Full stack development

Master of science internship

Max Halford

Supervisor: Raphaël Kolm
Referee: Thomas Madaule
April 11^{th} - September 9^{th}

Contents

Figures

Code

Acknowledgements

I would sincerely like to thank the whole staff of Privateaser. It's a small team and at times it really felt like a family. The atmosphere was extremely good and I got the chance to make the most of the nightlife in Paris.

I am grateful to the development team and my CTO for being patient with me and for having taught me so much. I am happy to have been considered somewhat of an equal to them and to not have been put in a box labeled "Intern". I hope I have been worthy and that the things I have worked on will not break!

Chapter 1

Introduction

1.1 The company

Privateaser is a start-up that aims at privatising bars, restaurants and clubs for medium to large groups of people. It targets both professionals (B2B) and private individuals (B2C). It aims to become a marketplace between event organizers and establishments. The revenue model is simple, the user does not have to pay whilst the establishment pays a prearranged fee. For the while Privateaser is only available in Paris, however covering other cities is part of the vision of the company.

As of early 2016, Privateaser employs around 15 employees. Being a start-up, it's usual for this number to fluctuate. At the beginning of my internship we were 20 and a few people joined the adventure as my internship went on.

The development team is composed of 5 people. It's a tight knit group and every member of the team has to be able to multitask.

1.2 The organization

Privateaser, from a developer perspective, is split into three entities:

- The public website, where users can find and then book places.
- The back-office, where the operational team manages people's bookings.
- The manager application, where establishments can view and give feedback on upcoming bookings.

Privateaser embraces AGILE concepts. When I arrived feature development was done during 2 to 3 weeks sprints. Meetings were planned during the sprints and an overview was performed at the end of a sprint. Stand-up meetings were done every morning. Installation and integration are done in a continuous manner. During my internship we experimented with variations of this so as not feel constrained by concepts such as sprints and deadlines.

Code releases are frequent, the idea is to ship often and to maintain a balance between intuition and A/B testing. Each release is versioned with a sequence following the $a.b.c$ format where

- a indicates the major release.
- b indicates a minor release, usually the output of a feature in a sprint (usually daily).
- c indicates a patch/hotfix that can be put in place at any time.

The usual scenario for implementing a feature is

1. brainstorming during a workshop,
2. establishing a specification document,
3. building a model,
4. developing the feature,
5. iterating before release,
6. reviewing the code,
7. merging the code into production.

As previously mentioned, developers have to be very aware of the development progress. As such, each developer is responsible for his features. Building with others is encouraged whilst defending a personal opinion is not. Peer programming is encouraged, as well as continuous refactoring, strong coding conventions and healthy naming. However, it should never be forgotten that in the end all that matters is *the user*.

1.3 The objective

During my internship there wasn't a subject *per say*. The fact that we worked in sprints and split each feature into smaller (and sometimes independent) tasks meant that the kind we performed changed on a daily basis. On a given day I could working on the public search page and on the next I could be checking log files for tracing bugs. In this sense the title of *full stack development* seems appropriate.

During the internship I got to work on many concepts and technologies mostly related to web development. This report is split into independent chapters and can be read as such.

1.4 The workflow

Although the exact process for developing a feature fluctuated as time went on, the gist of our workflow remained the same. Indeed, although the development grew quickly and we had to redefine some of the details of our process, we established a sturdy process for integrating a new feature into production.

As in any tech-driven company, the goal is to identify a business need and to translate it into a *feature* that will be coded and integrated into the current state of the application (called *production*). The business needs result from a *roadmap* that defines the medium to long term goals and a *backlog* that encompasses general ideas/fixes that can be brought to the application.

Features are regrouped into *sprints* that usually last two weeks. The goal during a sprint is simply to integrate each feature. Integrating a feature means

- making sure the business needs are understood,
- implementing the feature with code,
- testing the feature and writing unit tests if time allows,
- getting the code reviewed by other (usually two) developers,
- getting feedback from other people than the developers.

Features can take hours as well as they can take weeks, however most of the features are quite small and it's possible for many features getting integrated into production code every day.

The goal of a code review is to make sure the code in functional but also that it respects the established code style conventions. This isn't someone with a data science background is accustomed to. However for developers this is extremely important as it increases code maintainability. As a general rule, we strived for making sure the code was written as if a single developer had written it. Sometimes this took even more time than the actual implementation!

The process itself was quite classic, if a developer found a bug or a type in a feature he was reviewing he would *decline* it and the author of the feature would fix it and ask the reviewers to have another look until no issues remained.

Chapter 2

Application architecture

2.1 Goals

As of the period of my internship, Privateaser is growing quickly. The code base is increasing in size in consequence. As well as developing new features to expand the product, it is important for any developer to make early decisions so as to ensure a good architecture. Even more so for Privateaser, where even at the business level the application is split into three components, as explained in the introduction of this report.

We did some brainstorming and had a meeting so as to decide of the architecture at a “code” level. The objective is to use a battle tested and accepted standard so as to bring coherence into the application in order to scale.

My preference is Uncle Bob’s “clean architecture”. Although there as many architectures as there are applications, I believe this one encompasses many good practices that are the foundations of a scalable application. It’s pros outweigh it’s cons.

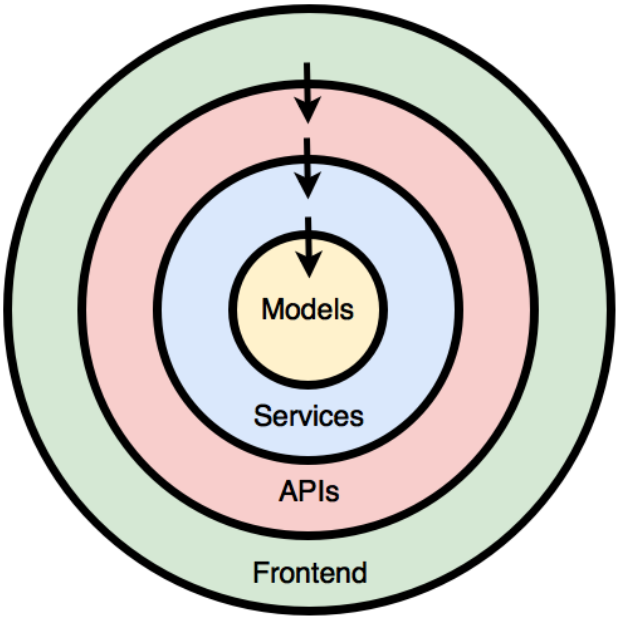


Figure 2.1: Architecture philosophy

The application is split into layers that communicate in a unilateral way (here from top to bottom). Each layer only "knows" the layer right below it. Layers shouldn't call distant layers, they should use intermediary layers. In theory there can be as many layers as needed. For example one may want to add a CDN or a messaging queue to handle the API requests to the service layer. However, a "simple" application like Privateaser's will only require the layers part of the previous figure, they constitute a vital minimum for any healthy application:

Amongst other things, using this architecture renders the application

- **independent of the user interface**, it can be switched out without having to refactor the backend.
- **independent of the database** because the rules are not bound to the database.
- **independent of any framework** because they are wrapped in functions that are considered as tools.
- **testable** because the business rules can be run without the existence of a UI or a database.

It's difficult at first to realize how much good this brings to an application. This is especially true for a small application where an architecture of this kind may seem too much. However, many case examples can be made in favor of taking the time to do this. For example if ever the storage backend changes because MySQL can't handle the load (unlikely) then only the storage service has to be changed. What's more this encourages building towards a service oriented architecture (SOA) where the components/services don't have any knowledge between each other. Everything going

in and out passes by an API, assuring separation of concerns and highly maintainable code.

This “clean” architecture is more of a general advice as to how to structure an application, not a specific blueprint for Privateaser’s needs. The next step is to descend to a lower level and detail how this is architecture translates to our specific application. In a nutshell our goal is to reach the target architecture in two phases.

2.2 Phase 1

First of all, we want to keep the fact of having a single application. However, we want to split the dependencies and isolate the logic between the 3 applications. Inside each application we want to make sure that the frontend is calling the backend through an API which calls a service layer (just like the first diagram of this chapter).

It’s important and necessary to take this first step because there are many common dependencies between each application. For example a lot of them share database models or reference data necessary for generating HTML templates. In a sense this step will still contain code duplication. However, clear factorizations will appear and a unified API layer will be longing to be put in place.

Many steps can be taken so as to split the dependencies between each application. A good example is to have separate Webpack bundles. Being large compared to the others, the public application compiles JavaScript and CSS libraries that are useless for the others. Thus compiling a separate bundle for each application seems like a good idea. Of course if each application uses the same the same library then it will be compiled and included separately in each bundle.

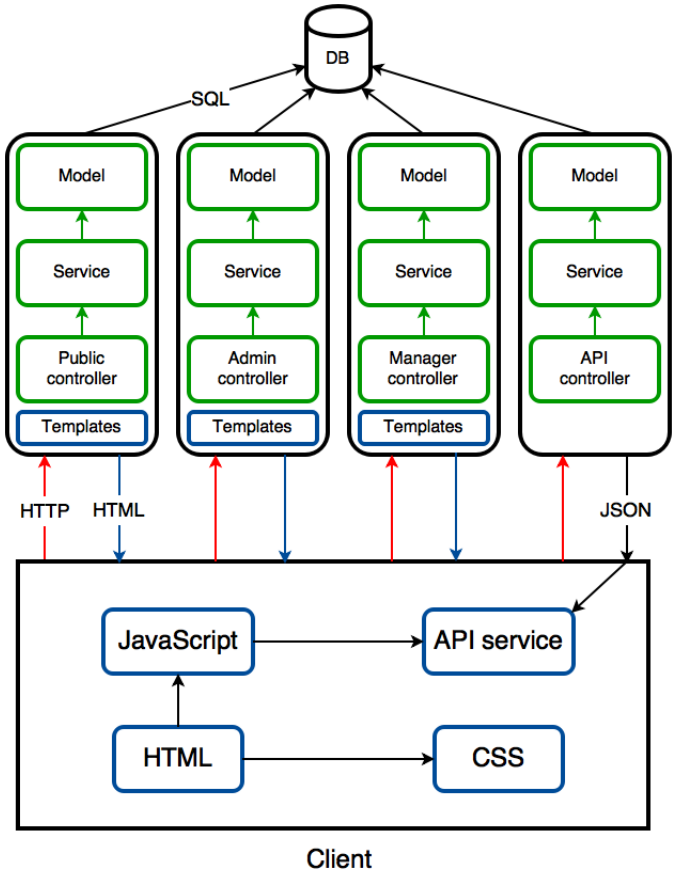


Figure 2.2: Architecture phase 1

2.3 Phase 2

The next (and final) step is to split the application into separate codebases. This somewhat drastic step will enforce us to split the dependencies between the applications because they won't even have access to each other. One of the issues that arises when applying Phase 1 is that one realizes that there is a lot of code duplication occurring. The solution is creation a separate code base that contains absolutely all the services and the layers underneath it, similarly following diagram.

This way each application (reduced to it's controllers and templates) doesn't know how the service layer functions. Interestingly, it shouldn't even know it's coded in Python. This is because each application's frontend should communicate with the service layer through an API with the HTTP protocol.

Another benefit of splitting the application this way is that the code bases become smaller, or at least their scope is greatly reduced. For example it would be possible for a newly hired frontend developer to work on the public application's templates and controllers without having any knowledge or interaction with the backend.

Of course this description is a fairytale and the fact that we are not many program-

mers who each have a hefty workload regarding the product doesn't make it each to put this dream architecture in place. A web application is a bit like a house in the sense that there is always something to do to make it "better".

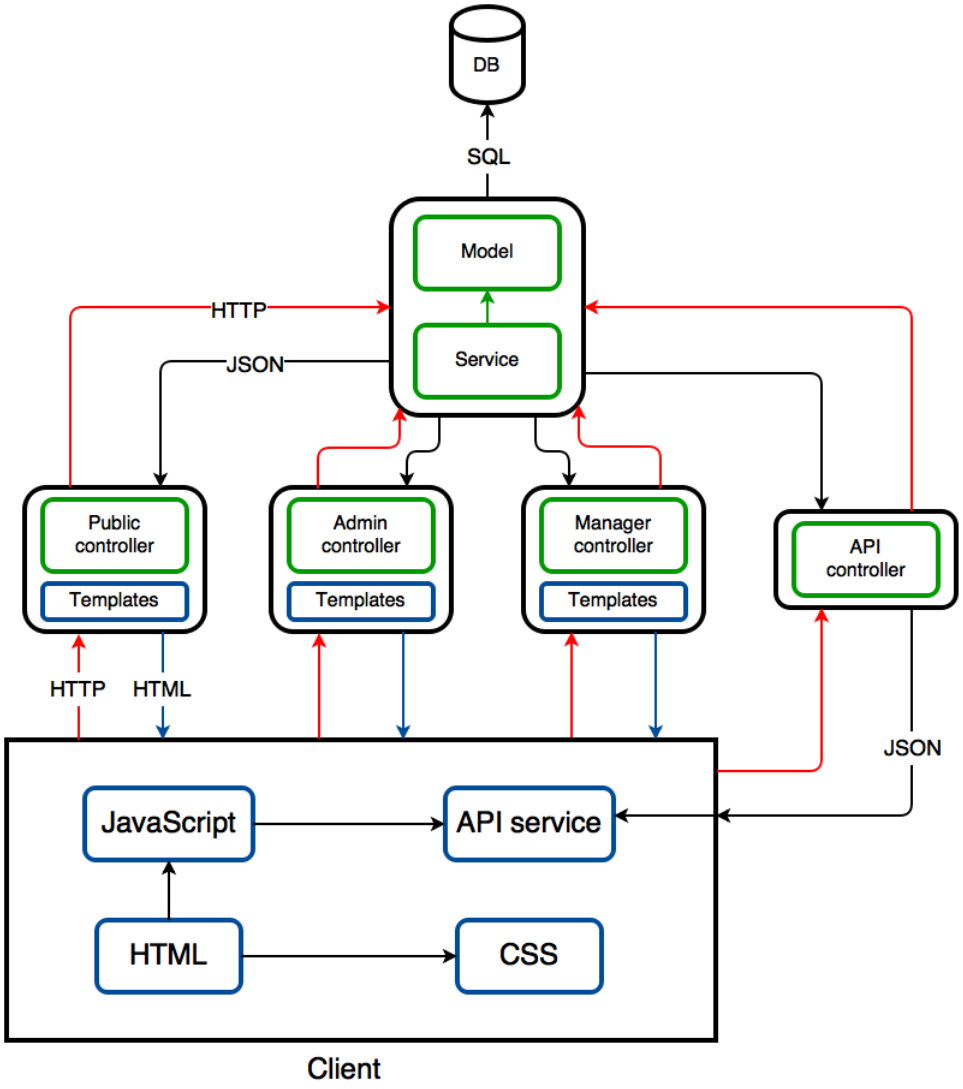


Figure 2.3: Architecture phase 2

2.4 Detailed view

The previous discussion may seem a bit esoteric. It didn't mention how all our tooling is used to manage the codebase. The following diagram tries to fix this by showing where tools like Grunt and Alembic (described in appendix A). Grunt takes care of generating the CSS files from the LESS files. Webpack generates the JS files with the ES6 files. Alembic takes care of the database migrations in the service layer code base.

For the while all the backend code is written in Python, which allows a few dis-

crepancies regarding the permitted use of the models by the controllers. Splitting a codebase into this kind of scaffolding is far from being trivial, hence some leniency is necessary to move forward. Our plan is to do the most possible during offsprints and when there isn't some urgent business matter to attend to (which doesn't occur often). However, it's nice to have these kind of diagrams in everyone's mind. At least the new features are being implemented with this architecture in mind.

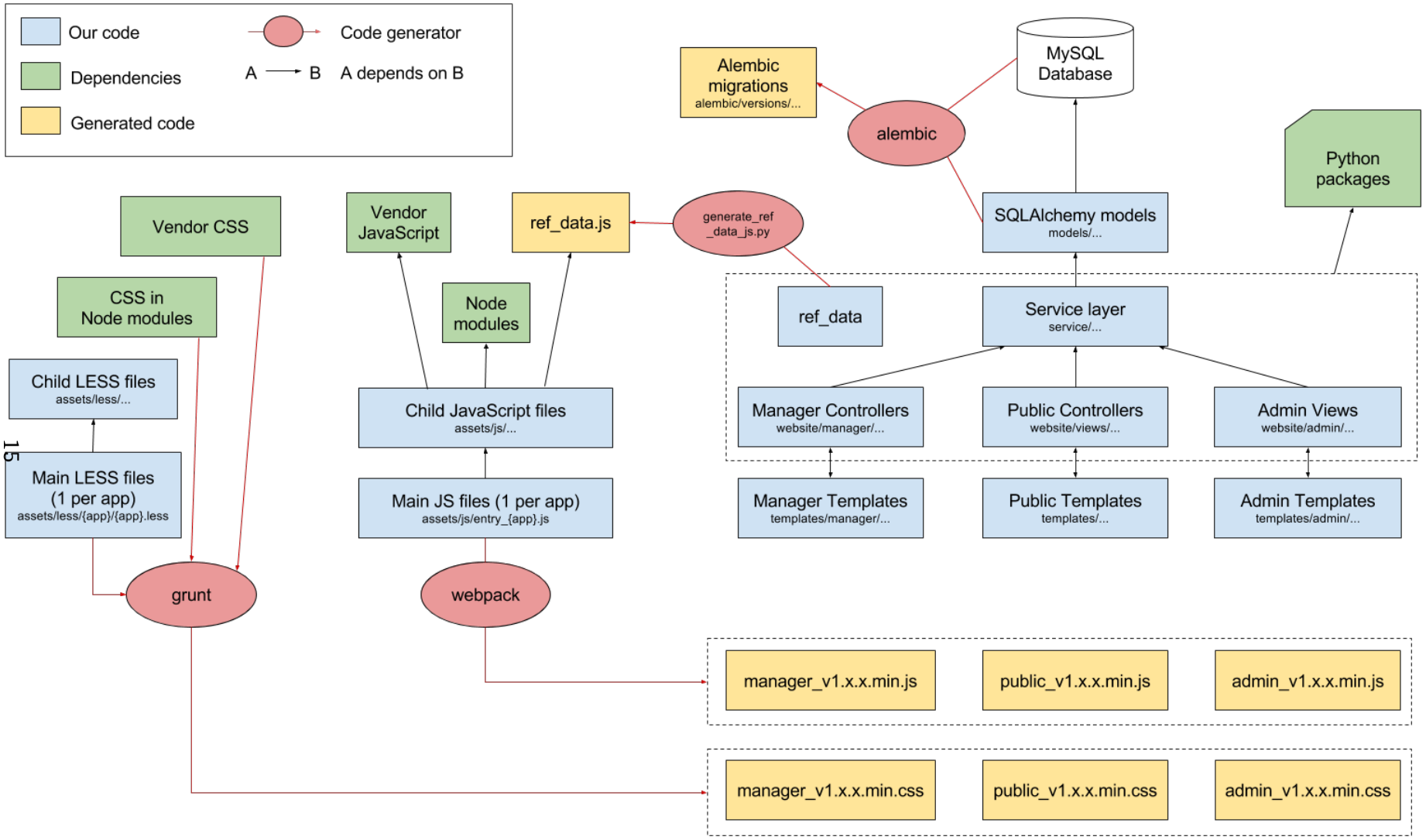


Figure 2.4: Architecture detailed view

Chapter 3

Features

A feature is an addition to an application. Each feature is the result of a thought process initiated by the thinking heads of the company. The features are organized in a roadmap (which is nothing more than an Excel file) according to the time at which they have/should be implemented.

Each feature is then split into smaller subtasks which can (and should) be dealt with in a matter of hours/days. This progressive approach means that pull requests are extremely frequent and code reviews are crucial to make sure buggy code doesn't get merged into production. Of course this isn't an exact science and mistakes happen.

It's important to emphasize the fact that the lifetime of a feature doesn't stop when it's implemented. It's got to be tested, code reviewed, product reviewed and finally tracked for bugs after being put into production.

3.1 Daily routine

As touched upon during the introduction, my internship's goals were not set in stone. In essence my duties reflected the dynamic nature of a startup's development team. First of all our relatively small size meant that we all have to be available when an urgent matter arises (bug fixing and the like). We also aim to split our features into small steps which can be progressively added to the production codebase. It can happen that most, if not all, of the team work on a particular feature through the proxy of small but meaningful contributions. My daily work included:

- Identifying and fixing bugs.
- Reviewing pull requests for code style and product issues.
- ES6ify our JavaScript code base.
- Review our logs (detailed in the next chapter).
- Refactor the backend code base with the new architecture in mind.

3.2 Establishment search refactoring

3.2.1 Current status

As of now, the biggest feature I worked on was the refactoring of our search engine. It's also the feature where I contributed the most. Being a crucial part of the website, the search page and the associated search engine were implemented by Privateaser's CTO right at the beginning of the company. In a sense the implementation was a bit "quick and dirty" and it's become legacy code. At the time Vue.js was not being used and there were not many coding conventions.

One of the future features on the roadmap is to refactor the UI and the UX of the search page. Currently the search page is not responsive, which is potentially hindering our conversion rate. Also there hasn't been any data driven approach to how the search experience could be enhanced. For example the following Hotjar heatmap shows how the users interact with the search page on average, but no decision has yet been made as to how to exploit this knowledge.

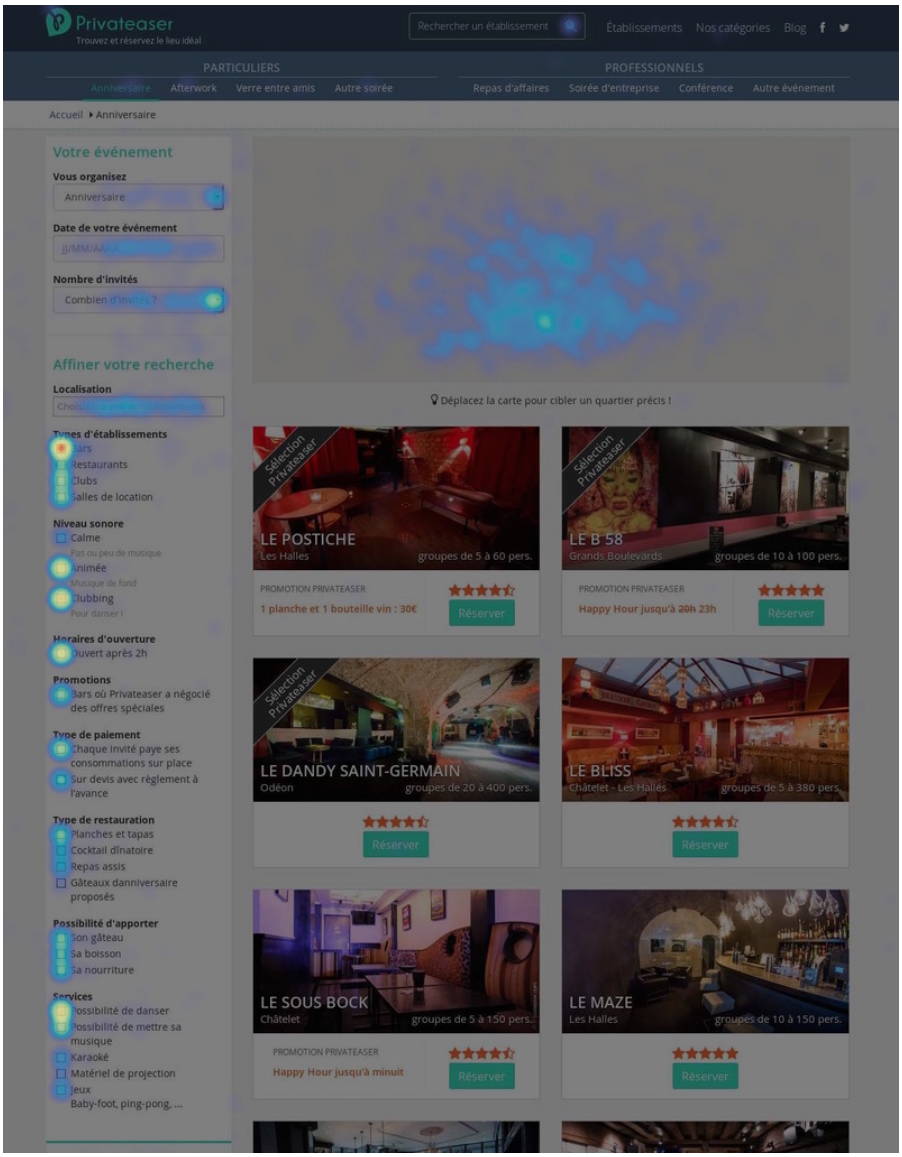


Figure 3.1: Snapshot of the user interaction with the search page

From a development perspective, not much can be done to the search page without an overhaul of the search page. Currently the implementation is difficult to understand and the code is far from being maintainable. Currently the frontend code is a tangle of JavaScript which is extremely difficult to grasp. Every time the user clicks on a filter or pans/zooms the map an AJAX query is sent to the backend and all the matching establishments are returned. It works but it's extremely difficult to build on top of it, thus a refactoring is required.

3.2.2 Goal

The goal is to refactor both the backend and the frontend logic of the search page. The backend is working fine but a tidying up is necessary. The frontend needs a complete

overhaul, specifically data binding through Vue can be used to automatize most of the updating of the UI. This stems from the fact that the establishment cards are supposed to be exactly the same as the markers on the map. It would make sense if a single AJAX would be made and a controller would hold the results before dispatching them to the map and cards automatically. This is one-way binding and is something modern JavaScript frameworks are based upon.

From a code quality perspective the goal is to implement extremely readable and maintainable code because the search page is probably the biggest feature of the Privateaser's website. It's essential that when new developers join the development team they can dig into the code and build on top of it immediately. For the frontend we can make extensive use of the ES6 syntax, both to reduce the size of the code base but also to gain in readability. Furthermore we can create independent LESS files so that frontend development isn't hindered by searching through files for specific CSS classes. There is already a dedicated function in the service layer for filtering establishments, there isn't much refactoring to be made on that side. However refactoring the logic of the frontend will inevitably bring side effects which will have to handled in the service layer.

There is also a strong need to keep the search page SEO friendly as it amounts for a fair share of the traffic on the website.

3.2.3 Architecture

The frontend architecture will revolve around Vue's capabilities. Like other frameworks, Vue encourages developers to think in terms of *components*. For example the search page can be schematized as follows.

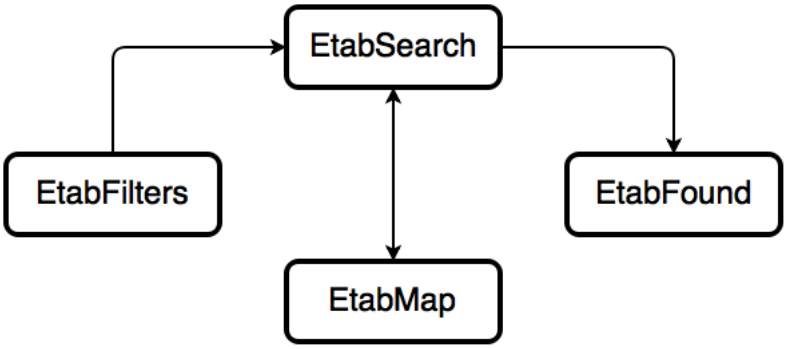


Figure 3.2: Vue components interaction

There are 4 components, each component being responsible for a part of the search page. The filters contained in `EtabFilters` are synced with `EtabSearch` which is the main component supervising the others. There are different kind of filters:

- The kind of establishment (bar, restaurant or rental room).
- The date of the event.

- The number of people (abbreviated to pax) attending the event.
- Boolean values indicating if there is a pool table, if the bar is open after 2AM, etc.

The `EtabSearch` component is responsible for queying the backend and syncing the resulting establishments with the map component (`EtabMap`) and the results component (`EtabFound`). `EtabSearch` is the only component which doesn't possess an associated UI expression because it handles the communication between the components.

3.2.4 User interaction

It's important to define a sort of "story" to understand and predict what users expect from a webpage. In our case a user wants to find an establishment which matches his desires. When she clicks on a filter he expects the interface to update itself. Specifically she wants the establishments to update and the markers on the map to be synced with the cards. She also expects the moving the map and zooming in and out will filter the results accordingly. Finally, she wants results to be displayed progressively as she scrolls down the page (infinite loading).

There are two ways to update the results. Either the user clicks on some filters and moves the map before clicking on an update button, either the results are loaded in real time based on what the user does. Websites like Airbnb have implemented the first option. As we discovered whilst looking inspiration, a lot of the websites who have implemented the second option have done it incorrectly. Indeed we noticed many websites had maps that didn't update the results although it was indicated that they should.

Another feature we want to the search page to have is to be "saveble". Specifically we want the URL in the search bar of the user's browser to update with her interactions. For example if the user is looking for a bar with a dance-floor then the URL should be `www.privateaser.com/reservation-bar?dancefloor=1`. The URL has to be expressive so as to be human readable (something content crawlers reward websites). First of all this is useful for users because they will be able to save a link that answers their needs. For example they can send the link to a friend. Secondly this will enable us to build on top of it and create preset pages which will serve as guides on the website. Finally this is also handy for our consumer service who are often looking for the same of filter combinations. Thankfully updating the parameters of the URL has become trivial with HTML5.

3.2.5 Search Engine Optimization

One of the great concerns of e-commerce websites is to be highly ranked on search engines. To do so the content of each page should be semantically correct and should also contain keywords which are considered "popular".

The way in which content crawlers (for example Google bot) parse a webpage is crucial to understand. It's important to understand that there are different stages

when a webpage loads. Currently most crawlers disable all the JavaScript on a page and only extract the HTML rendered by the server. This means that all client site content isn't noticed by content crawlers. Sadly, Vue doesn't permit server-side rendering. This is a key issue that often arises with modern frameworks. Vue is not yet as mature as React and Angular, and so the tooling around it is considered a bit weak.

The conundrum is that we want to preserve our SEO strengh by server-rendering HTML content whilst at the same making the most of Vue's capacities to synchronize data. The solution we took was to do an initial server-side rendering with unfiltered establishments and immediately removing the generated HTML with our Vue components. This replacing doesn't trigger for content crawlers because they disable JavaScript and thus don't allow Vue to function. One concern is that a "flash" might occur during the split second where the switch occurs, but as will see nothing is visible to the human eye.

3.2.6 Layout

The following figure shows how the Vue components and the initial HTML server-rendering (with Jinja) are organized.

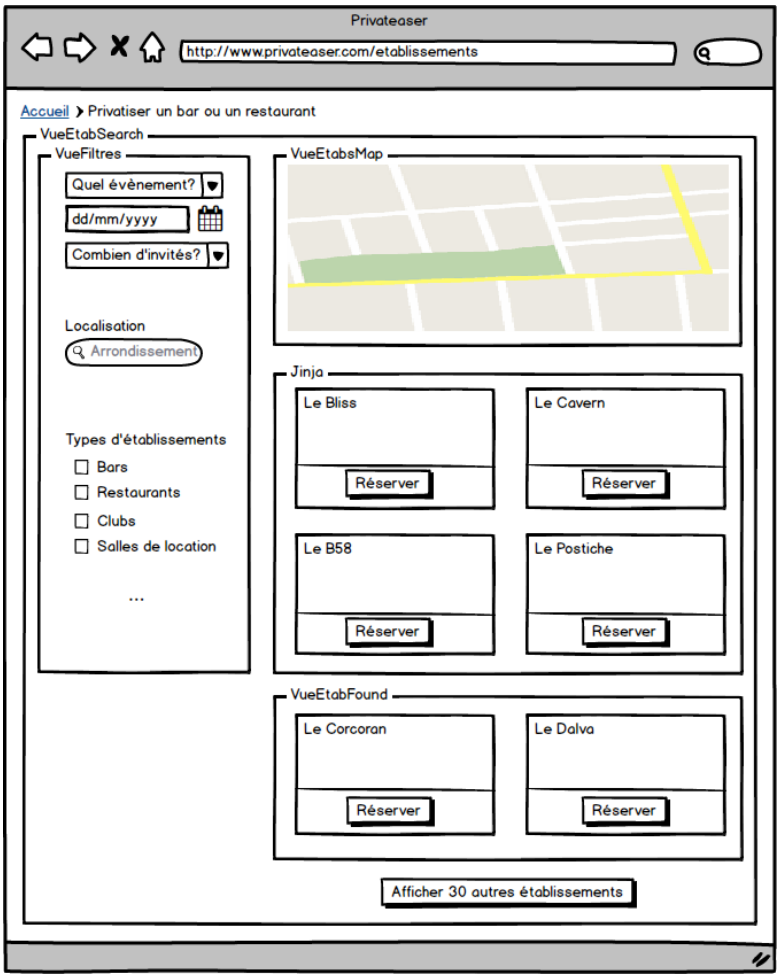


Figure 3.3: Balsamiq mockup

3.2.7 Implementation

Vue components

The first implemented component was the filters components. It's quite simple in the sense that it only has to be synced with the root controller. For this Vue's two way data binding features is extremely useful.

Props are a mechanism for passing data between components. Aside from this each component has it's private data that other components can't and shouldn't access. In this particular case the filters are passed as props and they are two-way binded (`twoWay: True`). This means that whatever happens to the `filters` object, it will be updated both in the filters component and the root controller. The components also notifies the root that it is ready, this is useful for updating the UI and enabling/disabling loading overlays that are useful for making users more enjoyably.

Each object of the component is then linked to an HTML component. When the HTML component changes the data and the props are automatically updated. For

example if the user clicks on a filter it then become `true`, and because it is binded with the root component then the root component also updates. This update mechanism enables us to code in a declarative fashion without having to worry of what happens behind the scenes because Vue handles it for us.

The same ideas prevail for the map component. The gist of it is that the root will query the backend based on the filters from the filters component and then "pass down" the resulting establishments through propping. The subtlety is that the results provided by the backend may (and often do) overlap the current results. Adding a new marker on top of an already existing is not a good idea. First of all the UI result isn't nice, secondly it adds a useless load to the map which is already full of markers. To sidestep this one can use symmetrical differences in the following way:

1. Only keep the new etabs that are not already on the map.
2. Remove the etabs that are not part of the establishments from the map.
3. Add the remaining new etabs to the map.

Luckily Google Map's API provides simple functions for adding and removing markers from a map. Each Vue component can add a `watcher` to any of it's props and data properties to detail what occurs when it changes. In the map component's case we want it to update the map each time it's list of establishment changes.

```
export default {
    props: {
        filters: { type: Object, required: true, twoWay: true },
        state: { type: String, required: true }
    },
    data: () => (
        {
            formattedDate: '',
            dateError: false,
            datepicker: null
        };
    ),
    watch: {
        'filters.date': function(newDate) {
        ↪  this.datepicker.setMoment(newDate); },
        'formattedDate': function(newFormattedDate) {
            const m = moment(newFormattedDate, 'DD/MM/YYYY');
            if (m.isValid()) { this.filters.date = m; this.dateError
            ↪  = false; }
            else this.dateError = true;
        }
    },
    ready: function() {
        this.datepicker = new Pikaday({
            field: this.$els.date,
            minDate: moment().toDate(),
            format: 'DD/MM/YYYY',
        });
        this.$emit('ready');
    }
};
```

Listing 1: Vue filters component JS code

```
<div class="radio filters-radio">
    <label>
        <input type="radio" value="reservation-bar"
        ↪  v-model="filters.universe">
        Reservation de bar
    </label>
</div>
<div class="radio filters-radio">
    <label>
        <input type="radio" value="restaurant-groupe"
        ↪  v-model="filters.universe">
        Restaurant pour groupe
    </label>
</div>
<div class="radio filters-radio">
    <label>
        <input type="radio" value="location-salle"
        ↪  v-model="filters.universe">
        Location de salle
    </label>
</div>
```

Listing 2: Vue filters component HTML code

```
watch: {
    etabs: function() {
        for (let etabID in this.markers) {
            if (!(etabID in this.etabs)) {
                this.markers[etabID].setMap(null);
                delete this.markers[etabID];
            }
        }
        for (let i = 0; i < this.etabs.length; i++) {
            const etab = this.etabs[i];
            if (!(etab.id in this.markers)) {
                const marker = new google.maps.Marker({
                    position: { lat: etab.latitude, lng:
                    ↪ etab.longitude },
                    map: this.map
                });
                (function(etab, marker, map, self) {
                    google.maps.event.addListener(marker, 'click',
                    ↪ () => {
                        self.infoWindow.setContent(
                            self.getInfoWindowContent(etab)
                        );
                        self.infoWindow.open(map, marker);
                    });
                })(etab, marker, this.map, this);
                this.markers[etab.id] = marker;
            }
        }
    }
}
```

Listing 3: Updating the markers on the map

Just like the filters component, the map has to send the current map coordinates to filter the establishments. Yet again two-way binding can be used. In a sense each sub-component (the map and the the filters) contains a subset of the filters contained in the root component.

To sum up, every time the UI changes because of user interaction (because of a click on a filter on a zoom on the map) the data contained in the Vue changes automatically. On top of this we add watchers to specify what should happen when the data changes.

```
capturePosition: () => {
    const bounds = this.map.getBounds();
    this.latMin = bounds.getSouthWest().lat();
    this.latMax = bounds.getNorthEast().lat();
    this.lngMin = bounds.getSouthWest().lng();
    this.lngMax = bounds.getNorthEast().lng();
}
```

Listing 4: Capturing the map boundaries

Each time the sub-components update the root component, a trigger is actioned to go fetch matching establishments. Again, this is done through a `watcher`. The following code snippet is the result of a lot of tinkering with the goal of mind of making the code readable and high-level. The `patchUriWithParams` does exactly what it's called. The advantage is that by creating external (somewhat generic) functions, the code becomes readable and a new developer can quickly understand how the data flows. The `internalFetch` is a tool we developed later to reduce the boilerplate code to perform an AJAX query to the backend. It takes as parameters an HTTP method and a URL and returns JSON data.

Filtering establishments

The `internalFetch` method in the frontend calls upon an API path called `api/etabs`. This path provides a link between the JavaScript frontend and the Python service layer. Specifically it taps into a method called `etab.filter` which takes as input a dictionary of filters and outputs matching establishments.

We make good use of the SQLAlchemy ORM with which we can conditionally filter an initial set of establishments with the *filter* method. Behind the scenes SQLAlchemy will translate the filters into real SQL code and run it when we call the `.all()` method.

Infinite scrolling

As was the case before and as it should remain, the user should be able to obtain more establishments if he scrolls down the page. An naïve implementation would be to return all the establishments when an AJAX call is made and then display them progressively with some JavaScript magic. However this seems quickly unreasonable because of the networking cost because As of May 2016 Privateaser has more than available 800 establishments. It currently manages the load but it is in no way scalable.

A smarter way of doing is to use an offset/limit method. The basic idea is to send streaming data to the frontend based on how far the user has scrolled down the page. For example if 250 establishments match the user's filters, there isn't much sense in loading each establishment because there isn't even enough space of the page to display them. The answer is to paginate the data. Classic pagination means that the

```
fetchEtabs: function(append) {
    this.state = 'loading';
    // Build the URL
    let fetchUri = URI(window.location.origin);
    fetchUri.pathname('api/etabs');
    fetchUri.setSearch({ 'universe': this.params.universe });
    patchUriWithParams(fetchUri, this.params);
    // In append mode (scrolling), add offset; otherwise (changing
    ↪  filters), reset
    if (append) this.offset += 30 + this.skippedEtabsCount;
    else this.offset = 0;
    fetchUri.setSearch({ 'offset': this.offset });
    // Call the API
    internalFetch('GET', fetchUri.toString())
    .then((json) => {
        if (!append) this.etabs = json.etabs;
        else this.etabs.push(...json.etabs);
        this.totalEtabsCount = json.total;
        this.skippedEtabsCount = json.skippedEtabsCount;
        this.updateHeaderText();
        this.state = 'ready';
    });
}
```

Listing 5: Fetching establishments from the backend

user can click on a "Go to the next page" button to view more results. More modern applications then to implement infinite scrolling so that the user doesn't have to click. This makes good UX sense; for example Google has shown how most users never go on the second page of list of Google results. Popular websites like Facebook and Instagram have implemented infinite scrolling to great success.

Code-wise the idea to use two variables called `offset` and `limit`. At the beginning the `offset` is worth 0 and the `limit` is worth k (with k usually around 30). When the page loads the first k establishments matching the current filters are added to the page. When the user scrolls down the page and has just about finished viewing the current k establishments, the `offset` and the `limit` both increase by k so that `offset` is worth k and the `limit` is worth $2k$. A new AJAX is then sent to the backend with the updated `offset` and `limit` values. Database-wise this works very efficiently because the SQL backend never has to return all the matching establishments. Thus the load is reduced between the backend and the frontend and also between the database and the backend.

An area of concern is to how to know exactly when to trigger the update of the `offset` and the `limit` variables. It took a few iterations and some feedback from the

```python
q = self.find(query=True)

if params.get('universe') == 'bar':
    q = q.filter(Etablissement.type_bar == True)
elif params.get('universe') == 'restaurant':
    q = q.filter(Etablissement.type_restaurant == True)
elif params.get('universe') == 'rental_room':
    q = q.filter(Etablissement.type_rental_room == True)

establishments = q.all()
return establishments
```

Listing 6: Applying filters in Python

users to make the transition smooth.

Managing availabilities

One of the trickiest things we had to handle was the availabilities of the establishments. Basically each establishments has a limited number of spaces on any given date. Specifically there is a dedicated set of functions to answer the question *Is establishment x available on data y for z customers?*.

When all the matching establishments were loaded to the search one simply had to call this set of functions for each establishment. However, because we are now promising of sending batches of k establishment to the frontend, we have to handle this in a different way.

The frontend *expects* to receive k establishments. The backend can be naïve and simply find the establishments between the `offset` and `limit` values before filtering them based on the availabilities. However if some of the establishments are not available then less than k establishments will be sent to the frontend. It's each to solve this problem if the entire set of establishments is available because one could filter the establishments b availability before applying the `offset` and `limit` variables.

The issue is that the database returns the establishments in chunks of size k. The solution is to query a chunk and check for it's availabilities. If enough establishments then they can be sent to the frontend. If there are not enough then we increase the `offset` and the `limit` and query a contiguous chunk. The matching establishments in the new chunk are appended to the initial chunk. The process is repeated until enough establishments are found and while there are remaining establishments in the database.

The goal of this way doing is to limit the bandwidth between the database and the backend. The bandwidth is already reduced between the backend and the frontend because the infinite scrolling paradigm. The naïve solution to solve the availability

issue would be to query all the establishments from the database and filter them in the service layer with Python. However this would max out the bandwidth between the database and the backend. The "chunk method" that I proposed strikes the right balance between speed and memory efficiency. Being a heuristic method it's difficult to know exactly how many establishments to query per chunk to obtain the desired number of establishments. Querying the establishments one-by-one in a stream-like fashion would be good bandwidth-wise but the number of queries would explode. A simple heuristic we adopted was to query 10% more than the k needed number of establishments per chunk. We call this `higher_limit` in the following snippet.

Initially this worked until we realized that we needed to know how many establishments were not available for subsequent calls. There was a bug on the website where establishment would appear twice on the search page. This was due to the fact that we didn't memorize how many chunks we had checked, subsequent calls would then tap into already checked chunks and would return establishments that were already loaded in the past. It took us quite some time to figure where the bug was coming from.

```
higher_limit = limit * 1.2
results = []
skipped_etabs_count = 0

while len(results) < limit and offset < total:
    chunk = q.offset(offset).limit(higher_limit).all()
    if params.get('date'):
        occupancy = s.ea.get_etabs_occupancy(chunk, params['date'])
        for etab in chunk:
            if occupancy[etab.id].free_pax > (params.get('pax_min')
            ↪  or 0):
                results.append(etab)
            else:
                skipped_etabs_count += 1
            if len(results) == limit:
                break
        chunks_lengths.append(len(results))
        offset += higher_limit
    else:
        results = chunk[:limit]

return results, total, skipped_etabs_count
```

Listing 7: Filtering establishments with the chunk method

Conditionally Disabling content crawlers

Another issue is that by nature content crawlers try to index every possible URL on a website. However the number of indexed pages is limited per website. This means that if two pages with distinct URLs share the same content then it's redundant to allow content crawler's to index both pages. In our case, we are appending parameters to the URL, thus we have extremely high number of pages who have the same content. For example slightly changing the geographical filters doesn't result in different results and yet the content crawlers will treat the resulting pages as a different one.

Our temporary solution is to completely disallow web crawling for URLs associated to the search page that contain parameters. This way only the unfiltered results will be indexed. This works well because the unfiltered results are supposed to display the 30 establishments of a given category with the highest ratings, and thus who possess a strong SEO impact. Later on, when implementing filter presets we can start allowing the crawlers to index certain pages because the page has be tailored for that particular SEO purpose. Specifically the access can be controlled by modifying the `robots.txt` of the application and by conditionally adding a `noindex` tag in the header of the search page.

3.2.8 Testing

Our development process doesn't require us to write unit tests for every single feature we implement. Because of this we don't have a testing suite or any strong opinions on how to run our tests. Python however possesses many testing frameworks such as `unittest`, `pytest` and `nosetests`.

I decided to implement some simple unit tests to make sure each filter on the search page is working from a service layer point of view. The idea is simply to check that by checking a filter the number of matching establishments is lower than the total number of establishments. This both serves the purpose of making sure the service is layer is functional when new features are implemented on top of it, but also as a proof of concept for future testing implementations. Specifically we agreed I could try implementing the tests with `pytest`, it being the most popular Python testing framework. The nice thing about `pytest` is that it doesn't impose it's own grammar. Conditional logic can be written in plain Python and that's it.

Something specific I got accustomed with was the whole concept of setup and tear down of a test. The idea is that each test should build itself from the ground up and build all the pieces necessary for it's specific scope. Testing on an example database isn't a good idea. First of all side effects can occur (sending involuntary mails for example) and secondly the database's state isn't known if it's not been setup right before the test. In our case the setup consists in creating an in-memory database (with SQLite) and refiling it's content with predictable data.

We are slowly leaning towards implementing many end to end tests with the popular `Selenium` framework. Specifically the idea is to mimick the behavior of a user by instructing a robot to programmatically perform actions on the websites and check that the UI responses are coherent.

```
class BaseTestCase(object):

    @classmethod
    def setup_class(cls): the new engine
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
        database.engine =
        ↪  create_engine(app.config['SQLALCHEMY_DATABASE_URI'])
        database.Base.metadata.create_all(database.engine)
        database.db_session =
        ↪  scoped_session(sessionmaker(bind=database.engine))
        app.config['NOTIFICATION_EMAIL_ADDRESS'] =
        ↪  'nobody@privateaser.com'

class TestEtabFilter(BaseTestCase):

    s = ServiceContainer(db_session(), app.config, app.jinja_env,
    ↪  logger)

    @classmethod
    def setup_class(cls):
        super(TestEtabFilter, cls).setup_class()
        cls.params = {}
        _, cls.unfiltered_count, __ =
        ↪  cls.s.etab.filter(**cls.params)

    def test_etab_filter_universe(self):
        universes = (
            'bar',
            'restaurant',
            'rental_room'
        )
        for universe in universes:
            self.params['universe'] = universe
           _, filtered_count, __ = self.s.etab.filter(**self.params)
            assert filtered_count < self.unfiltered_count
```

Listing 8: Testing the filtering in the service layer

3.2.9 Results

Implementing the new search engine was an extremely enriching personal experience. I already a lot of the concepts I used but putting them in place and seeing them put into production was undeniably satisfying. The peer reviews put things into perspective and allowed to really understand what was to be expected from a SOA architecture and allowed me to better myself in JavaScript. I'm happy to have been able to implement an infinite scrolling system and to have proposed the chunk algorithm we used. Without being too pretentious the search page is working extremely well and it's much more advanced that a lot of other more popular websites.

From a business standpoint the UI or the UX didn't really. However we noted a reduction in average querying time from 1.5 to 0.5 seconds, which is quite huge. This is most certainly to the more efficient bandwidth usage obtained with the offset/limit usage and the chunk method.

From a developer perspective the code is much easier to get into. Funnily enough when I was done with the code was put into production we noticed that there were more deletions (1725) than insertions (1668). In a sense this is good news because the application is doing the same job as before, but in a more efficient manner (speed and bandwidth) and with less lines of code.

3.3 Lazy store

Another feature I got to work was implementing a pattern for querying and memoizing queries. This stems from the fact that there are pages in the back-office where queries are performed multiple times, mostly because of weak organization of Vue components.

It sometimes occur that a Vue components queries the list of establishment of customers to put them in a select box or some fancy UI widget. Normally a parent Vue component should perform the necessary queries and should pass the data to it's child components by propping down the data, just like we did with the search page. One of the solutions would be to re-organize the Vue components so as to follow the recommended patterns. However this is a lot of work and we decided to go a different way.

The design pattern we agreed upon was similar to a *singleton*. The idea is that there is no sense in performing a query that has already been performed on the same page. This is where memoization comes into play. We want to create an object that can store and return the result of a query in *lazy* manner. The usage of the object should be transparent. We should be able to ask the object for a result and the object should decide what it has to do based on the it's current state.

We decided to call this object `lazyStore`. After some iterations and some discussions we decided it should have 4 states:

1. `empty`: The `lazyStore` doesn't contain any data and has never been asked to do anything.
2. `loading`: The `lazyStore` has been queried and it is fetching data.
3. `ready`: The `lazyStore` has finished it's first query and has stored the data internally.
4. `failed`: The `lazyStore` couldn't perform the initial query.

The first query will put the `lazyStore` in `loading` state. The next queries will then understand that a similar query has already been made and that they have to wait. If the `lazyStore` is `ready` then it doesn't have to re-do the query, it simply returns the stored data.

We went a bit further in the sense that we used the modern concept of *promises*. A promise is a object that can be used even it doesn't *exist*. Promises are difficult to grasp at the beginning and they can be a headache to put in place. The fact is that they should be used in modern applications for fetching data asynchronously. Basically a promise can be passed and used in functions until it's content is *really* needed. In this sense promises allow efficient lazy usage of data. Wrapping the whole promise machinery inside the `lazyStore` allows it's external use as a blackbox.

```
function LazyStore(loadFunction) {

    this._state = 'empty';  // empty, loading, ready, failed
    this._data = null;
    this._loadingPromise = null;

    this.getData = function() {
        if (this._state === 'empty') {
            this._state = 'loading';
           this._loadingPromise = new Promise((resolve, reject) => {
                loadFunction().then(
                    value => {
                        this._state = 'ready';
                        this._data = value;
                        resolve(this._data);
                    }
                ).catch(
                    reason => {
                        this._state = 'failed';
                        reject('loadFunction failed');
                    }
                );
            });
            return this._loadingPromise;
        }
        else if (this._state === 'loading') {
            return this._loadingPromise;
        }
        else if (this._state === 'ready') {
            return new Promise((resolve, reject) => {
                resolve(this._data);
            });
        }
        else if (this._state === 'failed') {
            return new Promise((resolve, reject) => {
                reject('loadFunction failed');
            });
        }
    };
}
```

Listing 9: Lazy store JavaScript implementation

The usage of the `lazyStore` is extremely simple. We basically have a `stores.js` file containing a store for each query towards that it often performed. The store can then be importing like any other JavaScript object between files. The last 3 lines of the following snippet clearly show how easy the usage is.

```
const allEtabs = new LazyStore(() => {
    fetch('/admin/api/etabs/find',
        {
            credentials: 'same-origin'
        }
    )
    .then(r => r.json())
    .then(data => data.etabs);
});

allEtabs.getData().then((data) => {
    // Do something with the data
});
```

Listing 10: Lazy store example usage

3.4 Validating payloads

Python is an interpreted language. In this sense the type and structure of objects it handles is only determined at runtime. The issue that often arises is that incoming data towards the API (called *payload*) doesn't have to be shaped of any particular way for an API endpoint to execture itself. However, each API endpoint is *expecting* the payload to contain certain parameters for it to behave correctly. Typed languages such as C, Java and Go don't have this problem because the data has to be unmarshalled; unstructured data isn't permitted.

The business issue is that we don't want to users to be able to input, for example, invalid email addresses and telephone numbers. Code wise there is a discrepancy between what the frontend returns, for example french formatted dates and times. Dates and times are usually manipulated through Python `datetime` objects. This is a simple example but the point is that incoming has always got to be transformed so it can suit the backends programming language data types. This is because JSON only strings and numerics.

Converting between universal data types and programming language specific types is called *coercion*. Coercion is automatically through type unmarshallig but it isn't something natively integrated to Python. Usually the data is parsed when needed. The ugly truth is that the API required a certain format but it accepts just about anything. To solve this problem we decided to use a data validation library called `voluptuous` which solves two problems:

- It makes sure a dictionary has a certain structure.
- It converts the dictionary variables as necessary.

In the following example the payload associated to an event creation is validated with `voluptuous`. The declaration is declarative, which is quite reasonable regarding data validation. The example is quite in the sense that the incoming payload doesn't contain any imbrication. However, a nice feature of `voluptuous` is that schemas can be composed so that complex data can be validated by using already existing schemas.

```
@app.route('/api/events', methods=['POST'])
@json_response
def event_create():

    validator = Schema({
        Required('event_date'): Date('%Y-%m-%d'),
        Required('event_time'): Time('%H:%M:%S'),
        Required('pax_min'): All(int, Range(min=0)),
        Required('pax_max'): All(int, Range(min=0)),
        Required('booking_mode_key'): Any(unicode, None),
        Required('booker_first_name'): Any(All(unicode,
        ↪  Length(min=1)), None),
        Required('booker_last_name'): Any(All(unicode,
        ↪  Length(min=1)), None),
        Required('booker_phone_number'): Any(All(unicode,
        ↪  PhoneNumber), None),
        Required('booker_email_address'): Any(All(unicode, Email),
        ↪  None),
        Required('booker_description'): Any(unicode, None),
        Required('requires_quotation'): bool,
        Required('etab_id'): int,
        Required('promo_id'): Any(int, None)
    })

    data = json.loads(request.data)

    try:
        data = validator(data)
    except MultipleInvalid as e:
        raise ValueError('Invalid incoming payload')
```

Listing 11: Validating the payload associated to an event creation

voluptuous allows and encourages developers to write custom validators. For example the following snippet shows how to coerce string to date and time objects and email and phone numbers.

```
def Date(fmt):
    return lambda v: datetime.datetime.strptime(v, fmt).date()


def Time(fmt):
    return lambda v: datetime.datetime.strptime(v, fmt).time()


def Email(msg=None):
    def validate(value):
        if '@' in value:
            return value
        else:
            raise Invalid(msg or 'Invalid email address format')
    return validate


def PhoneNumber(msg=None):
    def validate(value):
        if re.match(ref_data.PHONE_NUMBER_FORMAT_REGEX, value):
            return value
        else:
            raise Invalid(msg or 'Invalid phone number format')
    return validate
```

Listing 12: Custom voluptuous validators

To sum up, every time an API query is performed, the incoming payload is validated and coerced. If one of the two operations fails then an error is raised; the error has a 500 code attached so it can easily be understood and communicated to the user if necessary. Coupled to the fact that we can also sync the validators with the JavaScript frontend by using the regex validators this renders our API tight and should drastically reduce the quantity of incoherent data all across the application because we are doing the coherency checking right at where the data comes in.

Chapter 4

Data science

Privateaser is not yet big enough to employ full-time data scientists to make data-driven decisions. It does however encourage developers to backup their assertions with data. Moreover other parts of the company, particularly the marketing team, are avid for data and for insights into how the customers are behaving.

During the internship I spent some time during the offsprint to put in place some foundations for future data-driven work. I didn't really get to any advanced data analysis, I mostly got to organize the way in which would be analyzed when the time would be right.

4.1 Creating a data science code base

It seemed quite obvious that whatever the way the data was analyzed it should be done in a manner independent from the main application code base. Specifically it should tap into the database and logged data. We settled upon using Python 3 for manipulating the data. Python is growing popular inside data analysis communities because of it's wide range of data analysis tools.

Specifically I put in place a tool chain for parsing and running data analysis techniques with the `pandas` library; it allow the manipulation of dataframes and can easily connect to an SQL database to fetch required data.

The idea is that for each statistical analysis a single script can be associated to it. The scripts can then be run by a cron job on a dedicated server. This is practical because the statistical analysis that are required usually have a temporal dimension so as to study progress.

Physically there is a separate code base versioned with git. The code base contains a folder when scripts can be added. A main Python file runs each scripts and a cron job calls the Python script on a nightly basis. The output of each script (usually an Excel file or a chart) is a transferred towards a dedicated folder accessible via the `www.data-science.privateaser.com` URL for IP addresses that arise from the Privateaser office space. This way non-developers can access fresh results on a daily basis.

As of now some interns have been recruited to analyze data. They are currently getting used to Python, but when they are ready they will use the data science repository for their analysis so that other people can access their insights. This is a good solution to avoid that their work gets forgotten just because they didn't have any proper way of putting their scripts into production.

4.2 Analyzing log files

As of the beginning of my internship, there is a huge desire to pin down the various bugs all over the website. What's more we would like to monitor the traffic and understand what are the "peak hours". Basically we would like to start *logging* events on the website. This is something most applications require at some point in time. The earlier the better.

4.2.1 Producing log files

The general idea is that we want to be able to monitor any event we want. We can distinguish passive logging from active logging. The passive logging should monitor most of the traffic on the website. It should tell who went where when and with what device. The active logging will encompass less frequent events, such as when an event is created or cancelled.

Python's standard library offers a basic logging toolbox. It allows to log where and when an event was logged, along with metadata such as how critical the event is (information, error, warning, etc.). However it isn't very flexible and it doesn't "feel right". Logging events is good but it is useless to do it without considering how the logs will be analyzed. Grepping through logs is a poor man's way (and yet a popular one!). A better way to analyze logs is to use dedicated platforms (SaaS or custom). In any case the format of the files generated by the logging process is extremely important. A good practice is to find a balance between both human and computer readability. For this a solid practice seems to settle on JSON output. This way the logging format is universal and logs ranging from simple to imbricated structure can easily be constructed.

After doing some research I found that a Python library called `structlog` did exactly what we wanted. Implementing a logger revealed itself easy. A nice feature we decided to implement was to automatically log the date and time of log message along with the logging level and more importantly a session ID and an IP address. Basically we decided to store a random string into each user's session (by using cookies). This allows to track the users when he navigates through the website. This should allow us the better understand why and how bugs occur when users with the website.

The code of the logger's implementation isn't very interesting. It's more it's use and it's application that are noteworthy. It's been implemented in a way where logs can be initialized and appended to with conditional logic, such as in the following snippet. The advantage is that it avoids having to duplicate code in each `if` statement. Of course this is simply a comfort feature but at least it encourages developers to write logs, which can be perceived as quite a burden.

```
log = logger.new(event_id=event.id)

if alt.status == 'dropped':
    log.info(u'Event dropped')

elif alt.event.status == 'missed':
    log = log.bind(event_date=event.event_date_time)
    log.info(u'Event missed')
```

Listing 13: Conditional logging

4.2.2 Creating log dashboards with Logmatic

Logmatic is a *SaaS* (Software as a Service) that allows to parse and analyze log files by directly linking to a data source. In our case we set up a rotating file handler on our Amazon instance and piped Python's logs towards the file handler. Logmatic fetches the logs in a near real-time manner. I took the time to compare similar tools to Logmatic such as Loggly and Logentries but we agreed that Logmatic did the job, at least at our scale. It being based on D3.js, it's also visually appealing. Another solution we explored was to use the popular ELK stack (Elastic Search - Logstack - Kibana) but it seemed overkill and it required much more setup.

We want to help Logmatic gather insights for us. We want to be able to analyze and logs and build useful charts and leaderboards to examine what is going wrong on the website and to what extent. The following examples are a proof of a concept of what is possible and they serve as encouragement for other developers to log events. As can be seen Logmatic makes going through logs much more fun than running `grep` commands in the shell.

Logmatic automatically extracts fields from lines in a log file if they are of a certain format. For example JSON formatting is supported out of the box. We can then filter the logs if they contain a certain field or if a field in question matches a regular expression. Field filtering can then be filtered into *metrics*. For example each web request's duration time is logged for slow requests. The durations is logged as a floating point, the associated metric is in milliseconds. The metric can then be aggregated into charts, it can also be sliced and diced by comparing it against other metrics. Metrics can be aggregated on a given period of time. The following charts were aggregated over 7 days during the month of May when we started using Logmatic.

Endpoint total duration

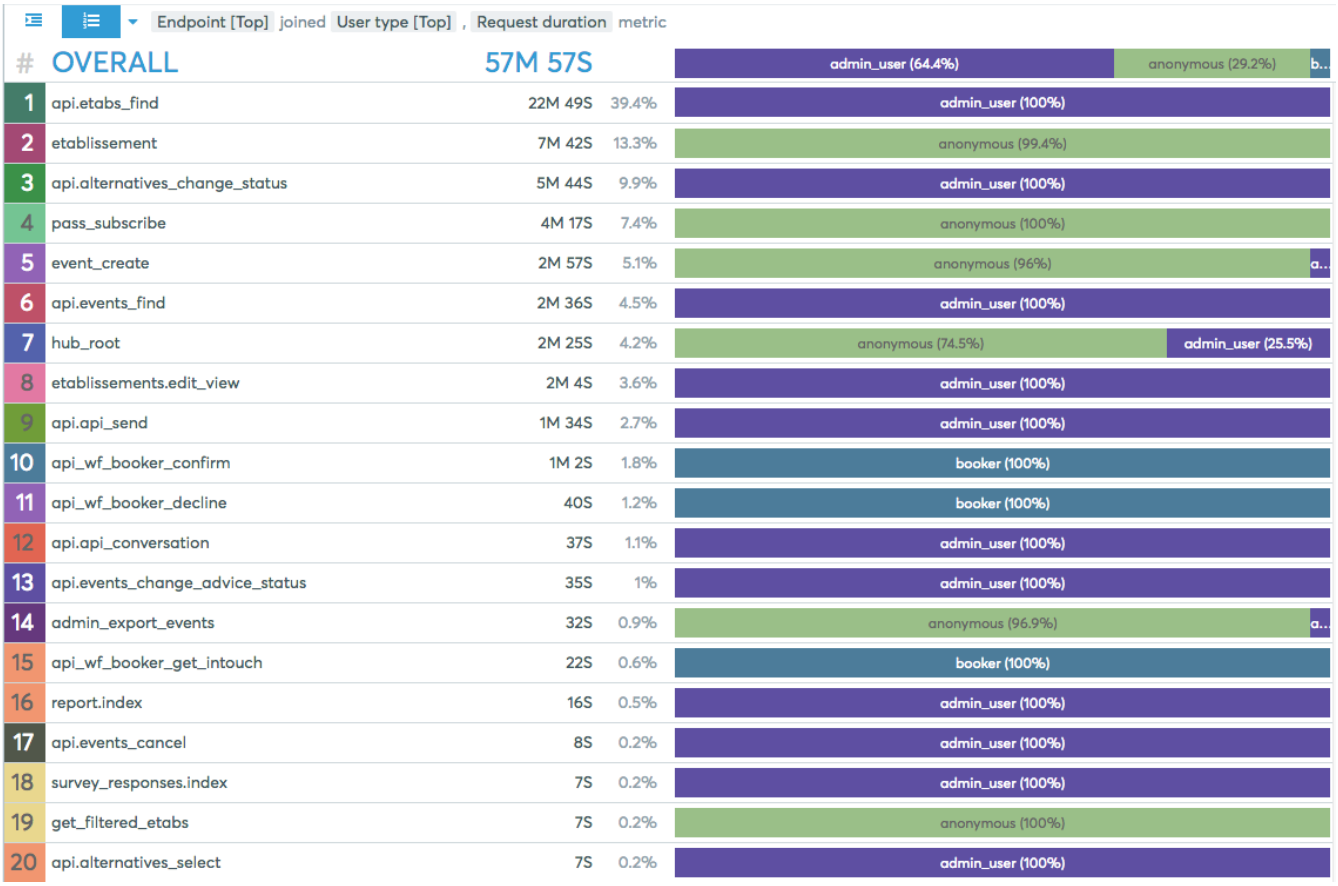


Figure 4.1: Endpoint total duration

This first chart aims to show where the slowest endpoints on the website. The endpoints are sorted by total duration during 7 days. The usage of each endpoint has been split based on the type of usage. Slow endpoints are more critical for public users (marked as ananymous) because a bad UX will result in lower conversion rates. This dashboard helps us to the target the slowest parts of the website before doing something about it. Of course we could compare two periods of time to measure how our optimizations are doing.

The first route is not of importance because only the admin users are using it. However, the second path is exclusively accessed by the public users. Luckily this has greatly improved since thanks to the new search mechanism introduced earlier in this report. We have also started logging the time spent performing SQL queries to understand where the bottlenecks are coming from. Logmatic allows to combine metrics, in this case we decided to compute the difference between the total duration time and the time spent in the database to obtain a good approximation of the rendering of each page.

Average event creation per hour

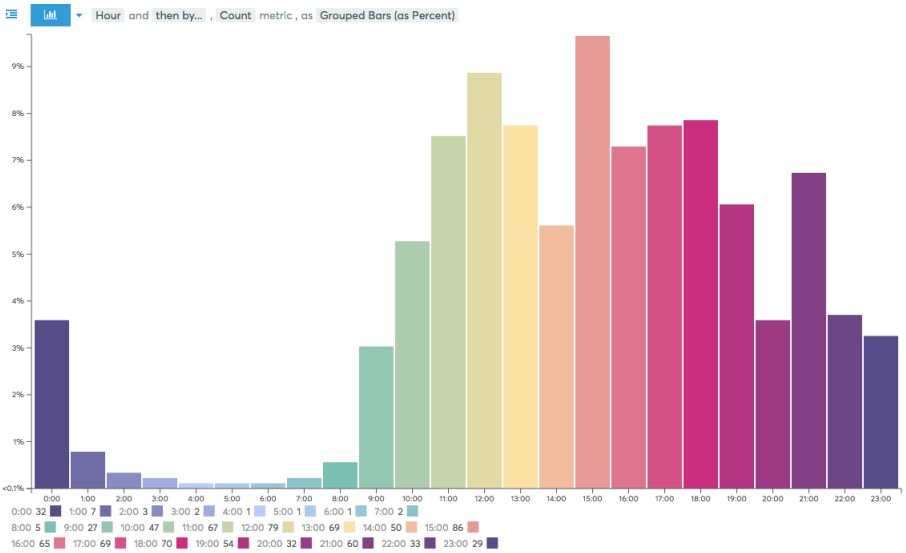


Figure 4.2: Event creations per hour

This chart shows the average number of event creations per hour of the day. This of course was common knowledge but it felt good to back up our feelings with a chart. Not surprisingly reservations are mostly done during the day and when people have breaks. Indeed, users usually prepare some time before making a reservation because it's usually an important decision to make. This cascades onto the following chart with shows the usage of the website per user type. As can be seen the website is mostly used by the public users, which makes it even more important to cut down their waiting time.

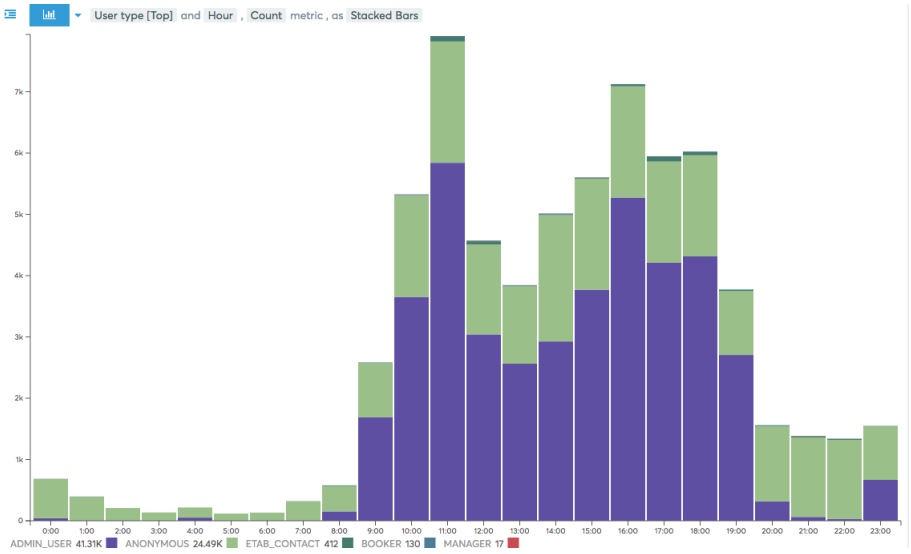


Figure 4.3: Usage per user type

Dashboard example

Figure 4.4: Usage dashboard

Logmatic allows to create dashboards where one can aggregate analysis that deal with the same topic. This enables anyone to get a global view on the topic. This particular example intends to show the usage of the website based on different metrics.

The top view shows the usage of the backoffice per user. It's a bit intrusive but we found it quite funny.

The bottom left chart is a Marimekko diagram that shows the activity on the website split between the day of the week and the hour of the day. It's interesting because it shows that the usage isn't similar between days, neither is it between hours.

4.3 Transactional mail frameworks comparison

Our life as developpers sometimes makes us have to decide between which library/framework to use for a particular task. Pure programming are often opensource, thus the decision often depends on some qualitative attribute. However more business oriented tools come with a cost.

Privateaser sends a lot of emails. When I arrived we were using Mandrill for sending them. It's robust and it handles the load. However, their payment plan changed and we wanted to decide if it was worth changing frameworks based on the change. First we identified potential replacements:

- Postmark
- Sendgrid
- Amazon
- Sparkpost

Each candidate had a pricing table on their website. I decided to make a chart that compared the pricing of each framework to help us decide, at least regarding the cost.

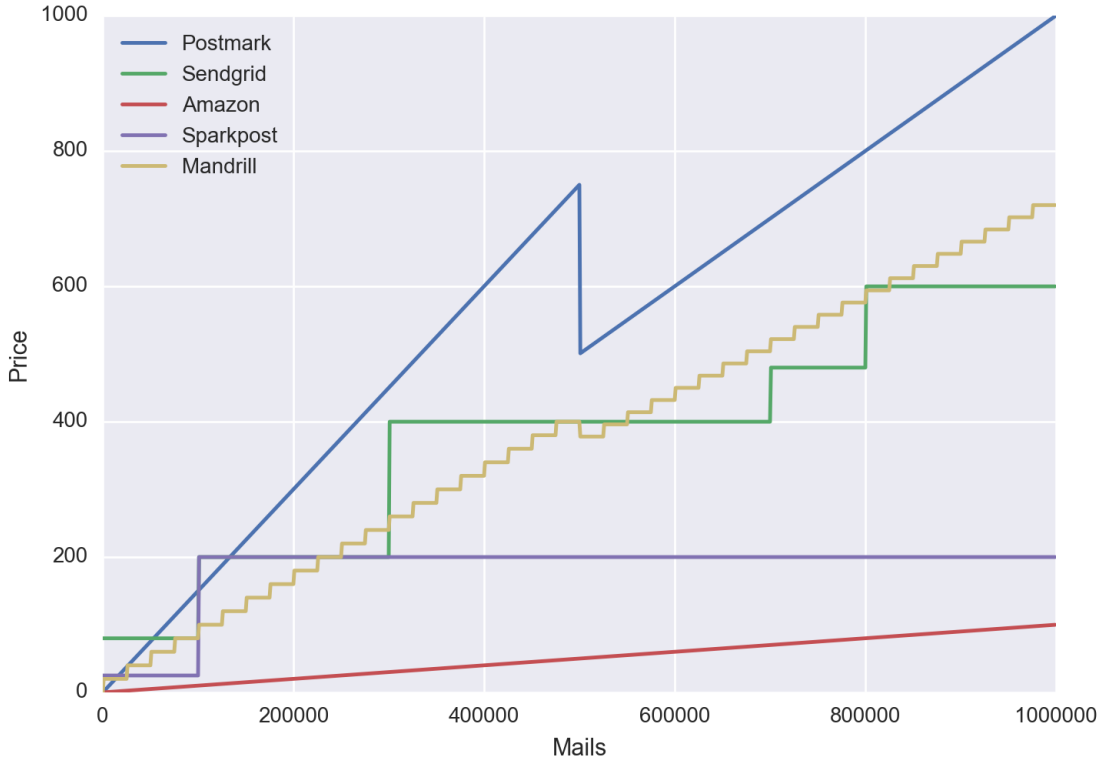


Figure 4.5: Transactional email providers comparison

Each and everyone of the frameworks functions with a breakpoint system. Past a certain threshold the price per email is lower, or the price per n is lower. By translating these pricing tables into functions that take as parameter a number of emails I was able to produce the previous figure. In the end we decided to stick with Mandrill because our current number of sent emails didn't make it ridiculously more expensive than other frameworks.

```
def postmark(n):
    blocks = math.ceil(n / 1e3)
    if n <= 5e5: return 1.5 * blocks
    elif n <= 1e6: return 1 * blocks
    elif n <= 2e6: return 0.75 * blocks
    elif n <= 5e6: return 0.5 * blocks
    else: return 0.25 * 5e2 + postmark(n - 5e6)

def sendgrid(n):
    if n <= 1e5: return 79.95
    elif n <= 3e5: return 199.95
    elif n <= 7e5: return 399.95
    else: return 399.95 + sendgrid(n - 7e5)

def amazon(n):
    blocks = math.ceil(n / 1e3)
    return 0.1 * blocks

def sparkpost(n):
    if n <= 1e5: return 24.99
    elif n <= 1e6: return 199.99
    else: return 199.99 + 0.15 * (n - 1e6)

def mandrill(n):
    blocks = math.ceil(n / 25e3)
    if blocks <= 20: price = 20
    elif blocks <= 40: price = 18
    elif blocks <= 80: price = 16
    elif blocks <= 120: price = 14
    elif blocks <= 160: price = 12
    else: price = 10
    return price * blocks
```

Listing 14: Transactional email providers comparison

Conclusion

Privateaser was an important step for me. Although I haven't been trained to a web developer, I had already worked on a my spare time on web programs and this internship was a chance to challenge my knowledge and give me an opportunity to apply to the real world. My previous internship had been more research oriented, with less business impact. On the contrary, Privateaser offered me a chance to build towards a product used by many users, with important ramifications regarding people's lives.

I feel much more confident working with other people. I now have a better understanding of how to manage intermediary between the business necessities and the code implementation. My knowledge of Agile principles has increased, I got the chance to apply my knowledge in the domain in a real case.

My fellow developers helped me better myself as a programmer. I now have a good conception of SOA applications, micro-services, coding conventions, git flows, peer programming, code reviews, monitoring...

As a data scientist I didn't really get the chance to do anything advanced but I helped to build towards a culture of being data driven. I helped put in place the foundations of our data science tools and I understood how useful data analysis regarding business prospects.

All in all Privateaser was a great experience, I got to meet wonderful people who were a bit older and with whom I had a great time. I would definitely re-iterate the experience of working for a startup.

Appendices

Appendix A

The tools

It's worth mentioning the tools that are and will be used during the internship. Indeed, these deeply affect the workflow and the way code integration is done. By tools I mean programming languages, software and APIs. The bulk of the code is written in Python, of which the flexibility cannot be overstated. One of the major benefits of Python is that it easily set up and very readable, simplifying integration and code sharing. A plethora of libraries and drivers have been written in Python to perform data science, deploy website, provide layers to databases... Privateaser's applications are powered by the Flask web framework, which incorporates most tools for building a modern website (parametrized URL routing, HTML templates, caching, redirecting etc.).

Python

Flask

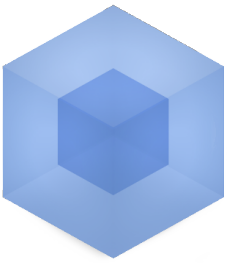
Regarding the frontend, we tend to use Vue.js. It is proven excellent for the back-office where many actions are triggered based on user actions, for example when new events or SMSs are refreshed. Vue.js is very much like React.js, it encourages to build with a component design. Every component is responsible for a part of the interface. Components can be composed so as to build complex structures. Vue.js emphasizes reactive design, where the UI changes in real time based on user actions. Vue.js also encourages to think of the interface as a pure result of the data. Little by little we are replacing legacy jQuery code with Vue.js. It's much more developer friendly also. Indeed, the HTML, JavaScript and CSS can all be included in a single where the full use of conditional logic can be used to modify the UI. Before frameworks such as Vue.js and React.js, developers tended to target their HTML divs with jQuery. This is fine but spaghetti code is naturally formed because there is no clear link between the HTML and the JavaScript. The HTML doesn't *know* it's being target by a JavaScript directive until it is, which makes it hard to grasp by a new developer.

For managing the frontend assets we used Webpack for the JavaScript and Grunt for the CSS. These tools compile ES6 and LESS code into plain JavaScript and CSS, respectively. Nowadays it's not very common for developers to code directly in JavaScript and CSS simply because they are too simple. LESS enables class inheritance, mixins, variables and conditional logic which makes it extremely more viable than pure CSS. ES6 on the other is a more functional approach for writing JavaScript, it makes it much more fun.

Vue.js

Grunt

As for our developer tools we are using Bitbucket for versioning, Asana for managing our tasks and Slack for communicating. Recently we have had a revamp of how we are managing features and we might switch Asana out and replace it with Trello or some other Kanban system. These tools are state-of-the-art, especially Slack which is atsounding for communicating. Personally I have a preference for GitHub because of the UI but Bitbucket does an OK job for doing code reviews.



Webpack

Bitbucket

Slack