Preamble

This document describes my solution to a design problem given for a job interview. With the exception of this paragraph, this document in its entirety is what I sent them. The design specifications and a architectural diagram can be found inside the same directory as this document.

While I was given several business requirements (21 to be exact) I was not given any parameters on the design, e.g. are we using an IaaS or a PaaS structure? So I have made the following assumptions:

- the term “app” refers to a software application that runs anywhere and not just on a smartphone
- the system will start out with a handful of users
- the system will eventually be handling Google-scale messages on a daily basis, i.e. billions of message, so it needs to *really* scale
- I am not going to discuss networking issues, i.e. where to put load balancers, etc.
- I am not going to discuss implementation specific items, i.e. whether to use a distributed database like MongoDB or to shard across MySQL instances
- That said, I will be referring to specific items when I feel it helps cement the idea in the reader’s brain. An example of that would be using “Google Calendar” instead of “calendaring app”.

With that said, let us begin

Overview

At the 30,000 foot level, there are three different information flows involved:

1. Information flowing between the user and Google Calendar
2. Information flowing from the user to the HeadsUp system.
3. Information going from the HeadsUp system to the user

I’ll ignore the first flow since we don’t care how the user gets information into Google Calendar, just that she does.

The second flow is just a bog-standard responsive web app. From a design perspective there’s nothing interesting or exciting there. I will, however, describe a design for it so you know that I know what a bog-standard responsive web app is.

The third flow, well, that’s where the fun is. My goal with this is to get something up and running as fast as possible that will scale. For the handful of initial users, my design is going to look ridiculously over-engineered. And it is! But if you have the hardware to throw at it, I believe it will scale nicely.

The Front End

The front end of the system is accessible from any platform: desktop, smartphone, etc., therefore a basic responsive website is preferred.

When the user arrives at the site, they are *not* prompted to immediately register. The initial page explains what the HeadsUp service does and how they can benefit from it. One of the options on the page is to register. If the user decides to register they are taken to a screen which, on the desktop would have tabs and on a smartphone would consists of different screens.

To start the registration process, the user enters a preferred username and password. If the username is not taken and the password is strong enough, they are stored in the backend database and the other fields are enabled. When the user fills in a an email, SMS, or calendar app field, a button will be enabled to let the user check the connection, i.e. if the user fills in the email field and clicks “Check email connection” the system will attempt to send an email via HeadsUp’s email servers to the address the user typed in.

To complete registration, the user will have to fill in the calendar app field and at least the email or SMS field (and have tested them at least once). A field in the database will mark that the user has finished registration which allows the backend system to start using her information.

Front End Build Process

As mentioned earlier, this is just a basic web app. I like to start with the UI, filling in the various aspects when I finish that piece of the UI, i.e. once I finish the email fields I write the code to send email.

While the backend architecture relies heavily on message queues (see next section), the web app would hit a backend program that would redirect the message to the appropriate server directly, the assumption being that there won’t be that much registration traffic to worry about scaling. The app could hit the servers directly, but going through a program on the server side allows for better traffic control and tracking. See accompanying diagram.

The Back End

I designed the backend system primarily to scale. Like all good things, it's decentralized and, I think, fairly robust. Of course, that last claim implies there are monitoring, self-repairing, and autoscaling procedures in place, none of which I discuss in this design document.

One of the design requests was to include some sample API calls. I'm not going to do that since A) the API Call depends on what technology you're using to communicate and B) my design relies heavily on message passing which, to my mind says "I don't care how you get the message there, just get it there. With that in mind, I will be showing what the message will look like at various points.

My design takes the various business requirements ("gets calendar information", "emails the user") and separates them out into individual software agents that communicate with one another via message queues. In the following, I will designate the software agents as "SA" and queues as, well, "Q".

The following SAs are used in the design:

- Fetcher - fetches data from the database marking them as "in process"
- Partitioner - partitions the data into individual users
- CTA - Calendar Transfer Agent communicates with the calendaring app
- Storage SA - puts messages into a datastore
- TextProcessor - processes text
- Email Processor - formats and sends emails
- SMS Processor - formats and sends SMSes (texts)
- UnFetcher - undoes the "in process" marking done by the Fetcher

Each SA, except for the Fetcher, listens to an event queue that the previous SA places messages on, e.g. one or more CTA SAs are listening to an event queue (called the CTA queue) that the Partitioner puts messages on. All of the information an SA needs to do its job is contained in the message it reads off the Q.

Information Flow

This section explains how the backend process works:

The Fetcher SA retrieves a (small-ish) set of records from the datastore based on which records have bedtimes in the next 15 minutes and puts the set onto the Partitioner Q. At this point, the data might look like this:

```
[ { uid: fred,  calendar:google, username: fredtheman, password: foo, bedtime:2700UTC,
    sendemail: yes, sendsms: yes },
  { uid: alice, calendar:google, username: alice2112,  password: bar, bedtime:0700UTC,
    sendemail: no,  sendsms: yes },
 …
]
```

The Partitioner takes the set of records and partitions them into individual records based on UID and places the individual records on the CTA Q.

The CTAs read the individual messages, connects to the appropriate calendaring app, and retrieves the messages of the next day. Depending on the number of calendaring apps supported, CTAs may be able to login to all calendaring apps or just specific ones. If the latter, they would listen only for messages for their specific app.

The CTAs format the data into a simple textual list, maybe something like this:

```
{ uid: fred, email: fred@google.com,
  events: [ 0800 => "Breakfast at Tiffany's",
            0900 => "Meeting with Bob",
            …]
}
```

At this point, you're probably thinking "the CTAs put the message on the Q and the TextProcessor reads them". Eventually, yes, but at this point, I have the option of sending them to the Email/SMS processors. Why? So I can get my product into the hands of the end user more quickly. The TextProcessor (according to the requirements, simply makes the text to be sent prettier; that can wait until after launch if necessary.

So the CTAs put their messages on the TextProcessor Q or the Email/SMS Processor Q depending on where we are in the build process. They also put a message on the Q for the Storage SA to store the data.

As you might imagine, the Email (or SMS) processor takes the text and wraps a pre-formatted email message around the text and emails it.

Finally, they place a message on the UnFetcher's Q which, when read, unmarks the user's record as "in process".

The Build Process

My philosophy of building systems is to get something up as fast as possible even (especially!) without bells and whistles. That will, hopefully, explain some of my choices here.

In order to build this system, the approach/order I would take is as follows:

1. Build a dev, staging, and production environments with the necessary tools.
2. Design and build a RESTful interface to *a* datastore, not necessarily the final one. The first one should probably be a dummy interface with canned responses. At the same time, set up the datastore and get the two working together
3. Design and build a Software Agent (SA) prototype that can be quickly spun up to create new types of SA as needed. The minimum functionality is to read/write from two different queues, a passthrough() function and a process() stub.
4. Do whatever design work you need to easily create a new type of queue.
5. Setup SA 1, SA 2 and a Q and get SA2 reading messages from SA 1 vi the Q
6. Have SA 1 talk to the server in step 2
7. Have SA 2 send a message to the server based on the message SA 1 sent
8. At this point, someone should start thinking about infrastructure to spin things up/down quickly.
9. The priorities from here are to work from the outside in (Partition Server, then the Email Processor,etc.), keeping in mind that at any point you can send a message from the server in step 2 through the system back to the server.
10. The last SA to be written is the TextProcessor since its primary function is to prettify the outgoing message and is not integral to the functionality of the system.

Conclusion

Thanks for reading this far. I hope this document gives you an idea how I think about the development process. I take a similar approach to operations and administration. I know I tend to be a bit verbose but it does help in my thinking process.