CHAPTER 6

Cataloging: From a Dusty Backroom to the World Wide Web

Owing to the mere fact of their accessibility, it has been established that digitizing little used, obscure materials can increase their usage. In the case of a collection of American literature from the nineteenth century at Cornell, the increase of use was so dramatic that it was concluded that "In hard copy the material may have seemed obscure; when digitized it becomes a core resource."[1] Manuscript collections only benefit more from this effect, as their items are unique and only to be handled under strict regulations. Then again, mass digitization will only make a real impact if it goes along with mass metadata, a digital catalog of some sort, with information like title, author, dating, and provenance.[2] In this sense, creating digital catalogs should be a priority. For the most part, this is the domain of librarians and curators, who are experts at this just as much as they know far better how to do the digitization than we, scholars, do. But just as we digitize on a small scale ourselves, when working on a research project, we will often find ourselves in a situation where it would be beneficial to create a small catalog for our private research needs. It should be noted that 'catalog' here can refer to anything as long as it is a collection of items described under certain terms. So, just as we can catalog books, we can also catalog illuminations, glyphs, or abbreviations. In this chapter, I present a case study to explore how we can do field work digitally and transfer the labor of that fieldwork into catalog data and then how, from that catalog data, get to a digital catalog accessible online, much like the online catalog developed by a team at the University of Trier of the collection of the German monastery of St. Matthias, but then developed by ourselves and without a budget. Much of

1 Hirtle, P.B. "The Impact of Digitization on Special Collections in Libraries." pp. 42–52 in *Libraries & Culture* 37, no. 1 (2002), p. 43.

2 Nichols, S.G., "Materialities of the Manuscript: Codex and Court Culture in Fourteenth-Century Paris," pp. 26–58 in *Digital Philology: A Journal of Medieval Cultures*, vol. 4, no. 1 (2015), p. 27; Ornato, E. "La Numérisation Du Patrimoine Livresque Médiéval : Avancée Décisive Ou Miroir Aux Alouettes ?" pp. 85–115 in *Kodikologie Und Paläographie Im Digitalen Zeitalter 2*, edited by F. Fischer, Chr. Fritze, and G. Vogeler. Norderstedt: BoD, 2010. p. 96; Riedel, D. "How Digitization Has Changed the Cataloging of Islamic Books." *Research Blog Islamic Books*, August 14, 2012. Dahlström, M. "Critical Editing and Critical Digitisation." pp. 79–98 in *Text Comparison and Digital Creativity*, edited by W. van Peursen, E.D. Thoutenhoofd, and A. van der Weel. Leiden: Brill, 2010, p. 90ff.

this work rests on web development technology, which is an especially potent part of computer technology for many parts of our work in the humanities.[3] Importantly, web development is very easy to learn since there are a plethora of resources freely available. At the same time, one significant downside to web development is that it is undergoing an extraordinary evolution, with new major innovations pushed out almost every year as new standard ways of working. Therefore, in this chapter, we shall focus on the fundamentals in the understanding that with this knowledge, you will be able to go out and adopt any new technology that might benefit you.

1 Field Research Workflow: From a Dusty Backroom to My Computer

In this section, I shall describe the workflow I settled on when I worked on cataloging a small collection of books, articles, and manuscripts. These artifacts are kept at Sankt Florian, a monastery near Linz, Austria. Sankt Florian, in its current form, was built in the 17th century in a baroque style. It is basically a monastery and palace in one. One part of it was for the Augustinian Canon Regulars to live and pray, while the other for the Habsburg Monarchy to stay for the night and conduct business. St. Florian is a center of music with a world-famous organ, but also boasts a very large collection of books, part of which are kept in a dramatic baroque gallery. On their website, it says that:

> In 1930, the Monastery library bought the literary remains of Rudolf Geyer (1861–1929), a Viennese orientalist. Still 20 years later, this collection was considered the most comprehensive one in Arabic literature between Berlin and Rome. Meanwhile, about a third of Geyer's books is indexed.

Naturally, I was intrigued. I wondered exactly how big a 'comprehensive' collection it would be, and I figured that given the life years of Rudolf Geyer, there must be rare or otherwise valuable books from the 19th century in this collection. After inquiring, the 'index' of one-third of the collection turned out to be a handwritten list of title, author, place, and year, only to be consulted in the library itself. Further details about the collection could not be given. An on-site visit was unavoidable, which I did in the summer of 2016. Arriving late on Monday and leaving early on Friday, I only had three full days to conduct

3 Susanne Kurz rightfully made it one of the pillars in her practical introduction to Digital humanities, see Kurz, S. *Digital Humanities: Grundlagen Und Technologien Für Die Praxis*. Wiesbaden: Springer Vieweg, 2015.

a preliminary investigation. As a testament to the power of a digital work environment, although the catalog was only completed two years later, it was all based on my fieldwork of just those three days.

Upon arrival, I came to know that the Geyer collection was not in the beautiful gallery part of the library, but in a dusty back room. It covered perhaps about twelve bookcases, each with seven shelves, many of them containing double rows of books. Inspecting the hand list that had been drawn up, I noticed that the previous cataloger had looked at certain sections that contained only European language resources, of which most of them were articles, individually bound and shelved. Browsing for half an hour made it clear that the majority were books. There were also a lot of articles as offprints, and a dozen or two manuscripts. In short, anything from a book review to a multi-volume primary source could be an item, with each one having a seal, like a post stamp with an index number. Interestingly, inside virtually all items, Geyer placed an Ex Libris sticker, which also shows a number, different from the one on the seal; thus, apparently, there are two numbering systems. In cataloging the collection, I only considered the index number on the seal, since this is on the outside of the item and is, therefore, easier to inspect while browsing the shelves. Next to this collection were boxes with notes, drafts, letters, and other things belonging to Rudolf Geyer. I did not investigate them in any detail, focusing on the proper collection.

Sitting down for each item of the collection and noting the catalog details would have been too time consuming. Even if I had much more time at the monastery, it seemed like an ineffective workflow. Given the size of the collection (about 1500 items) and the time left, I decided I could make a photographic index of all title pages, or at least of those items not mentioned in the hand list. I used my phone for this, an iPhone 6, which facilitated my process. Even while holding the phone in my hand, I could use both hands to pick up items and flip them open. Keeping them open with one hand, I snapped a picture with the other. For lower shelves, I used a trolley to load part of a shelf onto it, returning the item after taking a picture. This was also necessary to reach the second back row. On the top shelves, which had a single row, I simply stood on a ladder, using its top surface as a small table to keep the items straight while photographing them.

I did not use the stock camera application but Evernote.[4] Evernote is a simple note-taking application. It is essentially a database for notes of all kinds, typed text, drawn handwriting, audio, images, and PDFs, with a user-friendly

4 There are alternatives. For example, Tropy is free software specifically developed to take in thousands of photos a researcher takes at an archive and provide the user with a friendly way

interface built around it that is so polished and easy to use you rarely think of it as a database, but simply as a note-taking app. With an eye towards possibly reusing your current labors, I would recommend using such an application. It keeps all your notes together and stores them in a way that will be accessible and exportable for the long-term foreseeable future. Notes are generally inserted in different notebooks, but with one search, one can find notes across all notebooks. It also ensures an offsite backup, as everything is stored on Evernote's servers too. In the case of cataloging Geyer's collection, it allowed me to store photos of each item in separate notes, so that later on I would never have to doubt whether a photo belongs to one or the other book. Also, if I want to export the photos to make them ready for another application, a rudimentary division is already baked in. If I ever wanted to bring all the photos together, it would be possible too. Originally, I wanted to give each note the title of the item number as it is written on the post stamp-like seal. After only a few items, I realized that typing them out was taking up too much time. Instead, I opted to simply make another photo, this time of the seal. In certain cases, for example when there were multiple title pages in different languages, or when there were multiple volumes of one title, or when parts of the catalog information was not on the title page but on the last page, I snapped additional pictures. In total, I took probably around 2,500–3,000 photos. If we assume three full days of eight hours work, that would come down to half a minute per photo, or about a minute per item. According to my notes in Evernote and comparing their time of creation, I did indeed spend a minute—sometimes even less—per item.

However, snapping so many photos in Evernote came with a cost. To upload everything to Evernote's servers took over a week. Every photo taken in Evernote is also saved in your Photos app. Since all applications on an iPhone are sandboxed, the photos are physically stored twice on my phone. The upside of having the photos also in the Photos app is that I was able to quickly put them on my laptop. I only had to connect my phone, and the Photos app on my Macbook appeared, allowing me to select and download the images. The downside is that, after, having the photos twice on my phone seemed redundant, but trying to delete the photos from my Photos app was surprisingly difficult. Apparently, deleting two to three-thousand photos at once from Photos is an incomprehensible task for the phone. I tried it several times but failed. I also tried to do it for each day, basically in batches of about eight-hundred, but that too resulted in nothing. The iPhone simply remained unresponsive and

of making sense of all the photos back at home. A more fully-fledged alternative to Evernote is Onenote.

did not delete anything. I ended up deleting them in a hundred or so batches of twenty photos. It is entirely possible that such issues are resolved in the future, but undoubtedly, you will encounter other but similarly odd behavior. It seems, then, that we are stretching the capabilities of consumer electronics and stock apps. At the same time, it is pleasant to notice we can actually get by with these simple tools, and we do not need to acquire more professional hardware or software to do our job.

By then, I had the title pages of the Geyer collection right in my pocket, stored in Evernote. Considering the number of notes I ended up with, together with the written hand list, I estimated that the total number of items amounted to no more than 1,500. The next step was to extract the different elements of the title page (title, author, publisher, etc.) into plain text, collected in such a way as to be able to be constructed into a catalog. I considered if I could make the jump from images straight to professional, library-quality cataloging. This, however, was unnecessary. Libraries use database systems that require vast amounts of entries, constant updating, and write-access for multiple users. None of that applied to the Geyer collection. All I needed to end up with was a machine-readable list of all the items with their details so that we can then reuse the list in different ways; either to create a printed catalog or an interactive, online catalog or to load it into a bigger catalog. This list would contain only a limited number of entries, at least from a computer processing point of view. Furthermore, the list would require little to no update afterward so the ability to edit entries did not need to be fast and user-friendly. Lastly, I knew I was the only one who was going to put in hours of work into this, so there was no need to allow multiple users.

To reduce the hours of work needed and make the actual work as painless as possible, I considered making a custom application in FileMaker. This is a software with which you can create simple relational databases. Using its drag-and-drop elements, you can create forms to either display or enter records. A relational database is best visualized as several tables held together by relations. In each table, a row indicates a record, representing a unique object that is described by the values written in that row in several columns. For example, a table *persons* can have columns such as *first name, last name,* and *age*. Each row then represents a person, described by their name and age. This person is unique: it is only defined once in the table. It should be noted that a table is only a visualization. When we look at tables, there seems to be a specific order, with a top and a bottom for the rows and a left and a right for the columns. For databases, this kind of order is not actually there: all records are stored as though they are marbles in a bag in which you put your hand to reach for them blindly.

In this example, if you also wish to include a column *books*, to describe all the books the person wrote, you will encounter a problem. For some people, there will be no books; for some, one title; for others, multiple. We would ideally have a dynamic number of columns to fill the number of books per person. Let us assume for argument's sake that each book has only one author. Then, a better way to write this down is to open a new table called *books*, listing each unique book as a row with columns for *title* and *author*. In the author field, we only need to put a referral to the correct row in the *persons* table. Such a referral is called a key, a foreign key to be exact since the key belongs to an entry that is foreign to the *books* table. For each table, an extra column may be created to store a unique ID. This is also called a key, but now a primary key.

Storing information like this, in a relational database, has proven to be extremely useful in the digital world. Information can quickly be obtained, and very few pieces of information, if any, needs to be stored twice. This is not only convenient in terms of file size, but it also means that if information needs to be updated, you only need to change it in one place and, based on that edited record, it is updated everywhere else. FileMaker allows you to point and click your way through setting up such a database, and by making attractive forms, filling that database with information can be both quick and somewhat fun. A big advantage of FileMaker is that you can create forms that work on iPhones and iPads. This allows for entering information on these handheld, touch screen devices, which then sync back to the main database on a computer.

What I had in mind doing was to load all the Evernote photos automatically in FileMaker, and then create different forms that each would only add a small piece of information. To start with, I wanted FileMaker to present me with a photo of a seal, and a small text box to type the index number visible on the seal. This would have been a task that I could do on my phone while waiting or traveling, and by accumulating all these small moments, I would have entered all the index numbers without truly having lost time over it. With the title pages, I wanted to do something similar, but with a twist. The first step was to get a photo of a title page on my screen, and then I would have to press and drag to create a rectangle around the title. After releasing, the photo would stay visible for another two seconds to allow the user to cancel; otherwise, the area of the rectangle would be stored as the area where the title is, and a new photo would instantly appear. Thus, it would be a matter of endless drawing of rectangles around titles. After that is done, a second step would be activated, in which only that part of the images as defined by the rectangles would appear on the screen, along with a text box, to type out the title. A similar strategy would be applied for each element on the title page. By chopping up the work into these small, menial tasks, I intended to catalog the collection

in small, spare moments. The only problem was that to build that functionality in FileMaker would take a serious amount of time. Although I did have experience with FileMaker, I did not find the time to sit down and make it. I still think it is a good way to go about processing fieldwork, but it perhaps becomes more sensible when the corpus is bigger than a mere 1,500 entries and when there is a more immediate reason to get it done.

Instead, I fell back on a piece of software I had already been using for other parts of my research: Zotero. Zotero is a citation (or reference) manager, similar to EndNote and Mendeley, which syncs your references to its server. Zotero already provides that user-friendly interface I needed to type out the different details of the title pages. It has a function to export all the entries to a machine-readable format such as XML or JSON (more on this later). It does not work on phones or tablets since the interface of Zotero is designed to be used with a keyboard. After all, cataloging is generally a keyboard-reliant activity. I figured I could diminish that in FileMaker by creating custom inputs that would only require one or a few taps on the screen. Since Zotero or its third-party apps are not customizable, I had to change my workflow around the philosophy that cataloging should be done with a keyboard and in one go, collecting all metadata of one item before moving on to the next.

As a first step, I went through all the photos in Evernote where I had stored them, hand-typing the index number in the title of each note. This was a fairly painless job, since activating a note would instantly display the photo of the seal. This was more luck than wisdom, for if the photos of the seals came after the photos of the title pages, I would have had to scroll down on each note to reveal the index number. I figured it was worth the trouble of typing the index number in Evernote so that I could order the notes more easily and keep track of all of them better as I moved through the process of cataloging.

In the fall of 2017, I settled into a rhythm of working a little each night. Over two months, after some fifty hours,[5] I had pretty much completed typing out the catalog details. For some quick math, these fifty hours can fit neatly into two months if we assume an hour of work each day, spending about 2 minutes per item. Both estimates seem like a reasonable time. Little can be said about the use of Zotero since it is self-explanatory. The only odd thing is that the field to enter the location in the archive or the field to enter the call number is very far down. To reach them, I would have had to hit the Tab key a lot of times, which is both prone to mistakes and time consuming. Therefore, I ended up

5 I know because I had the tv-show *The Office* on in the background and by the end of my manual data entry I had reached Season 8.

using the Language field to enter the index number, which is usually only two tabs down from the field for Year.

Having gone through my notes once, it was time to clean up the data I had entered. I ordered the entries on the Title field, which grouped together all entries with an Arabic title. I had typed these out in Arabic script, and now I added a transliteration in the Short title field. After ordering on the Place field, I consolidated place names, for example, changing all occurrences of 'Wien' to 'Vienna'. Ordering on the Language field, I could check the index numbers, making a note of those numbers that were missing. I identified 51 (about 3% of the collection), including those of which I had no photos, and those of which my photos were too blurry to make use of. I asked somebody with access to the library to investigate these numbers and send me photos of whatever could be found. As it turned out, after making a public catalog out of my data, I once again discovered some aspects that required cleaning up. For example, in Zotero, I had initially entered in the Year field whatever was on the title page, meaning that if there was a year from the Islamic calendar, I typed out that ending with an *h*, for *hijrī*. Only later did I notice that this will not allow ordering by year. So, instead, I converted all *hijrī* years to *mīlādī* years in the Year field and added to the 'abstract' field the original *hijrī* year. Having a graphically more attractive presentation of the catalog also enabled me to spot typographical errors and other mistakes more easily. After I had exhausted Zotero's functionality, I decided to export all the entries in the file format 'CSL JSON.' This stands for Citation Style Language JavaScript Object Notation.

To understand either part, CSL or JSON, let us first introduce a third term, XML, which stands for Extensible Markup Language. An XML-file is like a plain text ".txt" file, which you can open with any text editor. However, you are not supposed to simply type whatever you want, but you need to enter your information in a specific way for it to be a valid XML-file. This is because while an XML-file is easy to read for us human beings, by nature of the regular patterns of the specific way XML-files are supposed to be written, computers can interpret them easily too. This specific way is rather simple: every piece of information should be surrounded by tags. For example:

```
<example>some information</example>
```

The word 'example' is called the tag, and it is written between the angular brackets so that a computer can know this is the tag. As soon as a computer sees an < and an >, it will remember the word in between and look for the same word but this time in between </ and >. The slash, then, indicates a closing tag. Anything in between the opening and closing tag is the information related

to the tag. Tags can exist within tags. For example, a description of a book can look like the following:

```
<book>
  <title>Among Digitized Manuscripts</title>
    <author>
      <firstName>L.W. Cornelis</firstName>
      <lastName>
        <surNameProper>Lit</surNameProper>
        <surNamePrefix>van</surNamePrefix>
      </lastName>
    </author>
  <publisher>Brill</publisher>
  <place>Leiden</place>
  <year>2020</year>
</book>
```

This format is probably quite easy to read for a person, although it is not a very attractive format. We can write code to have a computer go through it and place all the different elements in the right order using the right styling. For example, it can be printed to the screen as "Lit, L.W.C. van, *Among Digitized Manuscripts*, Leiden: Brill (2020)." The order, the addition of spaces, commas, and other punctuation marks, and the italics, are all done automatically—a great relief if you need to do this for hundreds of records or if you wish to change the styling later on.

XML does not impose any restrictions on what the tags should be. As long as all the tags close, it is valid. Whether the first tag reads *book* or *publication*, whether the nested tag in *lastName* reads *surNameProper* or *lastNameProper*, that is up to us. This makes XML usable for basically any situation in which some ordered data needs to be stored for computer manipulation. The drawback is, of course, that if I use *book* and you use *publication*, then a computer will not recognize them as the same. So, if somebody writes code that instructs the computer to take all the elements called *book*, it will not do anything if you have prepared a file where all these elements are tagged as *publication*. A standard that is accepted by everyone is needed, with rules that we all abide by, so that we can rely on the regularity of the rules to automatically extract and manipulate information from the XML-file.

Citation Style Language is such a standard. It is devised by companies behind three applications to manage references, Zotero, Papers, and Mendeley. CSL provides a way to combine all the different fields we used in Zotero into a

list that can be read by any software that also uses CSL. For example, it will be very easy to export from Zotero and import into Mendeley.

What Zotero produces is, actually, not an XML-file, but a JSON-file. Opening a JSON-file in a text editor will demonstrate its similarity to XML. It has tags, that can be nested, containing information. The difference is that JSON does not need a closing tag and uses a shorter punctuation, making a JSON-file much smaller. Let us consider the above example, this time in JSON format:

```
{
  "title": "Among Digitized Manuscripts",
  "author":
    {
      "firstName": "L.W. Cornelis",
      "lastName":
      {
        "surNameProper": "Lit",
        "surNamePrefix": "van"
      }
    },
  "publisher": "Brill",
  "place": "Leiden",
  "year": "2020"
}
```

From a human point of view, it reads much more like a table, making it more readable. From a computer point of view, it is directly usable by one very popular programming language, JavaScript. Of course, other programming languages also know how to deal with it. For example, a dictionary in Python (see Chapter Seven) is very similar, and it takes little effort to load a JSON-file in Python as a dictionary. Since the name JSON reveals its affinity with JavaScript, by loading the above in a variable, say *book*, we have created an object *book*, and all the fields are its attributes. This means that if we ask JavaScript to print *book.publisher*, we get "Brill." The question, then, of what to do with the catalog details of the Geyer collection in machine-readable plain text presents itself almost automatically: if we have made the catalog into a JavaScript Object, maybe we should make use of the web development technology to turn our data into a product that is easy to browse and search through; that is both pleasant to look at and pleasant to use. A printed publication did not seem the ideal solution from the start. Considering its small size and relative obscurity, it would hardly be commercially publishable, and printing it privately would

mean the distribution would be poor and it would be costly, while the entire project had a budget of zero Euros. Delivering the catalog digitally, preferably online, was the best idea, and web development technology seemed the right tool for it.

2 Web Development: From My Computer to the World Wide Web

For students and scholars of the humanities, since more and more of our fieldwork will take place in this sphere, proficiency in web development is a desirable skill to have. Whether we want to use manuscripts or a catalog that a library makes available online, or whether we want to use Google Books, Facebook, Twitter, or Wikipedia to scrape information in order to map out discussions that take place on the internet, since these resources are built on web technologies, we also have to investigate them using web technologies. The digital world, after all, is for a rather large part built on web development technology.

The good news is that of all popular technologies, web development is among the easiest to learn, simply by the mere fact of the vast amount of teaching resources available. You can get very far, in fact, without paying a penny. However, the bad news is that web technologies are, at the time of writing, rapidly developing and changing, indicating that learning should take a two-pronged strategy. There are the basics that one simply has to know and will likely remain useful for many years to come, and there are the actual, fully-developed technologies that should be seen as electives and should be learned on a just-in-time basis.

For the technology of the catalog, I first considered some off-the-shelf products like DataTables and Omeka, which only require you to input raw data. I concluded that they had added unneeded features but lacked certain aspects that were an absolute requirement for our case. Rather than trying to hack them into the shape I wanted them to be, I decided to build it up from the ground, so that it would be exactly custom-fitted for our situation.

Because others have explained web technology much better and more detailed than I can, and because a lot of it undergoes rapid changes, I shall not go over the code in detail. Instead, I will give a more conceptual overview of how and why I put together the code that I used.

First of all: what does web development mean? This term encompasses all the technology required to develop things that are transmitted and received over a network. Usually, this network is the world wide web, and these things are websites. As the complexity of a website grows and offers more functionality to

a user, we can better speak of a web application. For example, our catalog has all the functions of a catalog but is wrapped as a website, and users will have to reach it through a web browser.

The first basic division of web development is technologies that deal with the frontend and those that cover the backend. Frontend means what the user actually sees and can do in the browser, and backend is everything that goes on behind the scenes. Two similar (but not always the same) terms are client-side and server-side. Client-side refers to code that is downloaded and run on the computer of a user, while server-side code is run on a central server (the host of the website) and only gives the conclusion for the user to download and run. Server-side coding is largely synonymous with working with a database. A database is only necessary when the data of the web app is very large; it draws a lot of users, and/or its data requires very frequent updates or additions. Our catalog fits none of those descriptions. As a JSON-file, our catalog can be read by JavaScript and processed on the client-side.

What we want to develop, then, is a website that loads the JSON-file with all the catalog entries and displays them. We further want a simple search and a simple sort function, and we like the interface to be bilingual. It helps to sketch the layout of the website to understand what is needed.

It is useful to divide this type of development into four parts: one part of our code will govern the structure of our website, another will provide the content that goes into that structure, another piece of code will ensure how that content will be styled, and a final part will provide interactivity, not only between the user and the interface but also within the website itself. This division makes for flexible working. The structural level can ensure that there will be a Heading 1 title here or there, while the styling level can ensure that all those Heading 1's get, for example, a much bigger font size than regular text. On the level of content, we can define an English or German sentence for that heading while the interactivity can either program a button for the user to change the language or leave that functionality out and decide automatically which language to display.

The foundational programming languages for each of these levels are HTML for structure, JSON (or XML) for content, CSS for styling, and JavaScript for interactivity. They are each stored in a separate file, ending in either .html, .json, .css, or .js. We can have multiple files of each kind if a further distinction on each of these levels is required. For example, all the text for the interface and all the text for the catalog entries can be stored in two separate .json-files, for more clarity and flexibility. For example, if somebody asks for the catalog in JSON format, only one file needs to be given, about which nothing needs to

be done, instead of having to open the one file containing both interface texts and catalog with a text editor, cutting out all the parts related to the interface, saving it as a separate file, and only then sending it.

Because of the flexible structure with which we build this catalog, you will notice quickly that we are not so much developing a catalog for Rudolf Geyer, but a generic catalog into which you can pour any JSON that Zotero creates, with only a few adjustments that need to be made. The result would be a functioning, minimalist catalog appearing on the screen. I shall go over each part separately but since the most is happening in the JavaScript that part will take up the most space.

3 Structure: HTML

HyperText Markup Language was originally conceived to fulfill all the four roles that I just separated. It is the foundational language with which to create webpages that a browser, the application by which you surf the internet, can read and display correctly. For example, if some text is placed within b-tags, you would not see in the web browser some text, but you would see **some text**: the browser reads the b-tags and knows that that is an instruction to display the text in between in bold. Your web browser, then, knows how to read and display an HTML-file, but would not know what to do with any of the other files. So, in this sense, the HTML-file is the gateway to the rest of your online product (whether it be a catalog or something else). In fact, currently, the only thing we need is an *index.html*, which is traditionally the name of a website's landing page.

If you look over the code of the HTML-file, you will notice that it basically serves two functions. First, the *head*-tag is a shell that opens all the other JSON, CSS, and JS files. Second, within the *body*-tag, the HTML-file dictates a structure of where the different elements of our catalog should go.

First, a header is defined, which is divided into three parts. The header, as a whole, stands out for its different background color. On the top-left of the page, we want a logo of Stift Florian that links to the main website of the monastery. In the middle, we want a title that should look big and prominent, so I made it a Heading 1 title with the *h1*-tags. On the right, we want to have a button to change the language and a button to go to my own website (as a small nod to me being the creator of the catalog). The button to change the language should be a flag representing the language into which the user can change it. If the interface is currently in German (which it will be upon first loading the page, as

```
<!-- Copyright L.W. Cornelis van Lit, Please see https://
github.com/LWCvL/RudolfGeyerCatalog for details. -->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">

  <!-- Dependent on MicroModal, Bootstrap, Popper and
TippyTip -->
  <script src="https://unpkg.com/micromodal/dist/micromodal.
min.js"></script>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
com/bootstrap/4.1.1/css/bootstrap.min.css">
  <script src="https://maxcdn.bootstrapcdn.com/
bootstrap/4.1.1/js/bootstrap.min.js"></script>
  <script src="https://unpkg.com/popper.js@1"></script>
  <script src="https://unpkg.com/tippy.js@4"></script>

  <!-- Loading interface texts and catalog texts -->
  <script src="textsGeyerCatalog.json.js"></script>
  <script src="textsInterface.json.js"></script>

  <!-- Loading styling rules overriding standard Bootstrap -->
  <link rel="stylesheet" href="extrastyle.css">

  <title>Catalogue of Rudolf Geyer</title>
</head>

<body>
  <!-- Header with link to St. Florian, a title, link to
switch language, and link to Digital Orientalist -->
  <header>
    <div class="container-fluid">
      <div class="row">
        <div class="col-2 headerdiv left">
```

I instructed in JavaScript), the flag will be of the United Kingdom, representing English. If the interface is in English, it will be a German flag. The button to go to my website should be the logo of The Digital Orientalist, but it will be quite small. It should change color when the user hovers their mouse over it—to emphasize it and indicate that it is a button.

As you may notice, all the images are SVG-images. These are images that do not store color values pixel per pixel, but rather store the coordinates of shapes and their colors in an XML-like manner, which browsers nowadays know how to turn into images. They are essentially connect-the-dots puzzles that the computer of the user will solve on the spot. This means that these images are vector-based and will look sharp no matter how small or big you make them. It also means you can edit them in the text editor where you edit your code (see the Productivity section below), for example, to change the aspect ratio or the color.

Below the header comes an introduction section. It is divided into ¾ text and ¼ image. The image is Rudolf Geyer's Ex Libris sticker, which we will discuss further in the CSS section. The text consists of a heading and an actual introduction text. As you may notice by now, the HTML-file does not have the actual title or text, as these are stored in a JSON-file in two languages, and JavaScript will fill these different blocks with the correct texts. It can do so by calling the different elements in the HTML-file by their ID-name, such as *welcomeHeader* and *welcomeBody*. The frequent mentioning of *class* is to give the CSS-file a sign that this or that part needs to be formatted in a certain way.

Below the welcome text, a text box and a button needs to appear to give the user the opportunity to search. Under the search function, a horizontal line should be drawn, indicated by <*hr*>, to separate the interface from the actual catalog.

At the very top of the catalog, a title should appear. As we will see, this title changes frequently, for example, notifying the user of the number of search results. Next to the title, preferably on the same line, three buttons should appear on the far-right to sort the displayed entries (the entire catalog or just the search results) according to the title, author, or year. I did not deem necessary anything more than this. If a place name or something else is important, one can simply search for it.

After the title, an unassuming <*div class='container' id="catalogGeyer"*></*div*>, to be filled by a JavaScript function, defines the entire catalog. Similarly, the *popupInfo* plays an important role later and simply needs to be declared here as part of the structure. With it, we can program a pop-up to show more details of a catalog record. This is necessary since we just defined a lightweight,

```
          <a href="http://www.stift-st-florian.at/en/home.
html" class="btn" id="stiftFlorianTip" title="">
            <img src="iconStFlorian.svg" height="30"
alt="Stift Florian">
          </a>
        </div>
        <div class="col headerdiv middle">
          <h1 class="header" id="headline"></h1>
        </div>
        <div class="col-2 headerdiv right">
          <div class="flags btn" id="flagTip">
            <img src="iconflaggerman.svg" class="flags"
height="16" width="32">
          </div>
          <div class="flags btn DO">
            <img src="iconDigOr.svg" class="flags"
height="16" id="digitalOrientalistTip" onclick="window.
open('http://www.digitalorientalist.com/')"
              onmouseover="this.src='iconDigOrOrange.svg'"
onmouseout="this.src='iconDigOr.svg'" />
          </div>
        </div>
      </div>
    </div>
  </header>

  <!-- Space for welcome text -->
  <p></p>
  <div class="container">
    <div class="row">
      <div class="col-9">
        <h2 class="headline" id="welcomeHeader"></h2>
        <p id="welcomeBody">
          Loading...
        </p>
      </div>
      <div class="col-3 geyerImage">
      </div>
    </div>
```

```
    <!-- Search bar -->
    <p class="font-weight-bold">
      <div class="input-group">
        <input type="text" id="searchTip" class="form-control"
title="" placeholder="">
        <span class="input-group-btn">
          <button class="btn btn-blue" type="button"
id="searchButton"></button>
        </span>
      </div>
    </p>
  </div>

  <!-- Dividing line interface/catalog -->
  <hr>

  <!-- Title and buttons for ordering catalog -->
  <div class="container">
    <div class="row">
      <div class="col-9" id="titleCat">
        <h2 class="headline" id="titleSearch">
        </h2>
      </div>
      <div class="col-3" id="buttonsCat">
        <span id="sortBy"></span>
        <div class="btn-group btn-group-toggle" id="ordering"
data-toggle="buttons">
          <label class="btn btn-outline-secondary">
            <input type="radio" name="options" id="sortTitle"
autocomplete="off">
            <span id="sortTitleCaption"></span>
          </label>
          <label class="btn btn-outline-secondary">
            <input type="radio" name="options"
id="sortAuthor" autocomplete="off">
            <span id="sortAuthorCaption"></span>
          </label>
          <label class="btn btn-outline-secondary">
```

```
            <input type="radio" name="options" id="sortYear"
autocomplete="off">
            <span id="sortYearCaption"></span>
          </label>
        </div>
      </div>
    </div>
  </div>

  <!-- Actual catalog -->
  <div class='container' id="catalogGeyer"></div>

  <!-- Optional pop-up for extra details of an entry -->
  <div id="popupInfo"></div>

  <!-- Footer with link to Stift Florian and link to Digital
Orientalist -->
  <footer>
    <div class="container-fluid">
      <div class="row">
        <p></p>
      </div>
      <div class="row">
        <div class="col-2 footer left">
          <a href="http://www.stift-st-florian.at/en/home.
html" class="btn" title="">
            <img src="iconStFlorian.svg" height="30"
alt="Stift Florian" srcset="">
          </a>
        </div>
        <div class="col footer middle">
        </div>
        <div class="col-2 footer right">
          <a href="http://www.digitalorientalist.com/"
class="btn" title="">
            <img src="iconDigOr.svg" height="20"
alt="The Digital Orientalist" srcset="" onmouseover="this.
src='iconDigOrOrange.svg'" onmouseout="this.src='iconDigOr.svg'">
```

```
            </a>
          </div>
        </div>
      </div>
    </footer>

    <!-- Loading of functionality -->
    <script src="functions.js"></script>

</body>

</html>
```

compact catalog that would in the first instance only show the title, author (and editor), place, and year. A fuller record can then be shown when the user clicks on the entry.

We finally arrive at the footer of the page, which looks a lot like the header but without the 'change language' button. We close with a reference to a JavaScript file that actually is the motor of all our interactivity. It needs to be placed here at the end, for otherwise, it will not function properly. This is because it can only interact with things that are defined before it, so if it is placed at the top of the HTML-file, it would already start running and would want to try to refer to things that are not yet defined, resulting in nothing.

4 Content: JSON

If you have gone through my repository, you will have noticed that there are two JSON-files. In this case, they actually end in the extension .JS, as will be explained later on. One handles the interface (for both languages), the other contains the catalog. I structured the interface-file in a simple manner. At the top level, I defined the keys (the string on the left of the colon) as the same as the ID that the elements in the HTML-file have and to which the different texts belong. From a human-readability point of view, this should make it quite clear where each of these texts goes in the final website. In terms of the vertical structure, the order in which the keys are defined, it seemed to make the most sense to be as faithful to the structure of the website. Thus, header texts are first defined, and the texts for the modal, the pop-up showing the full record details, appear last.

```
var texts =
  {
    “headline”:
      {
        “english”: “Catalogue of the Rudolf Geyer Collection”,
        “german”: “Katalog von dem Nachlass Rudolf Geyer”
      }
    ,
    “stiftFlorianTip”:
      {
        “english”: “Go to Stift Florian’s Website”,
        “german”: “Gehe zur Website von Stift Florian”
      }
    ,
    “digitalOrientalistTip”:
      {
        “english”: “Made by L.W.C. van Lit”,
        “german”: “hergestellt von Dr. L.W.C. van Lit”
      }
    ,
    “flagTip”:
      {
        “english”: “Switch to English”,
        “german”: “Auf deutsch umstellen”
      }
    ,
    “welcomeHeader”:
      {
        “english”: “Welcome”,
        “german”: “Herzlich willkommen”
      }
    ,

var catData = [
  {
    “id”: “http://zotero.org/users/170941/items/8JKDXVME”,
    “type”: “book”,
    “title”: “كتاب شعراء النصرانية”,
    “publisher”: “Maṭbaʿat al-ābāʾ al-mursalīn al-yasūʿiyyīn”,
    “publisher-place”: “Beirut”,
    “event-place”: “Beirut”,
    “abstract”: “al-juzz al-awwal fī shuʿarāʾ al-jāhiliyya”,
```

```
    "shortTitle": "Kitāb shuʿarāʾ al-naṣrāniyya",
    "language": "XVI A 1",
    "editor": [
      {
        "family": "Sheikhu",
        "given": "Louis"
      }
    ],
    "issued": {
      "date-parts": [
        [
          "1890"
        ]
      ]
    }
  },
  {
    "id": "http://zotero.org/users/170941/items/DWFZQGTU",
    "type": "book",
    "title": "The Poems of Ṭufail ibn ʿAuf al-Ghanawī and aṭ-
Ṭirimmāḥ ibn Ḥakīm aṭ-Ṭāʾyī",
    "publisher": "Luzac & Co.",
    "publisher-place": "London",
    "event-place": "London",
    "abstract": "Arabic Text Edited and Translated. \nPrinted
for the Trustees of the ,,E. J. W. Gibb Memorial''",
    "language": "XVI A 2",
    "editor": [
      {
        "family": "Krenkow",
        "given": "F."
      }
    ],
    "issued": {
      "date-parts": [
        [
          "1927"
        ]
      ]
    }
  },
```

Each of these key:value pairs contain another object as the value, with the first key being 'english' and the second being 'german.' Their values are semantically the same, but they store the required text in two languages. This structure will allow for easy deleting, swapping, or adding of languages. If we would want to add an interface in Arabic, we would only have to go through this JSON-file and write a comma after each 'german' value, hit Enter, type "arabic": followed by the text in Arabic in between quotation marks. This is, I think, a better structure than, for example, having at the top level the keys of each language followed by an object containing all the interface elements, because if we would want to adapt some part of the interface, for example, the text of the introduction, we would have to scroll to different places in this file to change the text for each language. In this case, we only need to go to one place and instantly see the same text in different languages, making it easier to accurately change the text for each language.

For the JSON-file generated by Zotero, the only change that I made was to put var catData = at the very beginning of the file and change the extension to .js. This has a technical reason. By using it as a JavaScript file, the loading can simply be done by an HTML-command starting with <script src="..., and then the data is loaded. To make that a valid process, the data still needs to be a JavaScript Object, meaning that it needs a variable name to be stored in. In our case, this variable is called catData (short for catalog data). Once this file is loaded, we can use the catalog data by invoking catData, as we will do in the JavaScript part of this web app. The JSON-file that governs the interface texts was treated similarly, meaning it was in the end changed to a .JS extension, a variable name was included at the beginning, and the file was loaded using the HTML-command. If we had kept the files with a .json-extension, we would have to have used a special JavaScript command called XMLHttpRequest to load the files to the user's computer memory. This command, however, only runs when it is on a server. JavaScript cannot access local files, and this is for security reasons. Understandable as this may be, the use of the XMLHttpRequest command is not a good alternative in our case. This would make it impossible to download all the files belonging to the catalog, opening it on a computer, and making use of it since the files would run on a server. This is a moment where we have to be creative with the available technology and not dive headfirst into making the XMLHttpRequest command work. Rather, we should reconsider our options. The first step, in this case, is to reflect on the typical use case worldwide for JSON. The dominant usage is for sending or receiving data between machines. For example, when we let JavaScript talk to Wikipedia or Weather Underground or any other service (known as an API), the response will be in the shape of a JSON. We may notice that in our case we own the data

ourselves, and it does not change over time (or only rarely do so), so we do not need the same kind of dynamic set up to collect our catalog data. In that case, we might as well change them into JS files and add the extra line at the beginning to give the object a name. This does not change their being and function; we can still treat them as JSON-files, and hence, it will be clearer if we include '.json' in the filename.

It seemed a good idea to use the JSON-file created by Zotero with the smallest amount of tinkering so that we could create a different catalog in a pinch by exporting a different set of entries from Zotero. There are, unfortunately, some drawbacks to this, owing to the particular structure of JSON-files created by Zotero. The chief drawbacks are elements like *ID* and *event-place,* which are pretty much useless for our purpose, but it may also be noted that these files have an overly complicated way of storing names and dates. Once again, we need to reassess the possibilities of the technology to find our best strategy to deal with these drawbacks. One solution is to fiddle around with the JSON-file that Zotero produced and shape it into a much simpler form, a form exactly as we want it. We can do this by using the search and replace function, with regular expressions to capture the part in every entry that we want to be deleted or changed.

In the end, I decided to not make any alterations to the JSON-file but instead make the JavaScript code that extracts information from the catalog more complicated. This way, the catalog is more faithful to the data entry in Zotero, and it will be easier to implement it for another collection.

5 Style: CSS

CSS stands for Cascading Style Sheets. It is one or more sheets (files) that end in .css and which should be called at the beginning in the HTML-file with a <link rel="stylesheet" href="pathAndNameOfStylesheet.css"> command. Such a CSS-file is merely a list of instructions regarding how particular elements should look like. Proper styling is especially necessary as there is an ever larger variety of screen sizes and browsers that users of your product will use. Taking this into account is using what is called 'responsive web design,' which will ensure that your website will not shrivel to an unreadably small size because someone uses a four-year-old smartphone, but instead, it will respond by reshuffling and resizing the elements and the texts in them. Styling is also necessary to make the website actually attractive. A catalog entry stored in a JSON-file is sort of readable by an individual, but once it is presented in a table with ample white space around it, lined out neatly, with

good background colors and a font and font style that fit the context, it will be much more readable and enjoyable. In fact, as more and more websites do look good, users are simply starting to expect a certain level of sharp and attractive design.

```
headline {
  background-image: linear-gradient(to bottom right, rgb(167,
72, 61), rgb(253, 192, 47));
  color: transparent;
  -webkit-background-clip: text;
  background-clip: text;
  }
.geyerImage {
  background:
url(“GeyerExLibris1.jpg”) no-repeat;
   background-size: contain;
   background-repeat: no-repeat;
   width: 100%;
   height: 100%;
   padding-top: 27%; /* (img-height / img-width * container-
width) */
   -webkit-background-size: contain;
   -webkit-transition: all 1s ease-in-out;
   -webkit-border-radius:  10px;
  }
```

Whereas the structure of our website in HTML really is up to us, for CSS, we can and should use already existing resources that are capable of instantly making our site reasonably usable and attractive. Styling controls the look and feel of every aspect of the website, so starting from scratch would be a tedious job, especially since many things like how a button looks and behaves should be sort of similar regardless of the online resource you are building. The responsive part of CSS, too, is equal no matter which project you are working on. Two templates that are currently popular are Bootstrap and Materialize. The first was created internally at Twitter and subsequently released as open source. The second was developed later by students of Carnegie Mellon University. I have used Bootstrap for my catalog. The changes that I wished to make have been put in an additional CSS-sheet, and I have instructed the HTML-file to load this additional file after loading the Bootstrap CSS-file. The ‘cascading’

in Cascading Style Sheets means that if there are two rules about the same thing, it is the last rule that is actually obeyed. Thus, by loading a custom made CSS-file after Bootstrap, we can use Bootstrap as a foundation and make some over-ruling changes to it, defined in the other file. Knowing what exactly can be controlled and which commands to use is a matter of searching the internet, specifically websites like StackOverflow and the documentation for CSS and for Bootstrap. For beginners, it will be beneficial to watch a video or attend a workshop. In most cases, it is only when the need arises that you should look into exactly how to do something.

Perhaps the most noteworthy change is the definition of *headline* and of *geyer*. Headline is used for different headings, such as the title of the welcome text and the title of a holding in the pop-up. In my extra CSS, I made it so that the text of those headings are colored in a gradient. Done well, this can really pop without being overwhelming. Geyer is only used on one element in my HTML-structure, concerning the image that is supposed to appear next to the welcome text. Even though JavaScript is officially in charge of interactivity, you can see that I actually added some interactivity right here in the CSS-file with the transition command. The command needs to be repeated for several different browsers for it to work for all users. CSS also allows to style elements specifically when a user hovers over the element with the mouse pointer. In this case, a similar transition command is used but on a different image. It results in the following effect: when the user hovers over the image, one Ex Libris sticker image appears to morph into another Ex Libris sticker. Note the *width, height*, and *padding-top* instructions, which ensure that the image takes up the entire width of the defined area, which will differ from screen to screen, while still showing the entire height of the image proportionally to the width so that it always retains its correct height to width ratio. Without the padding-top instruction, the image would likely be cut off at some point.

6 Interactivity: JavaScript

In about five-hundred lines of code, the catalog comes to life. These lines are divided over 12 functions which, in some cases, require human interaction to be triggered and, in other cases, are auxiliary to other functions. If you open the JavaScript file in a good code editor and select *Fold All*, the editor will reduce all groups of code that belong together to a single line. In our case, this means that you will instantly see an overview of the division of functions and their triggers.

```
// Copyright L.W. Cornelis van Lit. For details see https://
github.com/LWCvL/RudolfGeyerCatalog
// (f) = function, (e) = event

// 1. Initializing webapp
window.onload = function() {
  // setting initial variables
  searchData = catData;
  shadowCatalog = {};
  accentMap = {
á: “a”, à: “a”, ā: “a”, â: “a”, ä: “ae”, æ: “ae”, é: “e”, è:
“e”, ē: “e”, ë: “e”, í: “i”, ì: “i”, ī: “i”, î: “i”, y: “i”,
ó: “o”, ò: “o”, ō: “o”, ô: “o”, ö: “oe”, ú: “u”, ù: “u”, ū:
“u”, û: “u”, ü: “ue”, ṭ: “t”, ṣ: “s”, ḍ: “d”, ḥ: “h”, ẓ: “z”,
ġ: “gh”, š: “sh”, ǧ: “j”, ḳ: “q”, č: “ch”, ‘: “’”, ’: “’”,
“.”: “ ”, “,”: “ ”, “:”: “ ”, “;”: “ ”, “?”: “ ”, “!”: “ ”
  };

  // Initial building of shadow catalog and interface
  Reducing_Catalogue(catData);
  Switch_Language(“german”);
};

// 2. Creating a shadow-catalog with simplified entries to
easier match a search term in the search function
// Called by (e)window.onload
function Reducing_Catalogue(catalog) {
  // Looping through all catalogue entries
  for (var y = 0; y < catalog.length; y++) {
    // Adding all fields for each entry in one string.
    entryY = “”;
    entryY += “ ” + catalog[y].title;
    entryY += “ ” + catalog[y].shortTitle;
    entryY += “ ” + catalog[y].publisher;
    entryY += “ ” + catalog[y][“publisher-place”];
    entryY += “ ” + Print_Names(catalog[y].author);
    entryY += “ ” + Print_Names(catalog[y].editor);
    entryY += “ ” + Print_Names(catalog[y].translator);
    entryY += “ ” + catalog[y].abstract;
    entryY += “ ” + catalog[y].language;
```

The order in which the functions are written here is based on their logical order, but I did not think all of this out at first, and I did not write these twelve functions neatly in this order. As is often the case, you simply start coding functionality that seems to be of the most immediate need from a usability point of view. By constantly testing the result in a browser, you will see how the code behaves. You will quickly run into issues in how programming languages require you to think and the limitations that JavaScript specifically put on you. At each step, it is good to consider if the functionality you are now thinking of writing is essential, and if the way you want to build that functionality is efficient.

Functions four and five form the core of this code, and to understand the entirety of it, it will be better to start there. The task of function four is to fill the interface with texts. Moreover, when the flag icon at the top-right of the website is clicked, it needs to change the texts to the next language. To keep the structure and content of the website strictly separated, I only declared an empty structure in the HTML-file, which means that when the user visits the website, this function needs to be called to populate all the fields. The function does not make many assumptions. For example, it does not specify the different fields by name but loads the name of those fields dynamically from the interface-JSON. This means we could add or delete certain interface elements by changing the HTML-file and the JSON-file, while leaving this code intact. Similarly, the number or kind of languages there are is not specified. Adding another language to the interface-JSON will be of no problem as long as one does not forget to also create a flag icon in SVG-format for that language. The language the interface is in right now is given to the function, and from there, the next language is established, both the index of the next language and the name. This index is used to access the correct text for all the fields, which is done by a simple *for*-loop, going through all the available elements. Within this loop, some specificity was inescapable, as accessing the text property of all fields could not be done with one command. Some require the command *placeholder*, others *innerHTML*, and others *title*.

Function five, *Render_Table*, creates the catalog underneath the interface. It takes a couple of parameters. First, the catalog to be rendered, in JSON format. Then a specification regarding the type of heading the catalog should have. If you look into the interface-JSON, you will see several different headings such as *beginTitleCatalog* and *noResultsTitleCatalog*. This function takes one of them to give a more specific feel to the rendered catalog. Next, the function takes in the language. Lastly, it takes the number of entries to be rendered, which could have been calculated within the function, but it seemed more readable to parse it into the function. The logic of the function is rather simple: it will

```
    // If field does not exist, 'undefined' will be pushed.
All 'undefined's are now deleted.
    var definedY = entryY.replace(/undefined/g, "");
    // Entire string is stripped of transliteration marks,
punctuation, capitals, and double spaces, and pushed into the
shadow catalogue
    shadowCatalog[y] = Simplify_Term(definedY);
  }
}

// 3. Returning simplification of transliteration signs,
punctuation, double spaces, and capitals
// Called by (f)Reducing_Catalogue and (f)Search_Catalog
function Simplify_Term(inputWord) {
  if (!inputWord) {
    return "";
  }
  inputLower = inputWord.toLowerCase();
  var outputWord = "";
  // Loop through every letter of a word
  for (var i = 0; i < inputLower.length; i++) {
    outputWord += accentMap[inputLower.charAt(i)] ||
inputLower.charAt(i);
  }
  returnedNoSpaces = outputWord.replace(/\s{2,}/g, " ");
  return returnedNoSpaces;
}

// 4. Switching interface to different language (currently
German-English)
// Called by (e)window.onload and (e)flagTip.onclick
function Switch_Language(language) {
  // Array of words that are both keys to the texts-Object
and elementIDs of the webapp
  toBeTranslated = Object.keys(texts);
  // Array which is agnostic as to how many/which languages
there are
  languages = Object.keys(texts[toBeTranslated[0]]);
  // index number of current language
  lanIndex = languages.indexOf(language);
```

create a variable containing a string that represents the HTML-code to display the catalog, for which the *table*-tag is used, and once it has filled that variable with all entries, it will fill the *div* in the HTML-file called *catalogGeyer* with that variable. The last thing it does is set the heading for the catalog.

It seemed cleaner and more flexible to take out the code for rendering one catalog entry as one table item and put it into a separate function called *Render_Table_Entry*. At an early stage, I realized it might be the case that a search function would require the code of this *Render_Table_Entry* function separate from the *Render_Table* function, in which case there are logical grounds to separate the code out as a function. Writing the same code twice or more is bad practice. Not only does it make your code longer than necessary, but it also means that if you want to change something in it, you would also need to change it in all those places. In the end, it turned out I would only call this function from within the *Render_Table* function, so separating the two functions has no logical reasons, but it is still cleaner and more readable. First, the *Render_Table_Entry* function declares a few variables that are filled with the correct texts in the correct language. The only reason to declare those variables is to make the rest of the code more readable. The HTML for the table-entry is generated piece by piece. First, a 'more information' button is generated, which triggers a pop-over that we will discuss later. Then, a variety of different things are included, such as title, author, place, and date. Since some of these things are missing from the catalog entry, they should only be included if they are there. Since the name fields can include many names—multiple authors or multiple editors, for example—one unified function to print these correctly is called, which we will discuss next. For the date and place, we cannot use a *hasOwnProperty*-function since the data structure that Zotero produces is a bit more complicated than the other ones, and JavaScript simply cannot handle this function for that data structure. Writing merely the first part of the name of that data structure will be enough to check for its existence. We could do likewise for the previous ones, the names, but the actual function *hasOwnProperty* seems more readable. Also notice that if the key in a JSON has a dash in it, like *publisher-place*, it cannot be accessed with a dot notation like *data*[*i*].*publisher-place*, but we need to use square brackets and quotes like *data*[*i*][*"publisher-place"*].

Print_Names, function number eight, takes a reference to a catalog entry and returns the names associated with that entry with appropriate formatting. The first thing the function does is to ensure the existence of names. If that is the case, a *for*-loop iterates over all the names and for each name checks if there is a first name, a last name, and a connecting particle (such as the German 'von' or the French 'de'). Since Zotero stores these values in separate

```
  if (lanIndex == languages.length - 1) {
    nextLanIndex = 0;
  } else {
    nextLanIndex = lanIndex + 1;
  }
  nextLanguage = languages[nextLanIndex];

  for (key in toBeTranslated) {
    // Filling different elements of webapp by their ID. Some
need different ways of accessing their content.
    if (toBeTranslated[key].includes("Tip")) {
      document.getElementById(toBeTranslated[key]).title =
Object.values(
        texts[toBeTranslated[key]]
      )[lanIndex];
    } else if (toBeTranslated[key] == "searchBox") {
      document.getElementById("searchTip").placeholder =
Object.values(
        texts[toBeTranslated[key]]
      )[lanIndex];
    } else if (toBeTranslated[key].includes("Catalog")) {
      null;
    } else {
      document.getElementById(toBeTranslated[key]).innerHTML
= Object.values(
        texts[toBeTranslated[key]]
      )[lanIndex];
    }
    // Setting button for switching language to the next one
    document.getElementById("flagTip").innerHTML =
      '<img src="iconflag' +
      nextLanguage +
      '.svg" height="16" width="32" class="flags"
id="flagTip" )">';
    document.getElementById("flagTip").title = Object.
values(texts.flagTip)[
      nextLanIndex
    ];
  }
```

fields, we need to stitch them together. All names are separated by a comma (and a space, of course), and the last name is closed with a period.

Function seven, *Popup_More_Info*, handles the event that a user clicks the circle with an 'i' in it to get more information about a catalog entry. The pop-up relies on what is called a 'modal,' a nice looking overlaying sheet which can be closed by either clicking a close button or by clicking anywhere outside of it. We rely on a third-party library. This is because it takes remarkably smart code to get a really good modal, and if somebody else already offers it for free, we are better off using that. In our case, we use MicroModal. In the HTML-file you may have noticed that we load a JavaScript file called micromodal.min.js; this is what makes it possible to refer to the MicroModal functionality in this *Popup_More_Info*-function. If a better modal library would come along, I could swap out this one for the other relatively easily. In fact, I already did so. Before I knew of MicroModal I used jQuery. jQuery is a multi-function library that has become outdated if used as a framework and it seemed that for a modal, MicroModal performed better.[6] Without getting too bogged down with details about libraries and frameworks, let us consider what this function does because if it works, it is probably good enough for our use case. In the beginning, several variables are defined whose sole purpose is to make the following code more readable. They consist of all the particulars of a catalog entry. Just like creating the table for the catalog was done by filling a variable with the right HTML-encoding and then filling a div in the HTML-file with that variable, so we do the same for the pop-up. To get the modal to work, we need to add a couple of different divs. They can be copied and pasted from an example on the internet. Then, one after the other, the particular details of a catalog entry are considered. We perform a check to see if the entry has a certain detail, and if it is there, it is added to the variable. Notice the use of markup tags like <i> for italics and reference to CSS elements such as class= "text-success" for the abstract.

With these functions, we now have a visible catalog. Of course, we want to add a search and a sort functionality. There are a couple of simple commands we can use to get that working. For the search function, we can use the JavaScript command *.includes*,[7] which will return *True* if the term is part of the catalog entry (it really is that easy). For sorting the entries, we can use *.localeCompare* for strings (such as titles and author names) and *.sort* for numbers

6 You can see the changes by looking through the commit history of my GitHub repository for this code.

7 I first considered filtering the generated table, but this caused a number of issues which I do not think are worth going into.

```
  // Catalogue needs to be re-rendered
  if (document.getElementById("searchTip").value) {
    Search_Catalog(document.getElementById("searchTip").value);
  } else {
    Render_Table(catData, "beginTitleCatalog", lanIndex);
  }

  // Fancy mouse-over tool-tip only activitated when not on
mobile. We simply delete previous instances and create new ones
  [...document.querySelectorAll("*")].forEach(node => {
    if (node._tippy) {
      node._tippy.destroy();
    }
  });
  if (
    /Android|webOS|iPhone|iPad|iPod|BlackBerry/i.
test(navigator.userAgent) ==
    false
  ) {
    tippy.setDefaults({
      animation: "perspective",
      arrow: "true",
      size: "large"
    });
    tippy("#flagTip", {
      content: Object.values(texts.flagTip)[nextLanIndex]
    });
    document.getElementById("flagTip").
removeAttribute("title");
    tippy("#digitalOrientalistTip", {
      content: Object.values(texts.digitalOrientalistTip)
[lanIndex]
    });
    document.getElementById("digitalOrientalistTip").
removeAttribute("title");
    tippy("#stiftFlorianTip", {
      content: Object.values(texts.stiftFlorianTip)[lanIndex]
    });
    document.getElementById("stiftFlorianTip").
removeAttribute("title");
    tippy("#searchTip", {
```

(such as the publication year). What requires some extra work is to get this functioning amidst the messy reality of our catalog. For example, when we search for "tufail," we want the *.includes*-command to return *True* for a catalog entry which includes "Ṭufayl." While our human eyes and brains instantly see the equivalence of tufail and Ṭufayl, a computer strictly compares the character set of the two terms, concluding that they are not the same. Similarly, when sorting entries by year, if a publication does not have the Gregorian year 1871 printed on the cover but instead the Hijrī year of 1288, it should not be sorted as though it comes from the Gregorian year 1288, but it should be sorted amidst books from 1871. While some of these things are better taken care of within JavaScript, some are not. For the year issue, as mentioned before, it was easier to normalize the data entry. That is to say, I went back to Zotero and calculated the Gregorian year for every date only given in the Hijrī calendar. I then added the Hijrī date to the 'abstract' field. Finally, I exported all the catalog entries again to obtain an updated JSON-file.

For the issue of typographically different but semantically equivalent strings, we can use function three, *Simplify_Term*. This function takes any string, changes all upper case letters into lower case, and then applies a series of reductions to it that are specifically designed for the kind of terms and letters used in transliterating Islamic languages (in various transliteration schemes). This schema needs to be defined somewhere. It could have been stored in the interface-json, but I decided to store it in the JavaScript file. It is called *accentMap* and sits in the first function, which is executed as soon as somebody enters the website. Here, we can see how an *ā* is defined to be reduced to an *a*, and an *š* becomes an *sh*, and so forth. Thus, *Šāhnāma* becomes *shahnama*. The function *Simplify_Term* goes through every letter of the string it is given and performs the reduction on it. Then, it deletes any extra spaces after which it returns the resulting string.

With the function *Simplify_Term* discussed, we can look at function two, *Reducing_Catalogue*. After experimentation, I found out that the search function probably worked best if we created another version of the catalog that would only contain the entries in reduced form. *catData* is the object that contains the catalog, and *searchData* will now become the object with the reduced catalog, what I call a shadow catalog. At first, I used the simple commands *.values*, which would give you all the values of an entry, such as title, place, and year; and I used *.join*, which takes all those values and stores them together in one long string. Elegant as this solution was, it was not performing well because the date and names are stored in a more complicated form than this approach allows. I finally settled on filling a variable with individual values of an entry, making profitable use of the function *Print_Names*. Since I did not perform

```
      content: Object.values(texts.searchTip)[lanIndex]
    });
    document.getElementById("searchTip").
removeAttribute("title");
  }
}
document.getElementById("flagTip").onclick = function() {
  Switch_Language(nextLanguage);
};

// 5. Rendering the table of catalogue entries
// Called by (f)Switch_Language and (f)Search_Catalog
function Render_Table(data, heading, language, numberEntries) {
  document.getElementById("catalogGeyer").innerHTML = "";
  htmlTableCat = '<table id="tableCatalog" class="table
table-striped"><tbody>';
  for (i = 0; i < data.length; i++) {
    Render_Table_Entry(data, i, language);
  }

  htmlTableCat += "</tbody></table>";

  document.getElementById("catalogGeyer").innerHTML =
htmlTableCat;
  if (typeof numberEntries !== "undefined") {
    document.getElementById("titleSearch").innerHTML =
      Object.values(texts[heading])[language] + " " +
numberEntries;
  } else {
    document.getElementById("titleSearch").innerHTML = Object.
values(
      texts[heading]
    )[language];
  }
}

// 6. Rendering one entry of the table
// Called by (f)Render_Table
function Render_Table_Entry(data, i, language) {
  var authorLan = Object.values(texts.authorCatalog)[language];
```

if-then checks to see if a particular value even exists, I got many 'undefined' in my string, which can be deleted at the end, after which the *Simplify_Term* function reduces the entire string to complete the entry for the shadow catalog.

We now arrive at function nine, *Search_Catalog*. Obviously, this function requires a string as its input, the search term, which it normally gets from the text box with the ID *searchTip*. The function is triggered by the user, who can click the search button, or hit the Enter key when the focus is on the text box. This last bit is established with the method *addEventListener*, which is a little function that keeps its ears to a specific type of event, and when it hears it go off, it will perform some code. In this case, this code is triggered every time the user releases a key when typing in the text box of the search bar. When the last key was Enter, the listener will engage the *Search_Catalog*-function. Additionally, I provided instructions that if the user has deleted all the text from the text box, the entire catalog should be rendered again. I experimented with letting the *Search_Catalog*-function fire off every time the user pressed a key to get the experience of a live update, so that with every additional letter the user types, the number of entries is restricted. With a catalog of over 1,500 entries, this proved to be too heavy on the calculation side; the code could not be executed fast enough to get a snappy feel. In its current state, there is still not an instant experience of the result, but since the user needs to press the Search button or hit Enter, there is a greater expectancy on the user's side that it can take a fraction of a second for the catalog to update.[8] The search function itself is fairly straightforward. First, it reduces the search term the user provided by means of the *Simplify_Term*-function. Then, it runs through the shadow catalog and sees if the search term is contained in it. If so, that specific entry will be pushed from the catalog (*catData*) into a new object called *searchData*. This way, we can build a subset of the catalog containing only those entries in which the search term is present. Perhaps you wondered why *Render_Table* took as one of its parameters something called *data*. Now you can see that we can give *Render_Table* either the entire catalog (*catData*) or only a subset related to a search term (*searchData*).

The functions ten, eleven, and twelve are fairly similar. I created them by looking up examples of how to arrange objects alphabetically, adapting them to my own situation. For example, I could not be assured that every item had a title, so I built a check for that. Had I not done that, not all pairs would be

8 Since the search function is embedded in JavaScript and therefore loaded onto the client-side, the experience of searching is still pleasant enough in that no page redirect or refresh is necessary, as would be the case if the search query had to be given to server-side code and then the results given back to the client.

```
  var editorLan = Object.values(texts.editorCatalog)[language];
  var translatorLan = Object.values(texts.translatorCatalog)
[language];
  var noAuthorLan = Object.values(texts.noAuthorCatalog)
[language];

  htmlTableCat +=
    '<tr><td><div class="entry text text-left"><p class="text
font-weight-bold"><svg onclick="Popup_More_Info(' +
    i +
    ", " +
    language +
    ')" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 50
50" width="15px" height="15px"><path class="iconModal"
d="M25,2C12.297,2,2,12.297,2,25s10.297,23,23,23s23-10.297,23-
23S37.703,2,25,2z M25,11c1.657,0,3,1.343,3,3s-1.343,3-3,3
s-3-1.343-3-3S23.343,11,25,11z M29,38h-2h-4h-2v-2h2V23h-2v-
2h2h4v2v13h2V38z"/></svg> ' +
    data[i].title +
    '</p><p class="text-muted text font-weight-light">';
  // Cannot be sure if entry has an author, date, and place,
so must check it first.
  if (data[i].hasOwnProperty("author")) {
    htmlTableCat += authorLan + Print_Names(data[i].author);
  } else {
    htmlTableCat += "<i>" + noAuthorLan + "</i>";
  }
  if (data[i].hasOwnProperty("editor")) {
    htmlTableCat += editorLan + Print_Names(data[i].editor);
  }
  if (data[i].hasOwnProperty("translator") == true) {
    htmlTableCat += translatorLan + Print_Names(data[i].
translator);
  }
  htmlTableCat +=
    '</p></div></td><div class="entry text text-right"><td><p
class="text blue">';
  if (data[i].issued) {
    htmlTableCat += data[i].issued["date-parts"];
  }
```

```
    htmlTableCat += "<br>";
    if (data[i]["publisher-place"]) {
      htmlTableCat += data[i]["publisher-place"];
    }
    htmlTableCat += "</p></div></td></tr>";
}

// 7. Generating more information on item in a pop-up
// Called by (f)Render_Table_Entry
function Popup_More_Info(number, language) {
    // For readability further on, these variables are defined
here.
    var authorLan = Object.values(texts.authorCatalog)[language];
    var editorLan = Object.values(texts.editorCatalog)
[language];
    var translatorLan = Object.values(texts.translatorCatalog)
[language];
    var noAuthorLan = Object.values(texts.noAuthorCatalog)
[language];
    var numberVolumesLan = Object.values(texts.
numberVolumesCatalog)[language];
    var publisherLan = Object.values(texts.publisherCatalog)
[language];
    var placeLan = Object.values(texts.placeCatalog)[language];
    var yearLan = Object.values(texts.yearCatalog)[language];
    var additionalLan = Object.values(texts.additionalCatalog)
[language];
    var callNumberLan = Object.values(texts.callNumberCatalog)
[language];
    var urlLan = Object.values(texts.urlCatalog)[language];
    var closeLan = Object.values(texts.closeCatalog)[language];

    // The div already declared in the HTML is popupInfo, the
div dynamically created is popupModalInfo. All of the dynamic
html is inserted in popupInfo, and then popupModalInfo is
made into a modal (pop-up) wih the package MicroModal.

    htmlPopupInfo =
      "<div class='modal micromodal-slide' id='popupModalInfo'
aria-hidden='true'>" +
```

```
    "<div class='modal__overlay' tabindex='-1' data-
micromodal-close>" +
    "<div class='modal__container' role='dialog' aria-
modal='true' aria-labelledby='modal-1-title'>" +
    "<header class='modal__header'>" +
    "<h3 class='headline' id='modal-1-title'>" +
    searchData[number].title +
    "</h3>" +
    "<button class='modal__close' aria-label='Close modal'
data-micromodal-close></button>" +
    "</header>" +
    "<main class='modal__content' id='modal-1-content'>" +
    "<p>";

  if (searchData[number].hasOwnProperty("shortTitle")) {
    htmlPopupInfo += "<i>" + searchData[number].shortTitle +
"</i><br>";
  }
  if (searchData[number].hasOwnProperty("author")) {
    htmlPopupInfo +=
      authorLan + Print_Names(searchData[number].author) +
"<br>";
  }
  if (searchData[number].hasOwnProperty("editor")) {
    htmlPopupInfo +=
      editorLan + Print_Names(searchData[number].editor) +
"<br>";
  }
  if (searchData[number].hasOwnProperty("translator")) {
    htmlPopupInfo +=
      translatorLan + Print_Names(searchData[number].
translator) + "<br>";
  }
  if (
    searchData[number].hasOwnProperty("author") == false &&
    searchData[number].hasOwnProperty("editor") == false &&
    searchData[number].hasOwnProperty("translator") == false
  ) {
    htmlPopupInfo += noAuthorLan + "<br>";
```

```
  }
  if (searchData[number].hasOwnProperty("number-of-volumes"))
{
    htmlPopupInfo +=
      numberVolumesLan + searchData[number]["number-of-
volumes"] + "<br>";
  }
  if (searchData[number].hasOwnProperty("publisher")) {
    htmlPopupInfo += publisherLan + searchData[number].
publisher + "<br>";
  }
  if (searchData[number].hasOwnProperty("publisher-place")) {
    htmlPopupInfo += placeLan + searchData[number]
["publisher-place"] + "<br>";
  }
  if (searchData[number].hasOwnProperty("issued")) {
    htmlPopupInfo += yearLan + searchData[number].
issued["date-parts"][0][0];
  }
  if (searchData[number].hasOwnProperty("abstract")) {
    htmlPopupInfo +=
      '<hr><p class="text-success"><b>' +
      additionalLan +
      "</b><br>" +
      searchData[number].abstract +
      "</p>";
  }
  if (searchData[number].hasOwnProperty("language")) {
    htmlPopupInfo +=
      "<hr>" +
      callNumberLan +
      searchData[number].language +
      " (" +
      searchData[number].type +
      ")<br>";
  }
  if (searchData[number].hasOwnProperty("URL")) {
    htmlPopupInfo +=
      '<a href="' +
      searchData[number].URL +
```

```
        '" target="_blank">' +
        urlLan +
        "</a><br>";
    }
    htmlPopupInfo +=
      "</p>" +
      "<button class='btn-blue' data-micromodal-close aria-
label='Close this dialog window'>" +
      closeLan +
      "</button>" +
      "</main>" +
      "</div>" +
      "</div>" +
      "</div>";

    document.getElementById("popupInfo").innerHTML =
htmlPopupInfo;
    MicroModal.init({
      openTrigger: "data-custom-open",
      closeTrigger: "data-custom-close",
      disableScroll: true,
      disableFocus: false,
      awaitCloseAnimation: true,
      debugMode: true
    });
    MicroModal.show("popupModalInfo");
}

// 8. Returning all the names for a given category and prints
them in the format 'first' - 'preposition' - 'last'.
// Called by (f)Render_Table_Entry and (f)Popup_More_Info
function Print_Names(entry) {
    if (!entry) {
      return;
    }
    // This will be the container for all names. Note that in any
category (author, editor, translator) there could be multiple
persons and each person's name consist of several elements
    namesString = "";
```

```
  // We are aiming here for a formatting of FIRSTNAME-
PREPOSITION-LASTNAME. Further, persons are separated by a
comma and the whole list ends with a period.
  for (var name = 0; name < entry.length; name++) {
    if (entry[name]["given"]) {
      namesString += entry[name]["given"];
    }
    if (entry[name].hasOwnProperty("non-dropping-particle")) {
      namesString += " " + entry[name]["non-dropping-particle"];
    }
    namesString += " " + entry[name]["family"];
    if (name == entry.length - 1) {
      namesString += ". ";
    } else {
      namesString += ", ";
    }
  }
  return namesString;
}

// 9. Searching keyword and showing results
// Variously called to initiate search (clicking button or
hitting Enter) or re-render catalog (if order is changed)
function Search_Catalog(input) {
  // Final result will be a subset of the catalog
  searchData = [];
  // We will not compare search term with catalog, but
simplified version of search term with shadow catalog.
  normalizedInput = Simplify_Term(input);
  for (i = 0; i < catData.length; i++) {
    if (shadowCatalog[i].includes(normalizedInput)) {
      searchData.push(catData[i]);
    }
  }

  // Analyze resulting subset; if zero, render entire catalog
with heading title 'no results'
  if (searchData.length == 0) {
    Render_Table(catData, "noResultsTitleCatalog", lanIndex);
```

```
  } else {
    Render_Table(searchData, “foundItemsCatalog”, lanIndex,
searchData.length);
  }
}
document.getElementById(“searchTip”).addEventListener(“keyup”,
function(event) {
  event.preventDefault();
  // 13 is ‘Enter’-key
  if (
    event.keyCode === 13 &&
    document.getElementById(“searchTip”).value != “”
  ) {
    Search_Catalog(document.getElementById(“searchTip”).value);
    // If user deletes search term, render entire catalog again
  } else if (document.getElementById(“searchTip”).value == “”) {
    Render_Table(catData, “againTitleCatalog”, lanIndex);
  }
});
document.getElementById(“searchButton”).onclick = function() {
  if (document.getElementById(“searchTip”).value != “”) {
    Search_Catalog(document.getElementById(“searchTip”).value);
  }
};

// 10. Sort and render search results or entire catalog by
title, putting empty titles at the top
// Called by (e)sortTitleCaption.click
function Indexing_Title_Catalog() {
  catData.sort(function(a, b) {
    if (a.hasOwnProperty(“title”)) {
      firstTitle = a.title;
    } else {
      firstTitle = “”;
    }
    if (b.hasOwnProperty(“title”)) {
      secondTitle = b.title;
    } else {
      secondTitle = “”;
    }
```

```
    return firstTitle.toLowerCase().
localeCompare(secondTitle.toLowerCase());
  });
  // New shadow catalog is necessary, to keep actual and
shadow catalog in same order
  Reducing_Catalogue(catData);
}
document
  .getElementById("sortTitleCaption")
  .addEventListener("click", function() {
    Indexing_Title_Catalog();
    if (document.getElementById("searchTip").value != "") {
      // This ensures only the search results are shown when
ordering
      Search_Catalog(document.getElementById("searchTip").
value);
    } else {
      Render_Table(catData, "beginTitleCatalog", lanIndex);
    }
  });

// 11. Sort and render search results or entire catalog by
author. If no author, then editor or translator. Empty ones
put at top.
// Called by (e)sortAuthorCaption.click
function Indexing_Author_Catalog() {
  catData.sort(function(a, b) {
    if (a.hasOwnProperty("author")) {
      firstAuthor = a.author[0]["family"];
    } else if (a.hasOwnProperty("editor")) {
      firstAuthor = a.editor[0]["family"];
    } else if (a.hasOwnProperty("translator")) {
      firstAuthor = a.translator[0]["family"];
    } else {
      firstAuthor = "";
    }
    if (b.hasOwnProperty("author")) {
      secondAuthor = b.author[0]["family"];
    } else if (b.hasOwnProperty("editor")) {
      secondAuthor = b.editor[0]["family"];
```

```
    } else if (b.hasOwnProperty("translator")) {
      secondAuthor = b.translator[0]["family"];
    } else {
      secondAuthor = "";
    }
    return firstAuthor.toLowerCase().
localeCompare(secondAuthor.toLowerCase());
  });
  // New shadow catalog is necessary, to keep actual and
shadow catalog in same order
  Reducing_Catalogue(catData);
}
document
  .getElementById("sortAuthorCaption")
  .addEventListener("click", function() {
    Indexing_Author_Catalog();
    if (document.getElementById("searchTip").value != "") {
      // This ensures only the search results are shown when
ordering
      Search_Catalog(document.getElementById("searchTip").
value);
    } else {
      Render_Table(catData, "beginTitleCatalog", lanIndex);
    }
  });

// 12. Sort and render search results or entire catalog by
year, putting empty years at the top
// Called by (e)sortYearCaption.click
function Indexing_Year_Catalog() {
  catData.sort(function(a, b) {
    if (a.hasOwnProperty("issued")) {
      // Invariably, if there is an 'issued' key, then the
actual year is stored two levels deeper
      firstYear = a.issued["date-parts"][0][0];
    } else {
      firstYear = "0";
    }
    if (b.hasOwnProperty("issued")) {
```

```
      secondYear = b.issued["date-parts"][0][0];
    } else {
      secondYear = "0";
    }
    return parseInt(firstYear) - parseInt(secondYear);
  });
  // New shadow catalog is necessary, to keep actual and
shadow catalog in same order
  Reducing_Catalogue(catData);
}
document
  .getElementById("sortYearCaption")
  .addEventListener("click", function() {
    Indexing_Year_Catalog();
    if (document.getElementById("searchTip").value != "") {
      // This ensures only the search results are shown when
ordering
      Search_Catalog(document.getElementById("searchTip").
value);
    } else {
      Render_Table(catData, "beginTitleCatalog", lanIndex);
    }
  });
```

compared and sorted, leaving the entries without a title scattered throughout the order of the other entries. Now, with this extra check, all entries without a title are sorted on top. When I discovered this flaw, I not only fixed the code but also decided to return to Zotero and try to give every entry a title, even if there was nothing on the cover. This interplay between coding the catalog and improving the catalog data is to be expected and might occur several times while setting up the catalog. At other times, you will run into issues for which cleaning up or improving the data will not help. For example, the sorting by author name is inherently messy, as multiple persons can be assigned to one item. In such cases, you can, at first, opt to make up the rules that entries should first be sorted by the original author; then, if there is none by the editor, and then, if there is not an editor, a translator.

As a last note, you may be wondering what the code in the first function does. This controls a fancy tool tip functionality when you hover with your mouse over a button or an element. Including this is as easy as including

scripts from Popper.js and TippyTip. This, of course, adds some file size to the website when somebody uses it, but in this case, I did not see an issue with that. The only thing that needed manual instruction is when somebody looks at the catalog from a touchscreen device. Since you cannot hover your mouse on a touchscreen device, TippyTip did not respond correctly, so it is better to simply disable the functionality for tablets and smartphones.

Much more functionality could be built. For example, right now, the sorting is very simple. We could imagine that the same button could sort in two directions (A-Z, and Z-A), or that when sorting by year, given an equal year number, the entries are sorted alphabetical by title. Perhaps performance could be optimized to run all of this faster and smoother. This is always project-dependent. Given the relatively small scale of the catalog, I decided that the functionality as it is is enough and adequate.

7 Productivity: Code Editor and Code Repository

Get where you need to be through the path of least resistance. Developing is not our core business. It is just a tool that we use and discard. And what we want is a working, finished product. It can be refreshing to remind ourselves of these maxims. It is all too easy to be blinded by the latest possibilities in web development, which come with a learning curve that will set back your deadline. If it works, it is alright, we do not need to produce perfect code. In case of doubt, rely on more foundational technology, because with them the chances of being supported for a long time and easily fixing your mistakes or imperfections, later on, are high. For most purposes, we do not need to worry about optimization in terms of processing time and download time. If a JavaScript module claims to do something you want, then it probably is best to include and make use of it.

The most important aspect of productivity is making sure you write your code with a good editor. Writing any of these files, HTML, CSS, JSON, or JS could be done in a text editor as simple as Notepad (Windows) or TextEdit (MacOS)—but that would be madness. I wrote my code for the catalog in Visual Studio Code, which is made by Microsoft and free to use. There are other fine applications out there, and in the future, no doubt, will there be new contenders that could make life even more easy. When in doubt, take the seemingly more popular choice. Several reasons make VS Code great. First of all, in VS Code, you do not just open one file, but rather a project, represented by a folder on your computer. All the files in the folder are shown in a list in VS Code, and by clicking on each file, you open that file in a tab. This makes it

incredibly easy to code in HTML, CSS, JSON, and JS simultaneously. VS Code displays your code in a color scheme, using one color for each type of thing (a method, attribute, or variable) and setting the background color to something other than white. I prefer a very dark grey as a background color, which is easy on the eyes, especially at night. With one key combination, *Shift+Alt+f*, the program will reformat your code to make it look neat and tidy again, indenting lines of code to show it belongs to a function, *if*-statement or *for*-loop. You can shrink or expand parts of the code; for example, if you write a long function, you can minimize it to its first line only so that you keep an overview of the surrounding code. Speaking of overview, on the right, there is a vertical visualization of the totality of your code, which you can use to browse through it. The real power of a program like VS Code comes in the assistance it gives in writing code. For example, in an HTML-file, only typing the exclamation mark and hitting enter will give you a boilerplate HTML-file, which includes a declaration of the file being HTML, a header with some information pre-filled, and body-tags. Writing a period and a word and then hitting enter will create div-tags with that word as a class, and doing the same with a pound-sign, #, will make a division with the word as its ID, and so forth.[9]

Extensions can be downloaded for the VS Code, which expands its functionality. For example, I use one called Live Server. When you open your index.html and click on Go Live in the bottom bar, your browser will be opened with the website you are creating. Every time you adjust any of the files in the project, the webpage reloads in the browser to give you an automatic update of what the website looks like and how it behaves under the adjusted code. This works even better when you have a second screen to place that browser window in so that on one screen you code, and the other you see the result.[10] Such extensions and other advanced features like debugging make VS Code a really great choice. It is also easy to pick up, but as you progress and become more skilled with coding, you do not need to switch to more professional software: VS Code is a popular choice among many professionals.

More advanced tools to help you have to do with version control. In JavaScript, you can often rely on code that somebody else has already come up with and is willing to share for free. For this, you will need to install packages, just as is the case with Python (see Chapter Seven). What *pip* is for Python, *npm* is for JavaScript. With npm, you can get those packages; just be sure that you have the version you want, and include them in any of your projects.

9 VS Code does this through integration of Emmet.

10 I used my iPad as a second screen with the app Duet Display, using a Mounty from TeniDesign to fasten my iPad on the side of my MacBook screen.

The code you write yourself also needs to be controlled, of course, for which currently one of the best options is the software Git, especially as it is offered through the free service GitHub. With GitHub, it should be noted, you do not only get version control but also a reliable way to distribute and collaborate on code. A free account on GitHub will only allow you to make public repositories, so be careful not to use it if you wish to keep your code to yourself. The upside is that with GitHub you also get free hosting and, by adjusting a few settings, your code is not just readable as code in the repository but can be seen working as the website it is supposed to be. This is convenient to demonstrate the project to others and get a reliable sense of how it performs online. You will do well to spend half an hour understanding the philosophy and mechanics behind Git and Github. VS Code supports Git right in the editor, allowing you to see at a glance how you have changed your code since the last save in the repository (which Git calls a 'push').

A note of caution needs to be made about GitHub and, in fact, about all software I refer to: at this moment of writing, they perform really well and are free to use, but both aspects could change over time. GitHub is a particularly good example of this, as its free use and absence of advertisements give off the impression of being part of a public space. The public space on the internet is, in fact, really small. Wikipedia and The Internet Archive are notable examples of stable, future-proof, non-profit organizations committed to keeping their websites open for free, but GitHub is in fact owned by a for-profit company. If it is bought up or pressured by shareholders, it could very well change its services. We have seen such a change of course with Academia.edu. This website purported to create a public sphere for scientists and scholars but shifted in one year, 2016, towards an aggressive strategy to shake money out of their users.[11] With whatever we do, it seems to me to be best to think in terms of abstract ideals and needs and consider which actual technical tool suits you best, keeping an especially close eye to the possibility to exit a certain technology without losing data. In other words: If the company that produces a certain software goes bankrupt, will I able to port my data to other software? Or is it now trapped inside the unstable, unsupported software? You should ideally have a positive answer to this question.

11 In early 2016, Academia.edu started charging money to have your new publication seen by others as 'recommended', a move so brazen it was mistaken at first for a scam, cf. Ruff, C. "Scholars Criticize Academia.edu Proposal to Charge Authors for Recommendations." *The Chronicle of Higher Education*, January 29, 2016. Later that year, a 'premium' service was introduced, broadly criticized as predatory, cf. Bond, S. "Dear Scholars, Delete Your Account At Academia.Edu." *Forbes*, January 23, 2017.

The last point on productivity has to do with knowledge acquirement. In the end, what will likely be the best way for us, students and scholars, to learn web development is to skim books and apply and fiddle around with actual code. There is, however, a large amount of consumable media such as videos on YouTube and podcasts, which are also helpful. We would not want to spend our working hours on them, but by subscribing to some handpicked channels, you may find yourself using these to fill the time you use to relax: for example, while exercising or while traveling on public transport. The best one to choose, in my experience, aim for a beginners-to-intermediate level and do not aim to cover the latest news but do long-form discussions of best practices. Even if at first you do not understand what they are saying, there is merit in listening. Over time, you will slowly pick up the vocabulary currently in use by web developers, and you will get a sense of what is currently hot and what is not.

8 Quantitative Analysis of the Collection

Having a data set like this in JSON format makes it ripe for quantitative analysis. To get started with that, we can simply look through the list of items in Zotero. We can write some simple calculations ourselves, and we can rely on already developed JavaScript components for more complicated things.

In the current method of cataloging, a total of 1528 objects were registered. The vast majority of them, 93%, are books. The rest are mostly journal articles and, notably, a total of twenty-six manuscripts. To display this properly, we would need a pie chart that first divides 93% books to 7% other materials, with the 'other materials' clickable to show a new pie chart dividing up these other materials into 70% journal articles, 24% manuscripts, 4% book sections, and 2% other kinds of documents. Obtaining this information was as simple as sorting the items in Zotero according to kind and then selecting all of one kind to see the number of items.

To extract more specific information, we do not need to go over to Python, but we can insert a few temporary lines of code in the JavaScript we already have. Let us start with looking at the year of publication. At the function *Render_Table*, just before the first for-loop, we can insert *var catchYears*, and inside the loop, we can then write *if* (*data*[*i*].*issued*) {*catchYears* += *data*[*i*].*issued*[*"date-parts"*] + *"v";*}. Right after the loop, we include *console.log*(*catchYears*). If we run the catalog in a browser, we can then open the console[12] to see the result:

12 Safari: Develop > Show JavaScript Console. Chrome: View > Developer > JavaScript Console. Firefox: Tools > Web Developer > Web Console.

namely, it will have printed the variable *catchYears,* which is a long string of dates separated by the letter v. This should be copied and pasted into your code editor. With regular expressions, we can break the line at every 'v' (and deleted that 'v,' of course). What is left is a list of more than 1200 lines on each of which is a date. From here, we can use a standard JS library to display bar charts. I used graph.js to make it. I added myself a few lines to give the bars a random color. I also added another package called chartjs-plugin-annotation.js to add a line indicating the halfway point. The result is a bar chart that gives an impression when the books in Geyer's possession were published. Notably, the halfway point lies in between 1902 and 1903, or, in other words, Geyer acquired many books as they were coming hot from the press and invested much less in older books.

Another piece of information is the place of publication. For analyzing this, we need to use uniform names. Thus, Petrograd becomes Saint Petersburg, and Den Haag becomes The Hague. Leningrad was similarly filed under Saint Petersburg to find it alongside the other publications from that city. Christianiae was filed under Copenhagen, and Bulaq and Misr were filed under Cairo. In all those cases, a note in the abstract was included to make the user aware of what is actually on the title page of the item. We could do a similar approach as in the previous paragraph, and this way we could find out how many uniquely different place names are used (96 in the case of Geyer's collection). However, place names also call out for plotting a map to visualize the information. In fact, plotting data on a map is a good example to see how knowing even a little bit of JavaScript can really make a difference. If we would have no skill whatsoever, we would be left with very few options. One possibility would be Palladio, a DH tool developed at Stanford. It wants the following:

– One Excel sheet with one column, indicating the originating city and destination (Vienna), separated by a comma, and 1209 rows.
– One Excel sheet with one column, 96 rows, with unique city name, a comma, longitude and latitude in between quotation marks, and separated by a comma.

Loading this into Palladio gives a pretty looking map with curved lines towards Vienna whose thickness is related to their prevalence in the data set. Besides the idiosyncratic demands on the input, there is no way of getting that map out of the Palladio work environment. They themselves suggest that making a screenshot is the best way. In the spirit of Chapter Four, we see yet again how a well-funded team project has failed to deliver a fully functioning product. With basic knowledge of JavaScript, we can make use of a library called *amCharts*

FIGURE 6.1 Above: Map created with Palladio. Below: Interactive map created with amCharts

that gives us much more flexibility and is, in general, just better than Palladio. Of course, amCharts also requires the data to be in a specific format, and the following is a short description of how I changed the place name data as it was in Palladio's requirements towards amCharts preferred format. From the Excel sheet with all the place names, I copied the column of 1209 city names into a new column and selected Data > Table Tools > Remove Duplicates. Then, in a third column, I typed =COUNTIF(A:A;B1). This counts the number of times the value in B1 is to be found in column A. I then selected and copied this cell, pasting it along the C column for as many rows as there were in column B (the list of unique cities). This will get you a unique list of cities together with the number of occurrences. Then, I created the column with latitude and longitude information in column D, mapping it according to the names of column B. For this, I used a website, on which I manually searched for the city name and retrieved its geolocation. I chose this over an automated approach using an API because the number of 96 cities was manageable, and a fair number of them are too obscure to be handled automatically as they would be misidentified or not identified at all. Finally, in column E, I made sure to get the name, location, and occurrence together, like the following:

```
Washington,“38.907192, -77.036871”,4
```

I copied this column into my code editor, Visual Studio Code, which has an advanced Find and Replace function. With the regular expressions function on, with trial and error, I landed on the following Find expression:

```
(([a-z|A-Z]|ü|ö|)+),“(\w.*), (-?\w.*)”,(\w*)
```

And for the Replace expression I took:

```
{\n“id”: “$1”,\n“svgPath”: targetSVG,\n“title”:
“$1”,\n“latitude”: $3,\n“longitude”: $4,\n“scale”:
$5*scaleVar\n},
```

The result can then be inserted into the JavaScript code of amCharts under “images” of the var map. Just above the var map, I made sure to include *var scaleCity = 0.0; var scaleLine = 0.2;*. This will allow us to play around with the city sizes and line sizes according to their occurrence. I have found 0 and 0.2 to be good values for this data set. The result is a dynamic map of the world with dots for each city that produced at least one book that made it into Geyer's collection, and curved lines towards Vienna whose thickness is commensurate with the number of books coming from that city. This map is zoomable, and when the user hovers the cursor over a city, it will see the city name and the number of items published there.

In the last two chapters, we laid the foundations for working with digitized manuscripts, concluding that we can best store text in a plain text file format such as XML or JSON, and store symbols or shapes in a vector image file format such as SVG, and, lastly, to inform and connect all these matters by academic standards such as TEI and IIIF. In this chapter, we learned of the basic skills to practically put all these files together to create a visual appearance—which we can call a website, web app, digital edition, or digital catalog. The two most important lessons to draw from this and the previous chapters are that technology only remains as powerful as the user wielding it and that we do not need to know everything—just those aspects that help us build a solution. This is why it is very important to keep an open and creative mind. With knowledge of the fundamentals of different technologies, it will be relatively easy to understand how a certain problem can be solved and whether you can learn how to implement that solution in a time that is sufficiently short enough. Now that we have acquired a few basic assets for our practical toolbox, let us, in the next chapter, reach for a higher level of technical skill by delving into the programming language Python and its application to codex images.