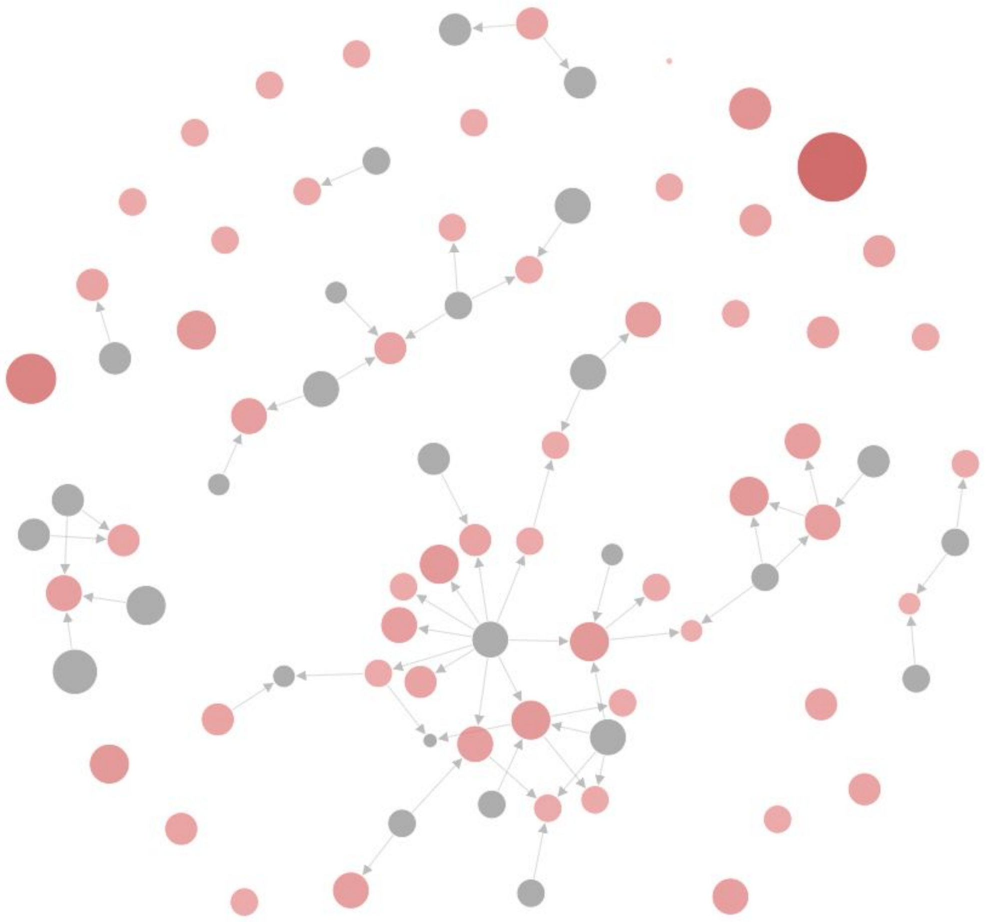
EPFL Interactive Coursebook

Go to app



Process Book

Data Visualization - EPFL 2020

Team Vizcachas

Michael Spierer

Michal Pleskowicz

Valentin Loftsson

Motivation

If you are a student at EPFL, you know that sometimes it can be hard to pick suitable courses for the upcoming semester. You might have browsed through the different study plans and course books and opened multiple tabs on your web browser in order to make sense of the highly interconnected system of courses. This is why we decided to create the EPFL Interactive Coursebook, an interactive tool and visualization of EPFL courses.

Project board and features

The project board and milestone can be viewed on GitHub. All features described in milestone 2 and many more have been implemented. The only exception is the data story, which we decided not to include as we felt it did not fit into the idea of our project. Since our project is a functional interactive tool for students, every user creates their own 'data story' depending on their interest. For insights about the data you are welcome to read the previous milestones and notebooks.

Data

This visualization project is based on data about EPFL courses and course registrations. An introduction to the dataset and data sources was provided in milestone 1. We first had to create our dataset by harvesting the data. Thereafter we processed and cleaned the data. Finally we had to store it and create an API.

Harvesting

The data was harvested using the Python library Beautiful Soup. We scraped 4,000 web pages to have a complete dataset. We faced a challenge in finding common patterns in study plan and coursebook pages and we were forced to deal carefully with exceptions. Pages were either in French or English so we had to take that into account. Some master's programs have specializations, so we gathered those too. Several iterations were required before we were satisfied with the format. We were faced with a different challenge when harvesting course registrations, in that IS-Academia's HTML and javascript was "old school" and harder to fathom.

Processing

For data processing we used the Python json module and Pandas. We faced multiple challenges in processing the data (see wrangling notebooks 1 and 2). We noticed that some programs listed in the EPFL study plans are not currently offered, such as Master's in Bioengineering, so they were removed. Also, a couple of master's specializations were not up to date and some study plans listed non-existing specializations. Many fields in the resulting data structure had gaps. We filled in the blanks using smart aggregation and inference techniques to minimize manual work. For instance, we detected inconsistencies using Pandas by grouping a dataframe with duplicate courses by the course id and filtering out consistent groups, leaving only a handful of courses which required manual overview.

One of the more time-consuming challenges in our project was cleaning up data for our dependency graph, that is the "Required courses" field in the official coursebooks. When it comes to that field, most of the professors don't provide the official course codes. Often, the course names would be written in french, or contain names of courses that don't exist anymore. In the good case where the course code was mentioned, it could appear in many forms (i.e. HUM-101(a) spelled as "HUM 101A", "hum-101" or any other combination of letter cases, dashes and brackets). At first, we parsed all the course codes that appeared in this field and matched them to proper existing courses. After that, we had to go over the data by hand and match the remaining ones. We carefully linked the courses mentioned in that field with the most appropriate existing courses and their respective course codes.

Permanent storage

We decided not to store the data on GitHub to separate code versioning from data versioning. Instead, we stored the data as a single json file (~6 MB) in an Amazon S3 bucket through the Bucketeer Heroku add-on. The file size exceeded the free limit so we used student credits to upgrade the plan.

Data fetching

The web app fetches the data using the AWS javascript SDK. Amazon S3 was new to us, so we faced a bit of a challenge in figuring out how to use the SDK, especially configuring the credentials and the service object. Finally, we managed to write a simple module for it.

Browser client storage

It would be annoying for the user if the app would have to load the data every time they visit or every time the data is queried. To avoid this the app stores the data as json strings in the browser's local storage. We chose this alternative over cookies for example, since local storage allows storing up to 10 MB of data and is designed to be used client-side. It is also convenient because the data isn't cleared until the user does it manually and the user doesn't have to wait for long after the first visit. The dataset is also small enough to fit. Moreover, it enabled us to create a synchronous API which simply fetches a string from the storage, parses the string into a JS object and computes the return value based on the query.

We quickly realized one caveat with this approach after allowing a few friends to test the app. If the data is updated, previous users won't receive the new version since the app detects the browser already has the data. That's why we implemented simple data versioning, by hardcoding a version string and checking if this string matches the one stored in the browser.

API

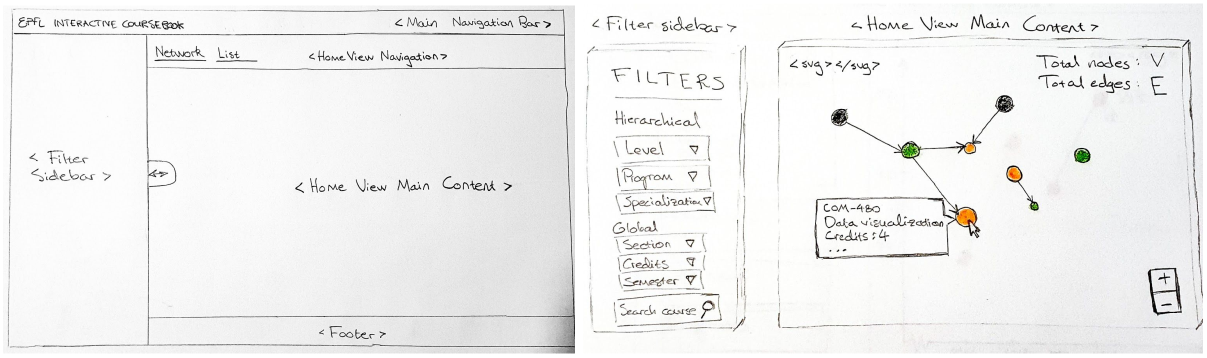
As mentioned above, we implemented an API to enable custom querying on the raw json data stored in the user's browser. We found it to be a necessary layer between the raw json and the web app. It is the basis for complex features, such as the filters and retrieving courses of master's programs which have specializations.

User interface

The web app is a responsive single-page application built with Vue.js, and works well on mobile and desktop. The UI is implemented with Vuetify. There are many aspects of the design and implementation we could discuss in this section but we will stick with key aspects for brevity.

First design vs. final design

In milestone 2, we provided sketches and ideas for the final design of the product. Below we can see sketches of the UI and graph visualization.



The final product reflects the original design in key aspects. We can name several small differences between the original design and the final product:

- There is no legend or a toolbar for the graph in the original design as in the final product.
- The original design has graph information displayed in the upper right corner. We decided it wasn't essential and rather incorporated node counts per color to the legend.
- The zoom buttons were not included in the final product to keep a minimalistic interface and it is more intuitive to zoom with the mouse/touchpad than to click on a button.
- The sidebar with the filters is a bit different in that the final product includes a button to open the sidebar and not only a chevron hover button (displayed when you move the mouse to the left edge of the screen).

Color theme

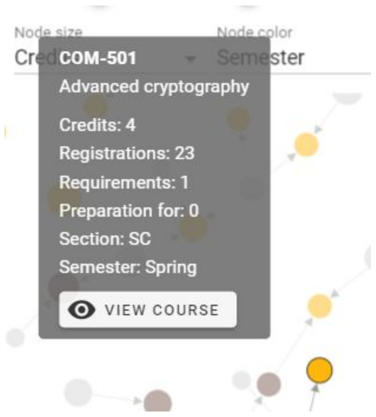
We picked *red* as the app primary color because it is the EPFL color. Otherwise we use Vuetify default theme for text and other components. It keeps the website clean and minimalistic.

Network view

The network (graph) view is the most important view since it shows the main visualization component of the project. We faced many challenges with the UI and we'll describe a few here:

- The tooltip which displays course info when hovering over a node was overflowing the container. We solved the issue by detecting the position of the cursor relative to the viewport on both axes and adjusting the position of the tooltip accordingly.

- We first thought about drawing the graph legend in d3 on an SVG but found it cumbersome to create a dynamic flexible legend all in a single SVG. Instead we resolved to write the legend in the Vue component itself, containing it in a Vuetify expansion panel and exploiting grid layout. We faked SVG <circle> elements using a plain <div> element and simple CSS tricks. The circles are animated in the exact same way as the circles in the graph.
- Another challenge is that we wanted to allow the user to go from the graph view to the course view easily. At first we thought of a shortcut like ctrl+click but it's not intuitive and doesn't work on mobile. We wanted to create an intuitive experience for both desktop and mobile users but we faced challenges with facilitating both features for touch devices, since hovering doesn't really exist for touch interfaces. On desktop, the user is able to hover over a node to view course details and click on a node to view the course page. We wanted touch device users to be able to see the tooltip and *also* view a course page. To solve this, we implemented touch interface detection to provide a friendly interface for touch device users. So, when clicking on a node, the user sees the same tooltip as desktop users, except it also includes a "View course" button.



List view

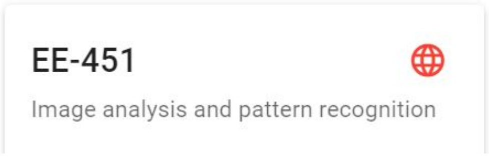
An alternative to the network is the list view which displays a paged list of courses, lexically sorted by the course name. It is there for convenience.

Course view

This is the view that pops up when clicking on a course in the network and list views.

In this view, we can see key information about the course in question as well as an interactive chart of course registrants over the last 5 years. At first one would think the chart only shows the total number of registrants for each year, but when you move the mouse over a bar the break-up by each program becomes visible. The chart is discussed in more detail in the "Visualization" section.

To the right of the course id we inserted a clickable globe icon which navigates the user to the source coursebook. One can also click the name of a professor to navigate to people.epfl.ch for detailed information. In addition, the user can navigate to "neighboring" courses.



When the course is selected, it is added as a tab to a slidable tabbed menu which can be opened or closed at wish. This comes in handy if the user wants to view many courses at once or go back to one visited previously. A difficult challenge we faced was to synchronize this menu with the main tabs (Network/List). Another challenge was implementing the open/close functionality.

Filters sidebar

The filters serve a key role for making the app interactive. We quickly realized that the filters didn't all relate in the same way with the data, so we had to organize them into two categories: Hierarchical and Global:

- Hierarchical parameters are level, program and specialization. These are not unique for each course. For instance, a course may be offered at different levels. Changing a hierarchical filter triggers filtering of dropdown options for lower hierarchical filters and all global filters.
- Global parameters are section, credits, and semester. These are unique for each course.

We also implemented a course "cherry-picker" via an autocomplete search box with multiple selections enabled. An important design decision was that the cherry picker takes priority over other filters so when you change any of the other filters, it is only the autocomplete options for the cherry picker that get updated.

Visualizations

Network-Graph

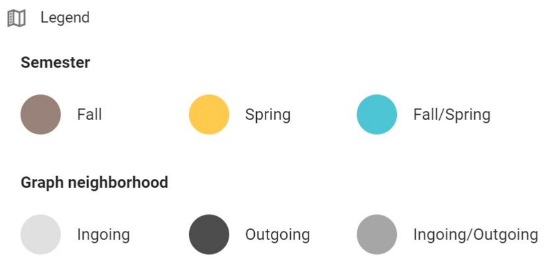
The graph allows the use of novel ways to browse the data. For instance, the nodes can be colored by certain course properties: number of credits, the EPFL section or the semester. We carefully selected the colors for each parameter in order to always render the graph with a pleasing and intuitive color palette. The user can drag and zoom, move the nodes and click on them to get to the more precise view of the course. Hovering over a course displays a tooltip with a short summary of the course and highlights the neighboring nodes.

Displaying the subgraph neighborhood and distinguishing it from the main selection was a conscious design decision we took so that the user sees all relationships of the courses which match the filters. We also implemented switches to toggle ingoing/outgoing neighborhood nodes and edges. All this required a bunch of extra work since we had to distinguish neighbor nodes and links from the rest.

Size

One of our questions was to know when choosing the size of the nodes according to a parameter if we should apply the sizing on nodes that match the query (the subgraph without neighbors) or all of the nodes including neighbors. It is certain that if we choose the size in relation to the number of credits, the question does not arise. However, if you choose the size in relation to the number of required courses, you have to make a choice: either apply the sizing to the subgraph or all the nodes including neighbors. An example would be that if a course such as Analysis II, being a prerequisite for many other courses, would be part of the neighborhood and was left out, it might appear less important than another when in fact it would be the opposite. Therefore, we chose to apply the resizing to all nodes.

Color and legends


Legend

Semester

● Fall ● Spring ● Fall/Spring

Graph neighborhood

● Ingoing ● Outgoing ● Ingoing/Outgoing

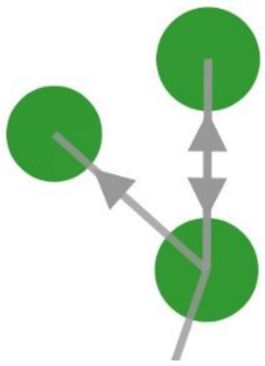
For several scenarios, we had to deal with a situation where we needed a lot of different colors, which made the graph difficult to read. We therefore brainstormed to find solutions to make the graph easier to read or more comprehensible. That's how we came up with the idea of making a dynamic legend.

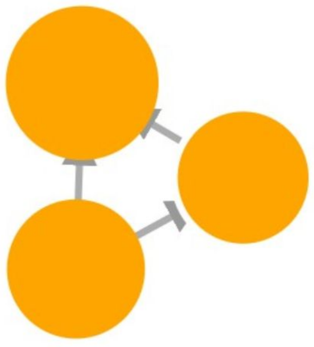
Another challenge we had to face was the coloring of ingoing and outgoing neighbor nodes. Just as a reminder, a node is ingoing or outgoing if it enters or exits the subgraph of nodes which match the filters. A subgraph neighbor node can be ingoing, outgoing or both. Our first iteration was to color all these nodes in grey but we wanted to differentiate them. For example if we are only looking for Master courses, most Bachelor prerequisite courses are ingoing nodes and will appear in grey. But courses leading up to the PhD will mostly be outgoings nodes and will also be in gray. So we chose to make the ingoing nodes light grey, outgoing nodes dark grey and nodes belonging to both just grey.

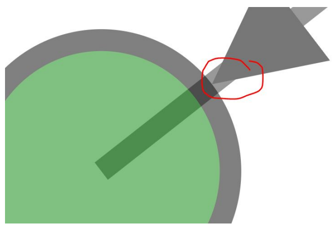
We had to face another decision about the uniform node color parameter. The latest version of the coloring scheme worked this way: when a filter is applied, the unchanged/updated nodes are yellow and the new/entering ones are green. However, this mechanism is not the most intuitive for the user and we faced technical challenges in implementing the legend for this special case since it isn't based on any course properties like the other parameters. We therefore had to decide whether we wanted to keep the feature and, if so, how to improve it. After much discussion, we decided to remove this feature even though it worked for this milestone in part for the sake of readability and clarity of the graph.

Computing positions of directed edges

By default, d3 draws the links between nodes as lines with starting and ending positions at the center of the nodes (see 1st iteration). First we found that we were drawing the links on top of the nodes, and fixed that (see 2nd iteration). Since we wanted the nodes to be transparent (see 3rd iteration), we had to recall some basic trigonometry to compute custom starting and ending positions of the links, depending on the node radius and stroke width. When that was done we faced issues when nodes would collide so we implemented collision detection to fix that.

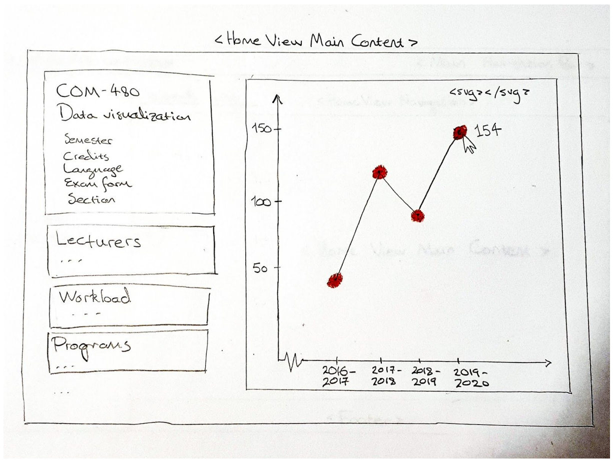

1st iteration

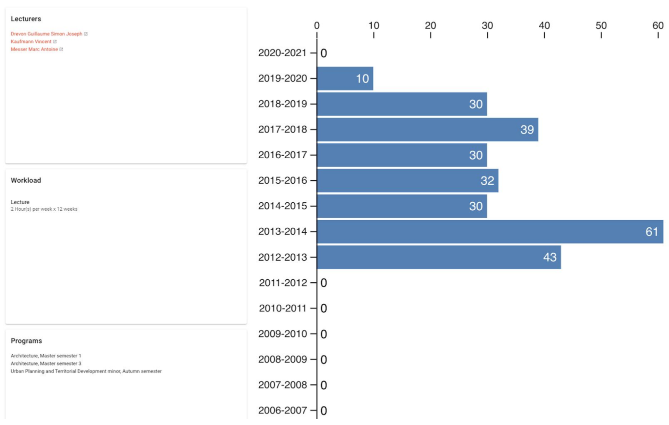

2nd iteration


3rd iteration

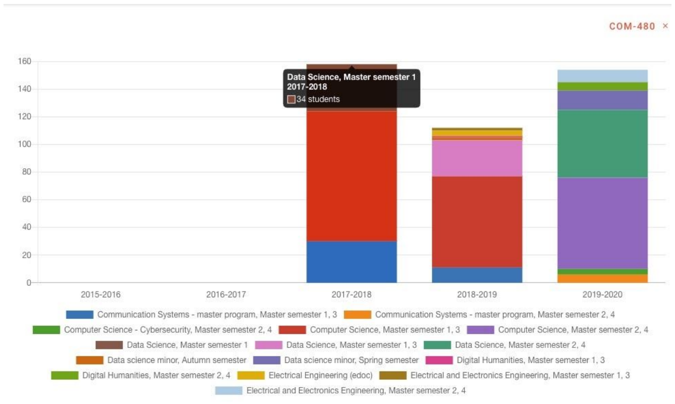
Course registrations chart

Concerning the original design vs. final design of the registrations chart, our final product is different in that we decided to implement a stacked bar chart instead of line chart. The only similarities are the axes and position on the view (except on mobile it appears below the cards). Also, we used chart.js instead of drawing it in an SVG with d3.

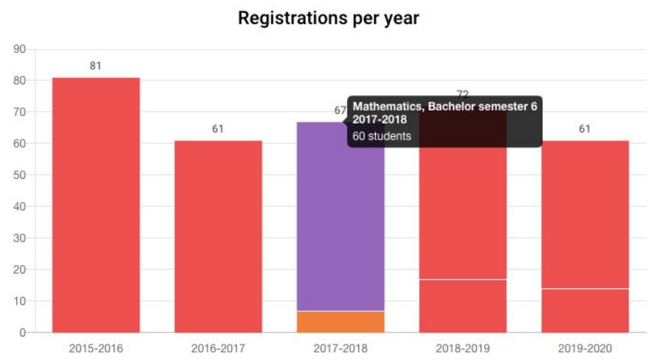




The registrations chart went through many iterations. At first it was a simple bar chart with blue bars, which soon proved to be unsatisfactory because it wasn't reactive like the rest of the website and it didn't fit its theme. It contained data for too many years, and thus many empty columns as a lot of courses were only created recently. Also, it didn't scale well, all of which you can see in the figure above.



The second, cleaner version consisted of vertical bars which showed the total number of students in a given year on hover, in a tooltip. We limited the number of displayed data points to the most recent five academic years. At that point we realized that we also possessed registrations data broken down by the program, and we decided to incorporate that into our chart. The stacked bar chart turned out to be too cluttered with information, both on the bars and in the legend below.



We solved this problem by using separate levels of detail - without interaction the chart displays red bars (with the sum of all registrations displayed above every bar), which reveal stacked registrations from different programs when hovered on with the mouse. There are visible lines between the stacks to show the user the structure. The final version can be seen in the figure on the left.

Conclusion

Aside from the fact that we were all mostly new to Vue and D3, our challenges were more related to the design of our visualizations and the user experience. Concerning the future, if we decide to continue the project, we will have to work on the speed of our site since it becomes quite slow when you clear all the filters to display all the courses in the graph.

Task allocation

Michal Pleskowicz

- Registration chart on the course detail page
- Preprocessing the data to create the json files with course and program information
- Manually processed the ‘requirements’ field in the course info and found corresponding course codes for the required course if they existed, as the data was very unstructured
- Color maps by semester/credits
- Small stylistical changes in the color palette, naming, fonts etc
- Default values for the initial page display in order to speed up the loading
- Basic legend in the main graph

Michael Spierer

- Zooming and moving around the visualization
- Color maps by credit in order to have a sequential mapping
- Preprocessing the data, dana analysis
- Manually processed the ‘requirements’ field in the course info and found corresponding course codes for the required course if they existed, as the data was very unstructured
- Drawing arrows between nodes
- Worked on the UI and on the filters
- Centering the graph (with button or when applying filters)
- Dynamic Scaling

Valentin Loftsson

- Data harvesting and processing
- AWS S3 storage and browser client storage
- API and filters, course cherry-picker
- Vue app configuration and prototype, web app deployment and configuration
- Worked a lot on the UI with Vuetify
- List view with paging functionality, course view (except registrations chart)
- Graph view, tooltip, computed link positions, toolbar with switches and dropdowns