The process of retrieving schedule data from the Novasoftware web application

John Rapp Farnes

April 18, 2016

Abstract

The process of retrieving and extracting schedule data from the Novasoftware web application is derived and described, providing general instructions independent of programming language and platform. An implementation of the process is included, together with installation and usage instructions. Some possible applications are discussed and reviewed.

Contents

1 Introduction and background

I have observed that many Swedish schools including Värmdö Gymnasium, the high school that I attend, rely on the Novasoftware schedule application. I have further recognized that there is a general consensus among students that the web based schedule viewer provided with it does not reach the expectations of a modern application, especially considering its design and mobile experience. Because of this it may be of interest to create another viewing application, satisfying the needs of a modern application, using the schedule data provided by Novasoftware. Beacause Novasoftware provides no public API (Application Programming Interface) or other simple ways of extracting this data, a method of obtaining the information from the provided public interfaces is required.

In the case of the high school I attend, the school provides a separate schedule for Tuesdays via an independent application. This inconvenience of using separate applications for different days of the week further increases the relevance of extracting the Novasoftware data, in order to unify the two schedules into one. Also, when thinking about the implications of having access to this data I realized that it could be used for research purposes; analyzing the schedules of different schools and drawing conclusions from them.

Any specific ways to use the retrieved data are outside the scope of this paper; I'm only mentioning these possibilities since they may interest the reader and because of their role as motivation for the project.

1.1 Purpose

The purpose of this paper is to provide a complete documentation, as well as a working implementation, of the process of fetching, parsing and extracting data from the Novasoftware public web interface.

The documentation can act a guide for the realization of the whole or parts of the process, while the included implementation provides a working program capable of extracting current schedule data. This data may for example be used to create custom viewers or for analysis. The documentation also acts as a description of the provided implementation so that it may be examined and reviewed.

1.2 Goals

The goal of the documentation is that it may be implemented in any programming language or platform. My goal with the program is that it should be simple to install and use and that clear instructions will be provided with it.

Further, the data derived from the presented process must be general and follow a logical format.

The process should be if not in its original form, then with only minor adjustments, appliable to every schedule provided by the Novasoftware application. Notes must be included where such adjustments may be necessary.

1.3 Delimitaions

Because the Novasoftware schedule data and its formats appears to be at least in part manually inserted by school representatives, possibly with varying conventions, it may not be consistent between schools or classes. Since not every case can be covered here – the process described will be adapted to and only guaranteed to work – on the current schedules from Värmdö Gymnasium. Despite this, tests on schedules from other sources indicate that the process is appliable to most schedules with none or only minor adjustments.

The process has been developed for the current version of the Novasoftware application, at the date of publication of this paper, and may not cover future versions.

Due to the limitations of this project, only one implementation is included. The selected language is JavaScript, written for the Node.js platform. The reason behind this decision is convenience; the language is widespread and simple. JavaScript is also suitable because of its close connection to the web.

1.4 Theory

The Novasoftware application is typically accessed via a web browser (or simply browser), which is a program capable of retrieving and displaying web content. The content is provided by a web server, usually as HTML (Hyper Text Markup Language) to the client (in this case the browser) via HTTP (HyperText Transfer Protocol) over the Internet.[1] A resource provided by a web server may be called an endpoint. In order to automate the process of fetching resources, a different HTTP client than a browser is required. In the implementation, Node.js is used to perform the requests. Node.js may be described in short as an environment for running JavaScript outside of the browser.[2]

1.4.1 HTTP-requests

An HTTP request is a message sent over the Internet from a client to a server; the server should then return a response containing the requested resource back to the client. HTTP requests are constructed of a URL (Uniform Resource Locator), a method, a set of headers and a request body.

The URL acts as an address for the requested resource, comparable to how postal addresses are used for mail. A URL consists of a protocol, a domain, a path and a query string. An example URL is
`http://www.novasoftware.se/ImgGen/schedulegenerator.aspx?format=png`

Here, the protocol is `http`, the domain is `www.novasoftware.se`, the path is `/ImgGen/schedulegenerator.aspx` and the query string is `?format=png`. The protocol denotes the type of request. In this paper only `http` and its secure counterpart `https` will be used. The domain specifies the server, while the path points to the requested resource on it. The query string is optional and may contain extra information about the request in a key-value structure. In the example above; the query string tells the server that the `format` should be `png`.

HTTP requests also include a `method`: only GET and POST will be used in this paper. The different methods differ in multiple ways. Primarily, they include a notion of convention, where GET means that some information is to be fetched and POST means that some data is to be passed to the web server. A POST request may also include data to be passed to the server, known as form data because of the traditional application of using forms to send it. In GET requests further information may instead be included in the URL query string, as specified above.

An HTTP request further includes a set of headers which provide more information about the request, again in a dictionary (key-value) structure. Headers may be divided into request headers and response headers. Request headers are sent from the client to the web server, while response headers are sent back to the client in the server response. The request headers may for example provide the server with information about the client, such as which software is performing the request, while a response header can indicate for example if the requested action was successful or not. This is often indicated in the `Status` field. Another relevant header field is the `Location` response header. If this is provided, along with a 30X (300, 301 etc.) response status, the server is indicating that the resource is rather to be found at the URL provided in value of the `Location` header.[3][4]

Another important feature of HTTP requests are that they can include cookies. Cookies are used by applications in order to save small amounts of data in the browser, and are sent back and forth in every request. This is often used to give the client an ID (Identifier) so that information about the client can persist between sessions even though HTTP is a stateless protocol.[5]

1.4.2 HTML

Because the HTML obtained from the server response is a string, it must be transformed into a searchable and more practical structure. Often the document is described in a DOM (Document Object Model), providing a more programmer friendly interface. In order to reference certain locations or data points of the HTML documents, CSS (Cascading Style Sheets) selectors are used, because of their popularity within the web. The method used in this process, which retrieves HTML and extracts information from it in an automated program, is often referred to as web scraping.[6][7]

1.4.3 JSON

The output of the program is in the JSON (JavaScript Object Notation) format which is human readable and particularly useful for the web because of its similarity to JavasScript; the de facto scripting language on the web.[8]

1.4.4 PNG and PDF

In the process, the PNG (Portable Network Graphics) and PDF (Portable Document Format) file formats are encountered. PNG is an image format, while a PDF is a document format. The information about PDF documents that is relevant to this paper is that they consist of pages, texts and fills (indicating parts of the document filled with color).[9][10]

1.4.5 Regular expressions

Regular expressions are used in multiple instances in the parsing process. A regular expression is a pattern designed to match (indicating if a string belongs to) a particular set of strings. The set is described using symbols, indicating different characters or operators.[11]

1.5 Conventions

Examples will be written in JavaScript. An ES5 (ECMAScript 5) or later environment is assumed. JavaScript code is syntax highlighted as follows:

```
1  var string = 'Hello world!'; //This is a comment
2  console.log(string);
```

Variables, object fields, keywords etc. are displayed in a `monospace font`.

URLs look like the following:
`http://www.example.com/path?author={name}`
The brackets denote variable substitution. In the above expression, the value of the variable `name` is to be substituted after the slash. If for example the value of the `name` variable is the string "john", the resulting URL would be:
`http://www.example.com/path?author=john`

HTTP request are displayed as follows:

```
POST
http://www.example.com/publish
query: {
  type: {type}
  reason: school project
}
form: {
  author: {author}
  experience: null
}
```

The first row indicates the HTTP method, while the second denotes the URL. The `query` dictionary describes the key-value entries of the query string to be included in the URL. The form `form` field exists only on POST requests and specifies the request form data. A `null` value denotes an empty value.
The above description, with the variable `type` set to "paper" and `author` set to "john", should result in a POST request to the URL `http://www.example.com/publish?type=paper&reason=school%20project`, with form data fields equivalent to `experience=&author=john`.

CSS selectors will be written `$(selector)`, for example `$(div)` would select all the `div`s in the document.

Regular expressions will be written as such `/expression/`, for example `/^\d/` would match all strings starting with a digit. Regular expression features and behavior of ES5 or later is assumed.

1.6 Method

The process has been derived via a series of experiments and attempts. The data and different HTTP endpoints have been discovered while attempting to reverse engineer the application by navigating and examining the different pages. Ways of reaching these using a non-browser client have been determined via trial and error.

The parsing method was derived by first studying different schedules; examining how they were organized and how the information could be extracted. From these schedules certain conclusions about the structure of the data could be drawn; using these an algorithm was proposed. Once implemented, this algorithm was tested on multiple schedules and tweaked until all (detected) errors were eliminated. In some cases a solution originally proposed was discovered to be insufficient and a new solution was sought.

The HTTP requests used in the process were found by imitating the web browser's request when using the Novasoftware application. These requests were accessed and viewed using the browser development tools. Because the requests often include redundant information, such as empty header or form data fields, only the relevant information was kept in order to make the requests as simple and elegant as possible. These were determined by the process of experimentation and elimination.

1.7 The Novasoftware application

In order to explain and understand the process, a certain familiarity of the Novasoftware application is required.

The application consists of the following relevant endpoints:

1.7.1 The index page

The index page is the page first encountered by the user.

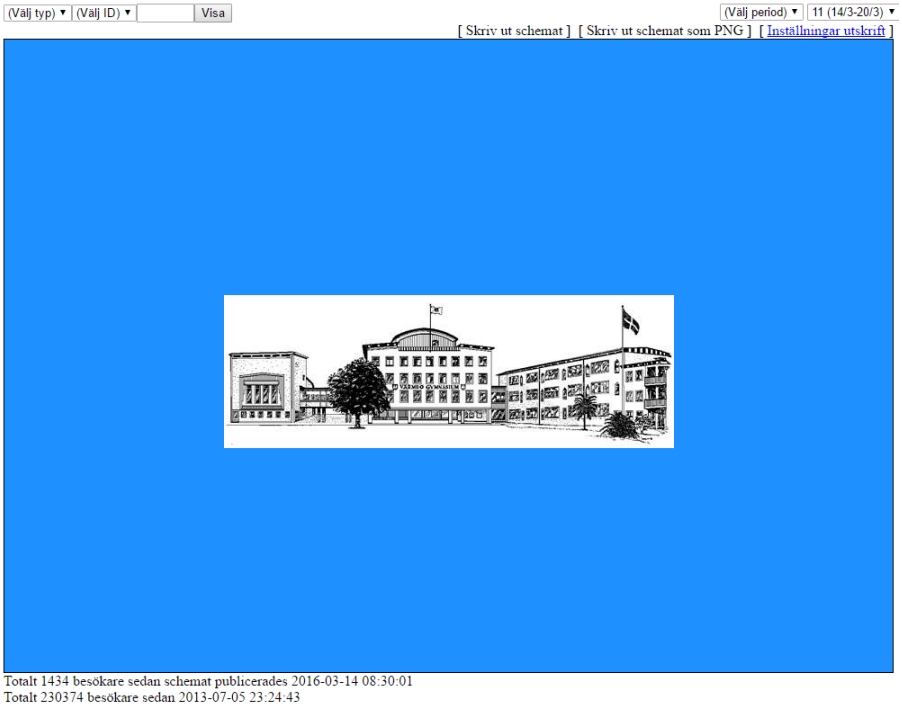

Figure 1

The page URL is

`http://www.novasoftware.se/WebViewer/{novasoftware-id}/MZDesign1.aspx?schoolid={schoolId}&code={code}`

The page is provided as HTML.
In the selects in the top left section of the page, a class may be selected.
In the selects in the top right section of the page, a week may be selected.

Once a class is selected, the schedule page is presented.

1.7.2 The schedule page

The schedule page is the page that the user views in order to see their schedule.

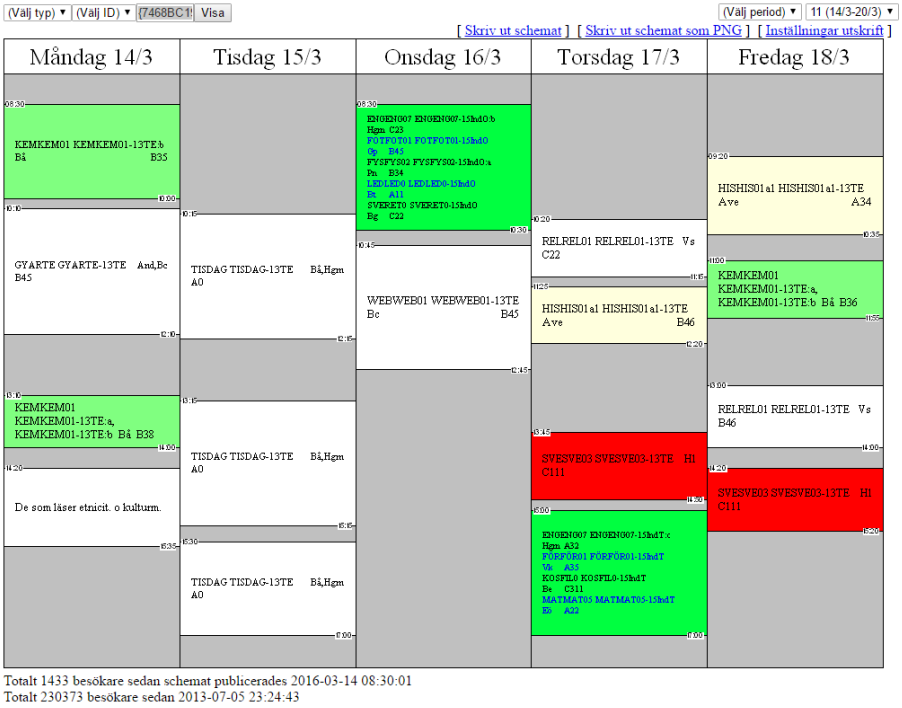

Figure 2

The page is provided as HTML, however the schedule is displayed as a PNG image. Via inspection using the browser development tools, the URL for this image can be obtained, which leads to the image endpoint.

1.7.3 The image endpoint

The URL of the image endpoint is

```
http://www.novasoftware.se/ImgGen/schedulegenerator.aspx?schoolid={schoolid}&
id={class-id}&format={format}
```

By default, the image endpoint returns a PNG of the selected schedule, however if the URL query string parameter `format` is changed from `png` to `pdf`, a PDF file is returned instead.

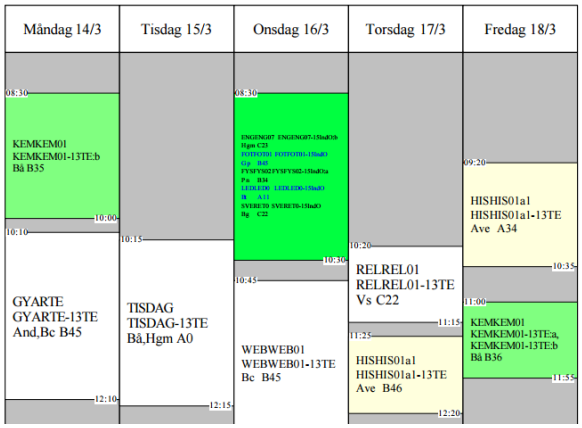

Figure 3

1.7.4 The lesson detail page

Going back to the schedule page, a lesson can be clicked on. This redirects the browser to a detail page, providing additional information about the lesson in the HTML format.

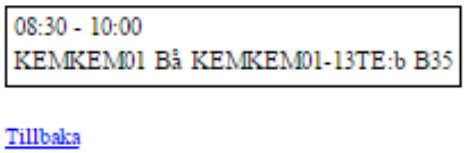

Figure 4

2 Planning

The planning phase of the project involved studying the Novasoftware web application and attempting to find endpoints that could be used. Because no public data API was found, it was determined that scraping was required.

By observing the schedules; the data fields and structure were determined. The schema is included in appendix A. Each schedule is associated with a `className`, a `week` and a list of `days` as well as a list of `lessons`. For each lesson there is a `startTime`, `endTime`, `day` and a `details` list.

The details list items are composed of a `course`, a `location` and a `teacher`. The details are in a list because it was observed that sometimes one lesson item on the schedule may include multiple lessons, as demonstrated below.

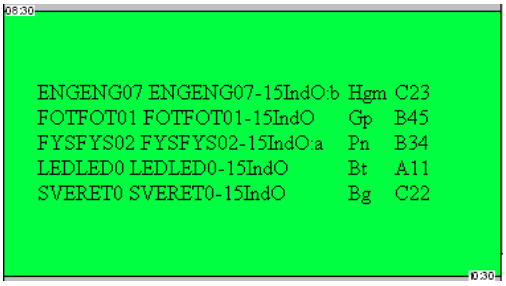

Figure 5

The `detail` list will typically contain only one entry, where there is only one concurrent lesson. Otherwise it will contain one entry per lesson.

3 Development

3.1 Viewing schedules using an HTTP client

Originally there were problems viewing the index page using a non browser HTTP client. The server responded with a redirect response instead of providing the HTML. Upon inspecting the URL it was noticed that it appeared to contain an ID which had previously been assumed to be a part of the path. An example of such an ID is highlighted in the URL below.

`http://www.novasoftware.se/WebViewer/(S(310b0lvc2tbhmb451vdk4c2v))/MZDesign1.aspx`

Because the URL was originally copied from the browser, the ID was copied along with it and also used in the HTTP client. It was discovered that the URL ID varied between sessions when using the browser and the conclusion was made that the ID was invalidated after a certain amount of time. This lead to the problems with the HTTP client: an old and invalidated ID was used. To solve this, it proved useful to leave the ID out of the URL as such:

`http://www.novasoftware.se/WebViewer//MZDesign1.aspx`

The server would then return a redirect response with a new URL containing a valid ID that could be used in future requests. However, performing POST requests to this new URL – essential to the process – would produce a 500 (Server Error) response instead of the requested resource. By examining the browser activity and through the process of experimentation, it was discovered that if a GET request was first sent to the URL, subsequent POST requests would yield the appropriate response. It was as if the ID had to be "activated" before it could be used.

3.2 Parsing the schedule

Next a method of parsing the schedule was sought. The originally discovered representation of the schedule, a PNG-image, was impractical for parsing because images are notoriously hard to decode and extract information from. A different format was sought in its place, which lead to the critical discovery of the PDF version, a format much better suited for information extraction. Because the raw PDF format is hard to read, a parser was required. JSON was determined to be appropriate and a PDF to JSON parsing library was found. The next step was to extract the schedule information from this JSON result.

3.3 Extracting schedule information

The structure of the parsed result is included in appendix B.

By examining the result the conclusion was made that the `Fills` corresponded to lessons and the `Texts` to lesson information as well as to start and end times. The days of the lesson fills and texts could be determined by their x-coordinates. The problem was determining which lesson fill corresponded to which texts.

3.4 Associating lesson fills with lesson texts

The first approach proposed for parsing the schedule was to do so in a linear fashion: one lesson at a time, from top to bottom. The problem with this method was that multiple concurrent lessons were very common. This was especially prevalent in some particular schedules, such as the one below:

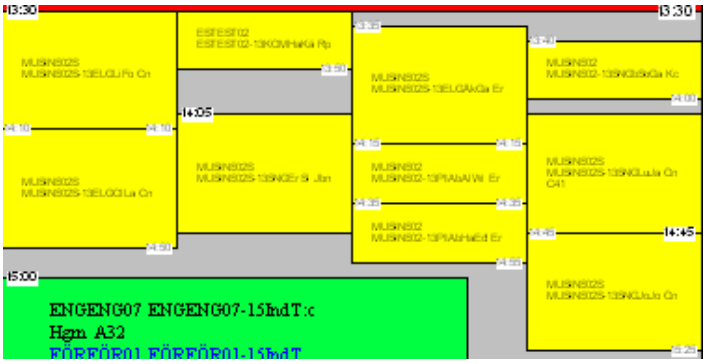

Figure 6

This problem was solved based on the observation that the texts corresponding to a fill always was contained by the correct fill, and no other. Because the fills and texts were described as rectangles (with a corner coordinate and a width and a height), the texts for a particular fill could be found using a simple rectangle intersection test.

Figure 7

Another problem that had to be solved was that start or end times were not always assigned to lessons. Instead, if multiple lessons started at the same time, the time text was only placed on the leftmost lesson. This also applied to end times and can be seen in the above image. The initial solution for the problem was to search among the neighbors of the lessons missing times. However this solution proved unstable and could not successfully parse every schedule. Instead, when thinking about how we humans understand such a schedule, the fairly obvious but very useful observation was made that the times were always along the same horizontal line as the start or end bounds of the lessons. This is how we humans read the schedule; if there is a lesson that for example lacks a start time, we look along the same line until we find a lesson with a start time. Using this method, the missing times could be found by performing rectangle-line intersection tests.

On some schedules, it was noticed that some times are not written out in full, instead only showing the minute (see figure 8). This was however easily solved by finding the closest above time and using its hour to complete the missing information. In some cases, where the next hour had started, the hour had to be incremented by one.

3.5 Determining lesson details

The lesson texts had now been found but a way of extracting the lesson information from them as described in the output format (such as the course) was needed. This was tricky because of inconsistencies between schedules, causing many edge cases. Some lessons for example lacked information about their location or would include extra fields, making it difficult to detect the relevant ones. A solution which was as general as possible, but which would still capture all the cases was sought. Unfortunately methods of detecting a course, a location and a teacher were required, losing some generality as the definitions may differ between schedules. This was primarily done via the use of regular expressions.

3.6 Incomplete lessons

It was quickly discovered that a problem with the PDF version of the schedule was that some lessons lacked text. This was especially prevalent where the lessons were very small and densely packed.

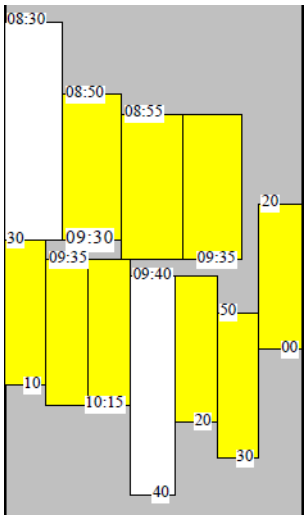

Figure 8

These lessons had to be completed from another source.

3.7 Completing lessons

As described in section 1.7.3, when clicking within the bounds of a lesson on the schedule page, a detail page would load containing the information of that particular lesson. This was exploited in order to fetch information for the incomplete lessons without text since this was available for every lesson.

It was discovered that the request that had to be performed in order to simulate the click contained the x and y coordinates for the click as well as the width and height of the schedule image that was being clicked upon. The next problem to determine these coordinates.

3.8 Determining the click coordinates

A set of coordinates were used in the PDF for the text and fills, but these did not match with the image. Below is a illustration of a image and PDF of the same width and height layered on top of each other. Note the mismatch.

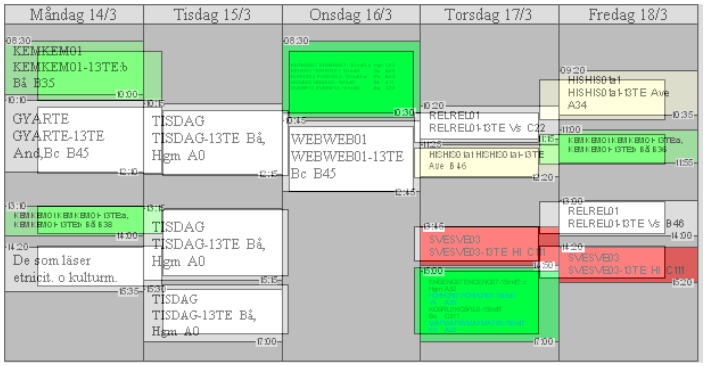

Figure 9

From the above figure one can see that the PDF lessons start further from the edges than the image. The misfit was assumed (which turned out to be correct) to be linear, meaning that the displacement was a linear function of the coordinates. This enabled the use of a simple linear transformation to correct for it, scaling and translating every x and y coordinate by a constant amount. If $P = (x, y)$ denotes the original position and $P' = (x', y')$ denotes the transformed and corrected result the equation for this process may be described in the following way.

$$\begin{aligned} P' &= (s_x * x + t_x, s_y * y + t_y) \\ \implies x' &= s_x * x + t_x, \\ y' &= s_y * y + t_y \end{aligned}$$

The width was also scaled with a constant s_w because the ends did not match up perfectly.

The constants for these transformations was found by the process of experimentation and are included in appendix C. With these constants an almost perfect match was achieved as displayed below.

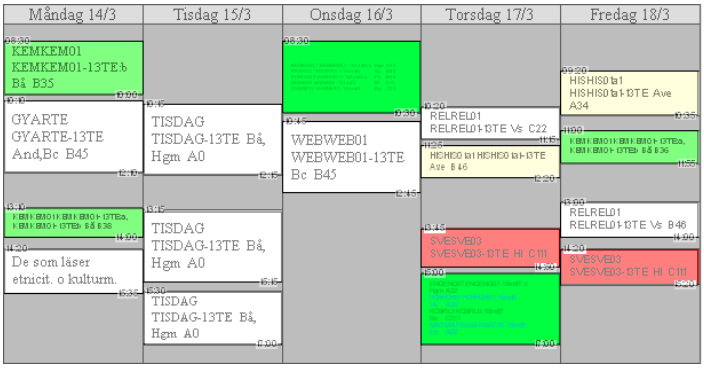

Figure 10

Please note that the included constants only work for PDFs displaying the schedule horizontally, as is the case for the majority of schedules I have encountered, including Värmdö Gymnasium but not for example the Stockholm high school Norra Real. Because the displacement in vertical schedules is also linear, the same method may be used using a new set of constants.

3.9 Performing the click

When performing the click the center coordinates were determined to give the best accuracy; reducing the risk of missing because of calculation error. These were calculated by averaging the corner coordinates.

The click simulation process works as follows: First a request is sent containing the coordinates. The server responds with a redirect request, pointing to the lesson detail page. The new URL obtained in this request is however static and does contain any information about which lesson was pressed. This was a problem when attempting to simulate the process with the HTTP client. After examination it was determined that this information was contained as an ID in the cookie, returned when requesting the schedule and was then expected to follow with the click request. This meant that a schedule first had to be fetched to get this cookie. In order to reduce the overhead of this request a PNG with dimensions 1 x 1 was requested.

3.10 Reading lesson information from HTML

The response of the final click request was the lesson detail page, provided as HTML. The relevant information was relatively easily found because they were contained in a table and could be obtained using standard HTML traversal methods.

3.11 Failing requests

It was noticed that some requests, in a seemingly random pattern, failed (produced no response). In order to continue the process the requests had to be retried and performed again. This applied especially to the click requests. After examining this problem, the cause seemed to be performing too many request too fast and with too little space in between. In order to solve this, all requests were made in sequence and a timeout was performed before making the next request. In order to determine which timeout time was optimal in order to minimize the total time to parse schedules (which is especially important when parsing many schedules) experiments were made. These experiments were performed by completing the process using a certain timeout and measuring the total time elapsed. If the timeout time was too short certain request failed and had to be retried, taking a lot of time. On the other hand, if it was too long, time was wasted. The resulting value was 100 milliseconds. Please note that the optimal time may be dependent on the system or Internet connection and may therefor not be appropriate for every setup.

4 Results

In this section the final process to fetch, parse and extract information from schedules is presented.

In order to perform the requests an ID is required to specify the school from which the schedules is to be fetched for. The ID can be found as the `schoolid` in the query string of the URL when viewing the schedule as a regular user. From the query string, take the `code` also as it is required for some requests. For Värmdö Gymnasium the `schoolid` is `99810` and the `code` is `945537`.

4.1 Obtaining a Novasoftware ID

Next, a Novasoftware ID must be obtained. First perform the following request:

```
GET
http://www.novasoftware.se/WebViewer//MZDesign1.aspx
query: {
  schoolid: {schoolId}
  code: {code}
}
```

The response will contain a `Location` header. Save this in a variable `path` as it contains a URL with an ID. Second, in order to "activate" the page, perform the following request:

```
GET
http://www.novasoftware.se{path}
query: {
  schoolid: {schoolId}
  code: {code}
}
```

4.2 Class IDs and weeks

In order to request a schedule, the `classId` must first be fetched, which may be found by performing the following request

```
POST
http://www.novasoftware.se{path}
query: {
  schoolid: {schoolId}
  code: {code}
}
form: {
  __VIEWSTATE: null
  __EVENTTARGET: TypeDropDownList
  TypeDropDownList: 1
  ScheduleIDDropDownList: 0
  PeriodDropDownList: 8
  WeekDropDownList: {week}
}
```

The server will respond with an HTML document. The list of IDs may be found with the following selector `$(#ScheduleIDDropDownList option)`. The class names are obtained as the text contents of the options elements and the IDs as their HTML `value` attributes.

On this page, the available weeks can also be read as the values of the option elements returned by the selector `$(#WeekDropDownList option)`.

4.3 Fetching the schedule PDF

Now that a classId has been obtained, the schedule PDF may be reached with the following request:

```
GET
http://www.novasoftware.se/ImgGen/schedulegenerator.aspx
query: {
  schoolid: {schoolId}
  id: {classId}
  week: {week}
  format: pdf
  period: null
  width: 1
  height: 1
}
```

4.4 Parsing the schedule PDF

The PDF has to be parsed into JSON or another equivalent structured format. In the parsed result, find the `Fills` and the `Texts` as well as the `Width` and `Height` as these are the relevant fields. The width must be scaled with the constant s_w. In the parsed result, the coordinates origin is in the top left corner, with x and y ascending to the right and down. Sort the texts by ascending y coordinate, with those sharing the same y coordinate sorted by ascending x coordinate.

4.4.1 Determining and preparing the lessons fills

Starting with the fills, we first want to select which fills belong to lessons. Assign a variable for the lesson fills and include all the fills that match the following conditions:

- Non empty, i.e width and height > 0
- Not a title fill, width != 18.391 and height != 2.75
- Not too high, height < 46
- Not too small, area (width x height) > 2

Next the lesson fills must be transformed to match the image. Calculate the corrected x and y coordinates using the formulas $x' = s_x * x + t_x$ and $y' = s_y * y + t_y$. Next the center x and and y are determined by averaging the corner coordinates, while the day is determined by dividing the center x with the total width and multiplying by 5 (the amount of days in a school week). The result must be floored, i.e rounded down to the nearest integer. Now, 0 denotes Monday and 4 denotes Friday etc.

4.4.2 Separating the texts

It was discovered that the text positions does not perfectly match up with the lesson positions. To compensate for this, they must be offset slightly using the following formula $x' = x + t_{tx}$ and $y' = y + t_{ty}$ (see appendix C). Because the texts are too big to fit inside the lessons, set the width and height to w_t and h_t. Next, in order to compare the text positions to the lessons, the above transformation must be applied to them also.

The texts are divided in to three parts; lesson texts, time texts and title texts (which are used in the `days` field, see appendix A).

Because of how the texts are sorted, the title texts will be the first five texts. Assign these to a variable and remove them from the texts array.

Now, each remaining text is either a time or lesson text. They can be separated because time texts follow the same format HH:MM. Some times however are displayed only partial, showing only the minute as seen in figure 8 on page 12. For each text, the texts that match the expression `/^\d{2}:\d{2}$/` are considered a time and the

texts that match `/^\d{2}$/` are considered partial times. The partial times may be completed by finding the closest above time text and taking its hour (the first two digits), appending the minute in the partial text to it. Since a new hour may have started between the partial time and the above time; increment the hour by one if the minute of the partial time is less than the minute of the above time. Now, the rest of the texts are considered lesson texts.

4.4.3 Finding the start and end times of lessons

Next the start and end time are sought for the lessons. For each fill; find all the time texts which bounding rectangle is intersected by the horizontal line of the lesson top bound using a line-rectangle intersection test. You only need to find one such time as all times on this line corresponds to the lesson start. Do the same thing for the end time, but now draw the line at the end of the lesson.

4.4.4 Assigning lesson texts to lesson fills

Now, the lesson texts must be assigned to their corresponding lessons. The texts for each fill are be found by finding all lesson texts which bounding rectangles intersect with the lesson rectangles.

4.4.5 Extracting lesson detail information from lesson texts

Next, the lesson details are to be extracted from the texts. Because of the complexity of this process, it is described in JavaScript code. Despite this it may of course be implemented in any programming language.

First define the functions:

```
1  function isOneOrMore(string, fn) {
2    //Split the string into parts at every comma
3    var parts = string.split(',');
4    //Verify that every part satisfies
5    //the condition specified in fn
6    return parts.filter(fn).length == parts.length;
7  }
8  function isLocation(string) {
9    //Matches for example B35, A11
10   //(conventions may differ between schools)
11   //skrivsalen is included because it is required for Värmdö Gymnasium
12   var regex = /^[a-z]\d{1,3}(-skrivsalen)?$/i;
13   return isOneOrMore(string, function(string) {
14     return regex.test(string);
15   });
16 }
17 function isTeacher(string) {
18   //Matches for example Bts, Bå (conventions may vary)
19   var regex = /^[a-zåäö]{2,5}$/i;
20
```

```
    return isOneOrMore(string, function(string) {
      return regex.test(string);
    });
  }
  function isCourse(string) {
    //Matches for example ENGENG07, TISDAG, GYARTE, MENTORSRÅD
    //(conventions may differ between schools)
    var regex = /^(([a-zääö]{3,}\d{1,2}\w?)|(gyar[a-zääö]{2,3})|(tisdag)
        |(mentorsråd)|([a-z]\d{1,3}(-skrivsalen)?))$/i;
    return regex.test(string);
  }
  function isUnknown(string) {
    return !isLocation(string) && !isTeacher(string) && !isCourse(string)
        ;
  }
```

Next, define the function `parseRows`. Its input is the array of lesson texts, `rows`, and the lesson fill and its output is the lesson details list.

```
function parseRows(rows, lessonFill) {

  if(!rows.length) { return null; }

  //May be any arbitrary string (but must not exist in lesson texts)
  var separator = '|||';
  var rowTexts = rows.map(function(row) {
    return row.text;
  });

  //Divide into parts
  var parts = rowTexts
    .join(separator)
    .replace(/  /g, separator)
    .replace(/ /g, separator)
    .replace(',' + separator, ',')
    .split(separator);

  //Match lessons such as "De som läser etnicitet"
  //containing no other information than the course
  var onlyWords = parts.every(function(string) {
    return /^[a-zääö.]+$/i.test(string);
  });
  if(onlyWords) {
    return [{
      course: rowTexts.join(' '),
      unknowns: null,
      teacher: null,
      location: null
    }];
  }

  var details = [];

  while(parts.length) {
    var course = parts.shift();
    var unknowns = [];
    while(parts[0] && isUnknown(parts[0])) {
```

```
39        unknowns.push(parts.shift());
40      }
41      var teacher = null;
42      if(parts[0] && isTeacher(parts[0])) {
43        teacher = parts.shift();
44      }
45      var location = null;
46      if(parts[0] && isLocation(parts[0])) {
47        location = parts.shift();
48      }
49      details.push({
50        course: course,
51        unknowns: unknowns,
52        teacher: teacher,
53        location: location
54      });
55    }
56
57    return details;
58  }
```

4.5 Fetching the missing information

After the above process is complete some lessons on some schedules lack lesson information. In order to fetch this further HTTP request will have to be performed.

First another Novasfortware ID must be fetched and activated. Perform the request:

```
GET
http://www.novasoftware.se/WebViewer//MZDesign1.aspx
query: {
  schoolid: {schoolId}
  code: {code}
}
```

Again, save the path acquired in the Location header in a variable `path` and perform the activation request. This time the classId must be included.

```
GET
http://www.novasoftware.se{path}
query: {
  schoolid: {schoolId}
  code: {code}
  id: {classId}
}
```

Next we will trick Novasoftware that we are viewing the schedule by loading a PNG of it. We will load it with width 1 and height 1 in order to maximize speed. Perform the request:

```
GET
http://www.novasoftware.se/ImgGen/schedulegenerator.aspx
query: {
  schoolid: {schoolId}
  id: {classId}
```

```
  week: {week}
  format: png
  period: null
  width: 1
  height: 1
}
```

It is important to save the cookie returned by this request as it contains a session that links us to this particular schedule, allowing us to continue the process.

The above process must be done for every week and class. Next we will simulate the click. For every center coordinate of the lessons with missing information (lessons where the `details` acquired from `parseRows` field equal `null`). First perform this request:

```
POST
http://www.novasoftware.se{path}
query: {
  schoolid: {schoolId}
  code: {code}
}
form: {
  __EVENTTARGET: NovaschemWebViewer2
  __EVENTARGUMENT: MapClick|{center x}|{center y}|{width}|{height}
  ScheduleIDDropDownList: 0
  FreeTextBox: {classId}
  PeriodDropDownList: 8
  WeekDropDownList: {week}
}
```

Note that the coordinates and dimensions must be rounded to the nearest integer. After this request has finished and a response has been received, look for the `Location` HTTP header and store it in the `detailPath` variable. Now perform the request:

```
GET
http://www.novasoftware.se{detailPath}
```

As you can see there is no information in this request specifying which schedule or lesson you want to reach, this information is contained in the cookie. Because of this it is important that these requests are made in sequence. The response will be an HTML page containing the information of that particular lesson.

4.5.1 Extracting the lesson details

Again, the process of extracting the lesson details is described using code. The `$` denotes a function that performs a CSS selector query on the HTML document.

```
1  function scrapeClickedLesson($) {
2    var rows = $('tr');
3
4    //The time is usually in the first row
5    var timeElement = rows.shift();
6    var time = getText(timeElement);
```

```
7
8       //However if it starts with block
9       //we should instead take the next row
10      if(/^Block:/.test(time)) {
11        //Take the next row
12        time = getText(rows.shift());
13      }
14
15      var times = time.split(' - ');
16      var startTime = times[0];
17      var endTime = times[1];
18
19      //There is always one redundant row in the end
20      rows.pop();
21
22      var details = rows.map(function(element) {
23        var columns = element.children().map(getText);
24        var course = columns[0];
25        var teacher = columns[1];
26        var unknown = columns[2];
27        var location = columns[3];
28
29        return {
30          course: course,
31          teacher: teacher,
32          unknown: unknown,
33          location: location
34        };
35      });
36
37      return {
38        startTime: startTime,
39        endTime: endTime,
40        details: details
41      };
42    }
43    function getText(element) {
44      return element.text();
45    }
```

5 Summary

The development of the process of extracting schedule information from the Novasoftware web application has been presented, discussing encountered problems and how they were solved.

Two major problems which presented them selves during the development of the process were to find an endpoint from where information could be extracted, as well as deciding the extraction process. This was solved by discovering the PDF image endpoint and then, through examining and experimenting with the obtained results, deriving a parsing method. The next major problem was retrieving the information that was missing from certain lessons. This was solved by finding the lesson detail page and then determining the process to simulate the click as well as to extract information from

the HTML result. Failing requests were managed by retrying them and performing all requests in sequence with a timeout in between. Remarks about limitations of the described method were made, in some cases presenting methods of extension.

Instructions for implementation of the process was provided using text and code; first describing methods of retrieving required data via HTTP requests and subsequently documenting methods of extracting relevant information. The process may be implemented in any language and platform supporting the required technologies.

6 Discussion

The data obtained using this process may be used for many applications. One example is to create a custom schedule viewer with the purpose of for example increasing the availability of the information or providing a mobile friendly experience.

If the data is used for a schedule viewer, new problems will have to be solved. For example the decision of when to fetch and parse the schedules has to be made. Fetching and parsing a particular schedule every time it is requested may be both slow and resource intensive. Instead a caching system, performing the parsing only once for each schedule and saving the result, may be required. This further poses the question as of how many schedules should be kept in the cache as well as when they should be considered invalid, since the schedules change every term (and has been observed to sometimes even change during the term).

The data may also be used for research purposes. For example one could analyze different schedules and present information about properties such as day lengths and starts, or drawing meaningful conclusions about the school experience.

6.1 Implementation

The implementation is hosted on the code sharing website GitHub.com at the following URL: `https://github.com/johnrapp/novasoftware-schedule-parser`. You are free to download, modify and use the source code of this project without asking for permission.

6.1.1 Installation

The project requires Node.js on the running machine. Instructions for installing Node.js are found at the Node.js website `https://nodejs.org/en/download/`.

Because the project uses external modules, these dependencies has to be installed using npm (Node Package Manager), included in Node.js.

To do this, run the following command in the terminal:

```
$ npm install .
```

6.1.2 Usage

To run the program, open a terminal window and run the command
`$ node .`
This will fetch a selected set of schedules from Värmdö Gymnasium and save them in the `schedules` directory.

6.1.3 GitHub readme

More information about the provided implementation, as well as more thorough instructions may be found in the GitHub readme at `https://github.com/johnrapp/novasoftware-schedule-parser#readme`.

6.2 Considerations

Anyone who plans to implement this method should consider that the Novasoftware may get overloaded if it is receiving too many concurrent requests. If not considered, the users activity will be inseparable from a DOS (Denial of Service) attack, which might damage the servers or the business. This will also limit regular users ability to use the viewer as it may slow down or restrict the availability of the application. In order to minimize the risk of slowing down or damaging the servers, one should definitely consider not sending more than one concurrent request, preferably with a timeout between each request. This is of course included in the provided implementation. It is also considerate to perform the fetching during non peak hours, such as in the night.

sluta på en annan note

Appendix A The schedule format

```
{
  "className": "13TE",
  "week": 15,
  "days": [
    "Mandag 22/2",
    "Tisdag 23/2",
    "Onsdag 24/2",
    "Torsdag 25/2",
    "Fredag 26/2",
  ],
  "lessons": [
    {
      "startTime": "08:30",
      "endTime": "10:00",
      "day": 0,
      "details": [
        {
          "course": "SVESVE02",
          "location": "C22",
          "teacher": "Bts",
        },
        ...
      ]
    },
    ...
  ]
}
```

Appendix B The parsed PDF


```
{
  "Width": 102.266,
  "Pages": [
    {
      "Height": 52.563,
      "Fills": [
        {
          "x": 4.813,
          "y": 1.757,
          "w": 42.858,
          "h": 9.452
        },
        ...
      ],
      "Texts": [
        {
          "x": 24.156,
          "y": 18.938,
          "w": 21.461,
          "h": 4.216,
          "R": [
            {
              "T": "TISDAG"
            }
          ]
        },
        {
          "x": 21.759,
          "y": 12.966,
          "w": 18.391,
          "h": 2.751,
          "R": [
            {
              "T": "08:39"
            }
          ]
        },
        ...
      ]
    }
  ]
}
```

Appendix C Constants

$$\begin{cases} s_w = 0.995 \\ s_h = 1 \\ s_x = 1.1 \\ s_y = 1.07 \\ t_x = -30.5 \\ t_y = -12 \\ t_{tx} = 0.7 \\ t_{ty} = 0.42 \\ w_t = 1.5 \\ h_t = 0.5 \end{cases}$$

Sources

Behövs inte källor, Wikipedia är bara referens typ av arete som inte gått ut på källforskning

All sources were reviewed at the date of publication of this paper. Because the information stated in the theory section of this paper (which is were the sources are referred from) is recognized as basic knowledge in the field, Wikipedia is considered a reliable source.

[1] Web browser - Wikipedia
`https://en.wikipedia.org/wiki/Web_browser`

[2] Node.js - Wikipedia
`https://en.wikipedia.org/wiki/Node.js`

[3] Hypertext Transfer Protocol - Wikipedia
`https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol`

[4] Uniform Resource Locator - Wikipedia
`https://sv.wikipedia.org/wiki/Uniform_Resource_Locator`

[5] HTTP cookie - Wikipedia
`https://en.wikipedia.org/wiki/HTTP_cookie`

[6] Cascading Style Sheets - Wikipedia
`https://en.wikipedia.org/wiki/Cascading_Style_Sheets`

[7] Web scraping - Wikipedia
`https://en.wikipedia.org/wiki/Web_scraping`

[8] JSON - Wikipedia
`https://en.wikipedia.org/wiki/JSON`

[9] Portable Network Graphics - Wikipedia
`https://en.wikipedia.org/wiki/Portable_Network_Graphics`

[10] Portable Document Format - Wikipedia
`https://en.wikipedia.org/wiki/Portable_Document_Format`

[11] Regular expression - Wikipedia
`https://en.wikipedia.org/wiki/Regular_expression`