially if you accidentally call `alert` in a loop!

☐ For this problem set, your work must ultimately behave the same on the latest version of at least two major browsers:

- ☐ Chrome
- ☐ Firefox
- ☐ Internet Explorer
- ☐ Opera
- ☐ Safari

Be sure, then, to test your work thoroughly with at least two of those browsers. One could be Firefox within the appliance; the other could be some browser on your own computer (besides Firefox). Or you can use two different browsers on your own computer. It's fine if you notice slight aesthetic differences between the two browsers so long as your work functions properly on both. Make sure that your teaching fellow knows which browsers to use whilst evaluating your work.

☐ Launch VirtualBox (as by double-clicking its icon wherever it's installed), and then boot the appliance (as by single-clicking it in VirtualBox's lefthand menu, and then clicking **Start**).

Upon reaching John Harvard's desktop, open a terminal window (remember how?) and type the below, followed by Enter:

```
sudo yum -y update
```

Input **crimson** if prompted for John Harvard's password. For security, you won't see any characters as you type. Realize that updating the appliance in this manner requires Internet

access. If on a slow connection (or computer), it might take a few minutes to update the appliance. Don't worry if the process seems to hang if it decides to update a "package" called `cs50-appliance`; that one can take several minutes.

If you see messages like **Couldn't resolve host** or **Cannot retrieve metalink for repository**, those simply mean that the appliance doesn't currently have Internet access. Sometimes that happens if you've just awakened your computer from sleep or perhaps changed from wireless to wired Internet or vice versa. If your own computer does have Internet access (which you can confirm by trying to visit some website in a browser on your own computer) but the appliance does not (which you can confirm by trying to visit the same with Firefox within the appliance), try restarting the appliance (as by clicking the green icon in its bottom-right corner, then clicking **Restart**).[1] If, upon restart, the appliance still doesn't have Internet access, head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` for help!

Once the appliance has been updated, you should see **Complete!** in your terminal window. If there was nothing to update, you'll see **No packages marked for Update** instead.

☐ Just to be sure that everything worked, go ahead and execute that very same command again in a terminal window (though not while its first invocation is still running):

```
sudo yum -y update
```

Again, input **crimson** if prompted for John Harvard's password. (If only a few minutes have passed since the last update, you might not even be prompted.) You should now see **No packages marked for Update**, which means that your appliance is now up-to-date! If you see some error instead, try once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

☐ Head to the URL below using a browser on your own computer (outside of the CS50 Appliance).

```
http://earth.google.com/plugin/
```

If prompted to download the Google Earth Plugin, do go ahead and proceed to install.[2] Once the plugin's installed, you may need to reload that page or restart your browser, but you should ultimately see a 3D Earth embedded in the page.[3] Close the page once you do.

Because the Google Earth Plugin only supports Mac OS and Windows, you won't be able to install it inside of the appliance.

---

[1] Alternatively, you can try typing:
`sudo service network restart`
But if that doesn't work, best to restart the appliance.

[2] Note that the Google Earth Plugin and, in turn, this problem set require that you meet the system and browser requirements documented at `http://maps.google.com/support/bin/answer.py?hl=en&answer=166094`. If you do not, email `sysadmins@cs50.net`. You may be able to run a supported version of Windows in a "virtual machine" on your computer.

[3] If informed that "you do not have permission to use this service over SSL," odds are you have Force-TLS or HTTPS Everywhere or the like installed; you'll want to disable any such plugin temporarily, at least for `www.google.com`, which is where the Google Earth API lives.

Incidentally, best to minimize the number of programs and windows you have open while working on this problem set; the Google Earth Plugin likes to consume CPU cycles and RAM. In fact, if you ever feel your computer slowing down, quit some programs or even reboot (after saving your work).

☐ Download this problem set's distro into your ~/public_html directory by executing

```
cd ~/public_html
git clone http://cdn.cs50.net/2011/fall/psets/8/pset8.git
```

in a terminal window. You should see **`Cloning into pset8...`** and then your prompt again. If you instead see **`fatal`** followed by **`not found: did you run`**, odds are you made a typo. Best to try again!

Once successful, you should find that you have a brand-new `pset8` directory inside of your home directory's `public_html` directory. You can confirm as much with:

```
cd ~/public_html
ls
```

Next ensure that `~`, `~/public_html`, and `~/public_html/pset8` are "world-executable" by executing

```
chmod a+x ~
chmod a+x ~/public_html
chmod a+x ~/public_html/pset8
```

so that the appliance's web server (and you, from your own computer) will be able to access your work. (Granted, the first two of those directories, `~` and `~/public_html`, should already be world-exectuable per Problem Set 7.)

Now navigate your way to `~/public_html/pset8` by executing the command below.

```
cd ~/public_html/pset8
```

Then run `ls`. You should see the below.

```
buildings.js  index.html  passengers     service.css  shuttle.js
houses.js     math3d.js   passengers.js  service.js
```

All of the work that you do for this problem set will reside in `~/public_html/pset8`. Go ahead and `chmod` the contets of `~/public_html/pset8` as follows. (Remember how?)

- ☐ All CSS (`.css`), JPEG (`.jpg`), HTML (`.html`), and JavaScript (`.js`) files should be readable and writable by you but only readable by everyone else (*i.e.*, 644).
- ☐ All subdirectories (*i.e.*, `~/public_html/pset8/passengers`) should be world-executable (*i.e.*, 711).

Now head to the URL below using a browser on your own computer (outside of the CS50 Appliance).[4]

```
http://192.168.56.50/~jharvard/pset8/
```

You should see University Hall.

Shuttletime.

☐ Your mission for this problem set is to implement your own shuttle service that picks up passengers all over campus and drops them off at their houses. Your shuttle is already equipped with an engine, a steering wheel, and 35 seats. Shall we go for a spin? Here's how to drive with your keyboard:[5]

Move Forward: W
Move Backward: S
Turn Left: left arrow
Turn Right: right arrow
Slide Left: A
Slide Right: D
Look Downward: down arrow
Look Upward: up arrow

Note that your mouse is not used for driving in this 3D world. In fact, don't even click on it, else you'll take away "focus" from our (and, soon, your) JavaScript code, the result of which is that the shuttle will no longer respond to your keystrokes. If that does happen, simply click the 2D map or anything above it (within the same window) to give focus back to the code; the shuttle should then respond to your keystrokes again.

Anyhow, go for a ride! (No bikes in the Yard, but shuttles are okay.) As you approach buildings, you may find that they get prettier and prettier as more satellite imagery is automatically downloaded. See if you can make your way to your own house. (Try not to take any shortcuts through buildings.) Along the way, you'll likely see some familiar faces. Those will soon be your passengers!

In fact, go ahead and click a few times the minus (–) sign in the top-left corner of the application's 2D map. You should see more and more red markers as you zoom out, each of which represents a passenger waiting for pickup. If you hover over each marker with your mouse, you'll see each passenger's name and origin. The blue bus, of course, represents you! You're welcome to click and drag the 2D map to see more of campus; as soon as you start driving again, the map will be re-centered around you.

[4] Note that this URL is equivalent to `http://192.168.56.50/~jharvard/pset8/index.html`.
[5] Indeed, your shuttle can slide left and right, as though all four wheels can turn 90 degrees. As for looking downward and upward, know that you can look straight ahead down to your shuttle's toes, then back up; you can't looker higher than straight ahead. (Sun visor's in the way.)

Above the 2D map is a list of your 35 seats, each of which is currently empty. Above that, in the application's top-right corner, are two buttons: **Pick Up** and **Drop Off**. Neither works yet, but both will before long!

If you happen to get lost, just reload the page, and you'll be returned to University Hall. Your soon-to-be passengers will also be pseudorandomly repositioned throughout campus.

☐ Alright, let's take a stroll through this problem set's distro. Open up `index.html` first and read over the lines of HTML within. Notice first how this file references several others in its head, namely `services.css`, `math3d.js`, `buildings.js`, `passengers.js`, `shuttle.js`, and `service.js`, as well as the URL of Google's JavaScript API. Notice next how the `body` tag has a few event handlers, each of whose implementations lives, as we'll soon see, in `service.js`. And notice how each of the `div` elements is uniquely identified with an `id` attribute so that we can stylize it with CSS or access it via JavaScript.

At the moment, the HTML within two of those `div` elements (`#announcements` and `#seats`) is meant to be temporary. Eventually, there'll be some announcements, and there will be passengers in seats!

Next open up `service.css`, which stylizes that HTML. You don't need to understand all of the CSS within, but do know in general that `div#foo`, refers to the `div` element whose `id` is `foo`.

Now take a peek at `buildings.js`. Declared in that file is `BUILDINGS`, which is an array of objects, each of which represents a building on campus. Associated with each building is a "root" (Harvardspeak for a building's ID), a name and address, and a pair of coordinates that represents an edge of that building. We could have declared this array in `service.js`, but because it's so big, we decided to isolate it in its own file.

Similarly do passengers have their own file. In `passengers.js` is `PASSENGERS`, another array of objects, each of which this time represents a passenger on campus. Associated with each passenger is a username (*i.e.*, a unique identifier), a name, and a house. Incidentally, each of those usernames maps to a similarly named JPEG in `~/public_html/pset8/passengers`. Note, though, that some houses comprise multiple buildings, and so even though "Mather House" appears in both `passengers.js` and `buildings.js`, "Quincy House" appears only in `passengers.js`. In `buildings.js`, meanwhile, are objects for "Quincy House New Residence Hall," "Quincy House Library," and (nice and confusingly) "Mather Hall, Quincy House." And so you will find in `houses.js` a third (and final!) data structure, this one an object whose keys are houses' names and whose values are objects representing houses' coordinates. And so you can access the best house's coordinates with something like:

```
var lat = HOUSES["Mather House"].lat;
var lng = HOUSES["Mather House"].lng;
```

You may assume that the coordinates in `houses.js` are the official coordinates for passengers' destinations. In fact, we've already planted yellow markers at those very coordinates on the 2D map to help out the driver. We've not bothered planting placemarks at those coordinates on the 3D Earth, since they're not as easy to spot.

Incidentally, if, during the course of this problem set, you'd like to look up where some building is on campus, feel free to search for it via CS50's own app:

`http://maps.cs50.net/`

Now take a peek at `shuttle.js`. This file's a bit fancy, inasmuch as it effectively implements a data structure called `Shuttle`.[6] You don't need to understand this file's entirety, but know that inside of a `Shuttle` are two properties: `position`, which encapsulates a shuttle's position (including its longitude and latitude), and `states`, which encapsulates a shuttle's various states.

Okay, now turn your attention to `service.js`, where you'll do most of your work, though you're welcome to modify any of this problem set's files, except for `math3d.js` (which is just a library), `buildings.js`, `houses.js`, and `passengers.js`. Atop `service.js` are a bunch of constants and globals. Just below those lines are two calls to `google.load`, which loads the two APIs on which this application relies:

Google Maps JavaScript API V3
`http://code.google.com/apis/maps/documentation/javascript/tutorial.html`

Google Earth API
`http://code.google.com/apis/earth/documentation/`

You won't need to read through the entirety of that documentation; odds are you'll explore it as needed. But do read the first page or so of each to get a sense of what each API does.

Implemented in `service.js` are all of those event handlers you first saw in `index.html`. Scroll down to the implementation of `load` first. It's this function that embeds the 2D map and 3D Earth map in your browser. Next scroll back up to the implementation of `initCB`; this function gets called as soon as the Google Earth Plugin has loaded, and so it's in this function that we finish initializing the app. (If something goes wrong, it's `failureCB` that's instead called.) Notice, in particular, that `initCB` "instantiates" an object, `shuttle`, of type `Shuttle`.

Next take a look at the function called `keystroke`. It's this function that handles your keystrokes. Odds are this function will bring back memories of `sudoku`, though this time we used `if` and `else` instead of a `switch`. In response to your keystrokes, this function changes the state of the shuttle simply by updating the appropriate property in `shuttle.states` with a value of `true` or `false`.

Now take a peek at the function called `populate`. It's this function that plants friendly faces all over campus. Each face is implemented as a "placemark" on the 3D earth and as a "marker" on the 2D map.
Finally, take a look at the function called `chart`. That function renders a seating chart (in the `div` whose `id` is `chart`) as an ordered HTML list (0-indexed for your debugging convenience) for the

[6] Although Shuttle behaves like a "class" (if familiar), JavaScript is actually a "prototype-based" language:
`http://en.wikipedia.org/wiki/Prototype-based`

shuttle's 35 seats. Notice how it iterates over `shuttle.seats`, an array that's initialized to be of size `SEATS` (*i.e.*, 35) in `shuttle.js`. It looks like `null` represents an empty seat!

☐ By the way, no need for any PHP or SQL for this problem set, just CSS, HTML, and JavaScript. Be sure, then, to reload your browser (or clear your cache) often so that you always have the latest versions of your code.[7]

☐ Alright, it's time start picking passengers up. Recall from `index.html` that, when the button labeled **Pick Up** is clicked, the function called `pickup` in `service.js` is called. Implement pick-ups as follows.

If the button is clicked when the shuttle is within 15.0 meters of a passenger and at least one seat is empty, that passenger should be given a seat on the shuttle and removed from both the 2D map and 3D earth. (If multiple passengers are within 15.0 meters, you should pick up as many of them as there are empty seats.) If the shuttle is not within 15.0 meters of any passenger, make an announcement to that effect. If the shuttle is within range of some passenger but no seats are free, make an announcement to that effect (and don't pick up the passenger). Any such announcements should be cleared (or replaced with some default text) as soon as the shuttle starts moving.

Not sure how to proceed? That's part of the challenge! Odds are you'll encounter similar uncertainty when you dive into your final project. Whenever in doubt, peruse the APIs' documentation, and turn to Google (the search engine) for tips. And you are encouraged to turn to `help.cs50.net` for guidance from classmates and staff. But we'll give you some hints to get you started on pick-ups.

To calculate the shuttle's distance, `d`, from some point (`lat`, `lng`), you'll probably want something like:

```
var d = shuttle.distance(lat, lng);
```

To remove a placemark, `p`, from the 3D Earth, you'll probably want something like:

```
var features = earth.getFeatures();
features.removeChild(p);
```

To remove a marker, `m`, from the 2D map, you'll probably want something like:

```
m.setMap(null);
```

Unfortunately, `populate`, at the moment, doesn't remember the placemarks or markers that it plants, so you may need to tweak `populate` as well, per its `TODO`. Perhaps you could store those placemarks and markers in some global array(s) and/or object(s) so that you can remove them from the 3D Earth and 2D map, respectively, later?

[7] In particular, you may want to force-reload your browser often, as by holding Shift when you reload.

And how to give a picked-up passenger a seat? Odds are you'll want to update `shuttle.seats`, but be sure not to add extra seats. And odds are you'll want to update `div#seats` anytime that array is updated by calling `chart`. But, at the moment, `chart` only knows how to display empty seats; be sure to replace its `TODO` with some code that displays picked-up passengers names and houses, so that you know who and where to drop off.

As for making announcements, recall that code like the below will get the job done:

```
document.getElementById("announcements").innerHTML = "hello, world";
```

☐ Alright, it's now time to implement drop-offs! Recall from `buildings.js` that each passenger lives in a house. And recall from `index.html` that, when the button labeled **Drop Off** is clicked, the function called `dropoff` in `service.js` is called. Implement drop-offs as follows.

If the button is clicked when the shuttle is within 30.0 meters of an on-board passenger's house, that passenger should be dropped of by emptying their seat. (If there are multiple passengers on board that live in that house, all should be dropped off in this manner.) If the shuttle is not within 30.0 meters of any passenger's house, make an announcement to that effect. Any such announcement should be cleared (or replaced with some default text) as soon as the shuttle starts moving.

No need to re-plant a placemark or marker for dropped-off passengers; assume they're going to head straight inside. As for passengers' who're still on board after a drop-off, be sure not to make them change seats just because some other seat is now empty.

☐ Nice work so far! Suffice it to say this shuttle service is starting to feel like a game. It's time to allow you some creativity. (Think back to your Scratch days!) Implement any one (1) of the features below.

- ☐ Implement points, whereby the driver earns one point for each passenger dropped off. Be sure to announce the driver's score anytime it changes. And be sure to announce when every single passenger has been picked up and dropped off.
- ☐ Implement a timer, whereby the driver only has some number of minutes or seconds in order to pick up and drop off some number of passengers. Odds are you'll find `window.setInterval` and/or `window.setTimeout` of interest.[8,9]
- ☐ Rather than just list seated passengers' names and houses, group them somehow by house so that it's obvious how many of your 35 or fewer seated passengers are destined for a particular house.
- ☐ Implement the Konami Code in such a way that, if inputted, the shuttle acquires the ability to "fly."[10] But it can only pick up and drop off passengers with its wheels on the ground. We leave it to you to decide which keystrokes to use for flight.

[8] `https://developer.mozilla.org/en/DOM/window.setInterval`
[9] `https://developer.mozilla.org/en/DOM/window.setTimeout`
[10] `http://en.wikipedia.org/wiki/Konami_Code`

- ☐ Replace the blue bus on the 2D map with some kind of arrow that points in the direction that the shuttle is actually facing (in the 3D world). Odds are you'll want to rotate a `canvas` element or use as many as 360 different icons (one for each degree). If you take the latter approach, odds are you can get away with fewer icons, one every few degrees.
- ☐ Implement the ability to teleport the shuttle instantly to any building on campus, as by selecting that building's name from a menu.
- ☐ Implement the ability to increase or decrease the shuttle's velocity.
- ☐ Ensure that passengers do not get pseudorandomly positioned by `populate` at their own houses, lest the driver get annoyed that they only want to travel a few feet.
- ☐ Implement the ability to take joy rides across the campuses of friends' universities and your hometown (then invite your friends and family to get behind your wheel!).
- ☐ Implement the ability to inform passengers, as via an announcement, of the shuttle's current location. Odds are you'll find `http://code.google.com/apis/maps/documentation/geocoding/#ReverseGeocoding` of interest.
- ☐ Implement auto-pilot whereby, when some button or link is clicked, the shuttle drives (without teleporting) itself to a passenger or house. It's fine if auto-pilot likes to drive through buildings.
- ☐ Implement fuel, whereby the shuttle only starts with a finite amount and can only travel some number of meters before it runs out. Build one or more gas stations on campus at which the shuttle can refuel (as by clicking a button or link when nearby or driving through the station).
- ☐ Make-your-own feature. If you would like to implement a feature not listed here but that involves similar effort, you may do so long as your teaching fellow approves in advance.

Although you are only required to implement one of the features above, you are welcome to impress us (and your friends) with more just for fun! And you are welcome to alter your user interface's aesthetics, including your 2D map's icons.[11]

[11] See `http://mapicons.nicolasmollet.com/` for some fun icons.

Sanity Checks.

Before you consider this problem set done, best to ask yourself these questions and then go back and improve your code as needed! Do not consider the below an exhaustive list of expectations, though, just some helpful reminders. The checkboxes that have come before these represent the exhaustive list! To be clear, consider the questions below rhetorical. No need to answer them in writing for us, since all of your answers should be "yes!"

- ☐ Do you only pick up passengers if they're within 15.0 meters of the shuttle?
- ☐ Do you only pick up passengers if there are seats empty?
- ☐ Does your seating chart display passengers' names and houses for non-empty seats?
- ☐ Do you only drop off passengers if they're within 30.0 meters of their house?
- ☐ If multiple passengers are within a prescribed radius, do you handle them properly?
- ☐ Did you implement at least one additional feature?

As always, if you can't answer "yes" to one or more of the above because you're having some trouble, do turn to `help.cs50.net`!

CS50 in the Cloud.

- ☐ Once you think you're all done, it's time to invite one or more friends to try out your site. Of course, your website isn't technically on the Web just yet, since it lives on your own computer within the appliance. But recall that you have, thanks to Problem Set 1, a CS50 Cloud account. Not only does that account allow you to log into the course's website and run `submit50`, it also provides you with access to `cloud.cs50.net` on which you can host your implementation of CS50 Shuttle!

  If you would like to upload your work to `cloud.cs50.net` so that friends and family can try it out via their own computers, head to

  `https://manual.cs50.net/Cloud`

  for instructions. If you'd rather not, that's okay too, but at least invite someone over to try out your site on the appliance itself.

  Just realize that, upon migrating your implementation of CS50 Shuttle to the cloud, you'll need to sign up for your own Google Maps API key (for the Google Earth API's sake) at

  `http://code.google.com/apis/maps/signup.html`

  and then update a `script` tag in `index.html` accordingly. The key that comes with this problem set's distro is tied to `http://192.168.56.50`.

How to Submit.

In order to submit this problem set, you must first execute a command in the appliance and then submit a (brief) form online.

☐ Recall that you obtained a CS50 Cloud account (*i.e.*, username and password) for Problem Set 1. If you don't remember your username and/or password, head to `https://cloud.cs50.net/` look up the former and/or change the latter. You'll be prompted to log in with your HUID (or XID) and PIN.

☐ Just in case we updated `submit50` since you started this problem set, open a terminal window and execute the below, inputting **crimson** if prompted for John Harvard's password.

```
sudo yum -y update
```

If there was something to update, you should see **Complete!** after a few seconds or minutes. If there was nothing to update, you should instead see **No packages marked for Update**. If you see any errors, try the command once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

☐ To actually submit, first open a terminal window and execute:

```
cd ~/public_html
```

Then execute:

```
ls
```

At a minimum, you should see `pset8` (if not `index.html` and `pset7` as well). If not, odds are you skipped some more steps earlier! If everything is as it should be, you are ready to submit your source code and database to us. Execute:

```
submit50 ~/public_html/pset8
```

When prompted for **Course**, input **cs50**; when prompted for **Repository**, input **pset8**. When prompted for a username and password, input your CS50 Cloud username and password. For security, you won't see your password as you type it. That command will essentially upload your entire `~/public_html` directory to CS50's repository, where your TF will be able to access it. The command will inform you whether your submission was successful or not. If provided with the URL of a PDF of your code (which further confirms its submission), right-click (or ctrl-click) the link, then choose **Open Link** from the menu that appears to open the PDF in Document Viewer.

You may re-submit as many times as you'd like; we'll grade your most recent submission. But take care not to submit after the problem set's deadline, lest you spend a late day unnecessarily or risk rejection entirely.

If you run into any trouble at all, let us know via `help.cs50.net` and we'll try to assist! Just take care to seek help well before the problem set's deadline, as we can't always reply within minutes!

If you decided to upload your work to `cloud.cs50.net` but ended up making some changes there to your code, be sure that you made those changes on your appliance as well so that you ultimately submit the right version of your work.

☐ Anytime after lecture on Mon 11/7 but before this problem set's deadline, head to the URL below where a short form awaits:

`https://www.cs50.net/psets/8/`

Once you have submitted that form (as well as your source code and database), you are done!

This was Problem Set 8. Your last!