API Documentation

*Super Seminar Scraper*

Batu Inal

*bi49@cornell.edu*

M.Eng Project

May 14, 2016

Table of Contents

Introduction 2

Previous Work 3

Current Work 3

Outline of the System 3

High-Level Flow 3

Super Seminar Scraper API Flow 4

School Executors 5

Output Files 8

Classes and Libraries 9

How to Use the System 10

System Requirements 10

Running Locally 11

Setting Up and Running on AWS 11

Cron Job Setup 12

How to Add a New School to the System 12

Statefulness and Search 13

Storage – Persistency 13

Search Based on Tags 14

Recommended Future Work 14

References 16

# Introduction

Every major university runs seminars and colloquia, and data about speakers, titles, and abstracts is often posted online. The topics and speakers often say a great deal both about the venue and about what is hot in the field in general; but while the information is available "in the wild", it is not in a form that is amenable to machine parsing. This project will address the issue of scraping colloquium information from top departments into a consistent, machine-readable format in order to support data mining.

## Previous Work

This is a project that has been carried out, under the supervision of David Bindel, by four students up-to-date; Jiankun Lu, Jing Jing, Patrick Chen and most currently Batu Inal . The contributions from the previous three students range in different ways; thus it is one of the objectives of the project to connect these pieces together to create a unified functioning system.

The only component used in the current project, from the previous three projects, is the data-driven website built by Jiankun Lu, using JavaScript and HTML. It handles the data parsing by Angular.js and animations by jQuery. *The newly created Super Seminar Scraper (SSS) API feeds the front end through a JSON file.*

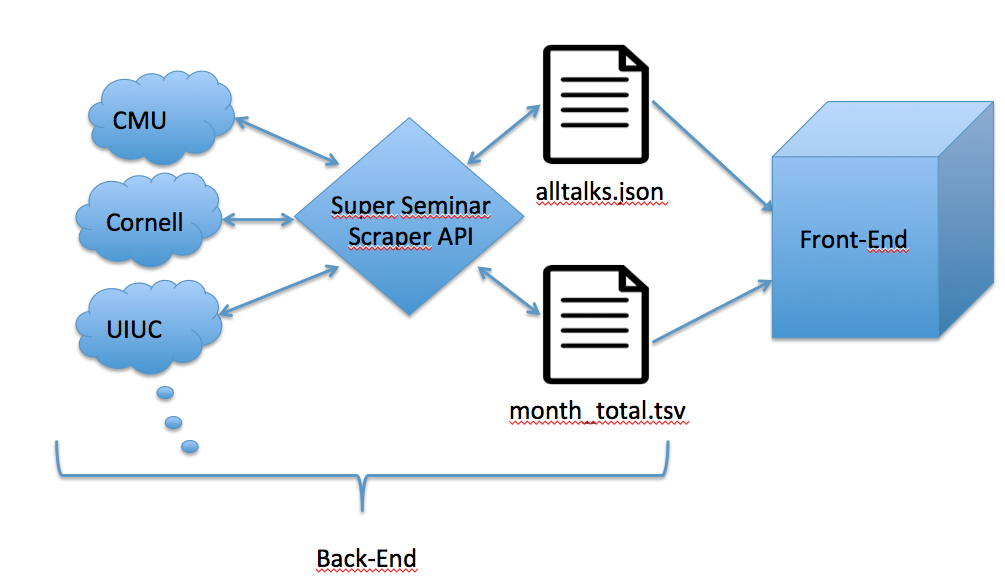
## Current Work

Since we wanted to create a general framework to scrape CS colloquia. The API has been written to scrape information from Cornell, Berkeley, CMU, UW, and UIUC. We believe this is a diverse enough set of schools (data formats) to give us some feel for what would have to go into a more general framework. The scraped data is then fed into an existing web front-end in order to get a complete web application that displays and queries these colloquia in a consistent manner. This documentation serves to describe the solely SSS API, not the front-end source code.

# Outline of the System

## High-Level Flow

Back-end’s role is to create the data to be fed into the front-end. In order to do so, our SSS(Super Seminar Scraper) API crawls the colloquia sites of various universities, then process the raw HTML’s, to eventually create two files; my\_SSS\_all\_records.json and month\_total.tsv. Both the files are stored under the “data” folder and are utilized by the front-end to generate the content of our website.



*Figure 1: Represents the high-level structure of the System*

## Super Seminar Scraper API Flow

The flow of the main module, located in the file main.py, is fairly simple. The steps can be summarized as the following;

1. Create an SSS(Super Seminar Scraper) object (which essentially holds a dictionary of schools)
2. Create School Objects (which hold list of colloquia objects and their respective metadata) to be fed into our SSS object.
3. Call the scrape\_data() function from the SSS object to utilize the Scrapy Library. Scrapy crawls all the colloquia websites provided in the School objects created. It finally saves the raw html of each page under the metadata of the correct School object.
4. For each School a custom “executor” is called whose main job is to parse the raw html and data-mine all the relevant information in the page. It is important to note that even though each school has its own “executor” they all utilize the same API to get the job done. (In latter parts of the documentation the process is explained in more depth.)
5. Finally the save() method from the SSS object is called to save our data by serializing into a .json file

## *Untitled:Users:batuinal1:Desktop:Screen Shot 2016-05-10 at 8.58.51 AM.png*

*Figure 2: Represents the connections between the files in SSS API, numbered edges are described above*

## School Executors

Even though we argue that this is a general framework to data-mine colloquia sites the programmer must first manually examine the source html of the site. The API was written such that general themes/patterns were examined in HTML’s of colloquia sites. Thus the functions in the API are general enough that they can be used on most colloquia sites to extract the metadata that the programmer wishes. Though the programmer may need to accommodate the needs on some parts of extraction, of a particular site, if it differs widely from other colloquia sites in common themes.

The executor, custom parser and information extractor, for each school is stored under the school’s name in the “schools” folder. The executors are imported into the main and called once the School objects are created and the scrape\_data() method is called on them. This extracts the raw HTML of the school to be stored under the respective School Object. The raw HTML is used by the school executor to parse and extract information.

Note that a school executor can be in two forms, either to parse and extract information from a raw html or from the .ical file, located under a link in the extracted raw HTML. Ical files are easier to work with as they are well formatted. It is up to the programmer to determine if an Ical file of the colloquia exists or not by visiting the site.

*The steps of a school executor that solely uses HTML can be summarized as the following:*

1. Get the school object from the SSS Object
2. Call extract\_content() method to extract an initial list of contents from the xpath provided to method. The xpath is manually inspected from the page’s source.
3. Call filter\_content() method to filter the list that was extracted by the extract\_content() method. You feed a list of the strings you would like the initially extracted list to be filtered based on. You can either create the list yourself and feed it or use the filters used in the shared.py module.
4. We have realized that from the list obtained we can extract every colloquia by first going through the whole list element by element, retrieving the date in the element (if it exists) and creating a colloquia entry for it (in the correct school’s colloquia list). Looking for dates as a key for distinguishing between a colloquia and non-colloquia entry lets us create a correct sized in-order array of colloquia objects with their date metadata set. The process can be done by calling the retrieve\_dates() method.
5. Call the retrieve\_metadata() method, on the list obtained from the previous step, multiple times to extract metadata such as topic, speaker and url of each colloquia on the colloquia-site.
6. Once the speaker, topic and url has been set for each of the colloquia; we set a global venue and official school name for all the colloquia’s within a school object. We chose to use this design pattern for venue and school’s official name as they’re usually stated in a single non-explicit place in the html and are hard to find. Nevertheless, school’s usually have a single venue dedicated to conduct these colloquia throughout the year.

*The steps of a school executor that uses .ical files can be summarized as the following:*

1. Get the school object from the SSS Object
2. Call extract\_content() method to extract a list of .ical paths from the xpath provided to method. The xpath for the .ical url is manually inspected from the page’s source.
3. Call extract\_ical\_content() method on the correct element(URL) of the list created from the previous step. The method opens the .ical file and stores its content into the right School’s metadata.
4. This step is fairly easy but requires the manual investigation of the .ical file. However the steps are pretty much similar across most .ical files so the shared.py library is highly utilized and eases the process significantly. The high-level explanation of the step is to; walk through all the VEVENTs (entry for a presentation, possibly a colloquia, with its metadata in the .ical file) and easily use the icalendar.parser to retrieve all the colloquia with their metadata.
5. (Optional) The last step is to set a global venue and(or) official school name for all the colloquia’s within a school object.

Once the school executor for a particular school is run; the list of colloquia, along with their metadata, is stored in the School Object within the SSS Object.

In order to write your own school executor please refer to previously written examples of school executors under the “schools” folder.

## Output Files

The system outputs two files called <name\_of\_SSS>\_all\_records.json (currently my\_SS\_all\_records.json) and month\_total.tsv under the folder data. The front-end The format of the two files can be summarized as the following:

* <name\_of\_SSS>\_all\_records.json is a json file which is created by going through all the School Objects in the SSS Object and creating colloquia entries in a json file. The file is used to generate the records in the main page of the front-end. Below is an example of the <name\_of\_SSS>\_all\_records.jsonfile:

records": [

{

"Topic" : "Program Analysis and Transformation for Scientific Computing",

"Speaker" : "Paul Hovland",

"Time" : "2015-10-07 09:30",

"Venue" : "Gould-Simpson 701",

"University" : "University of Arizona",

"URL":"https://www.cs.arizona.edu/news/events/evdetail.html?ID=681",

"Description" : "We discuss...",

"Tags" : ["Scientific Computing", "University of Arizona", …]

}, … ]

* month\_total.tsv is a tab separated values file which is created by keeping track of the count of colloquia in each month of the current year. The file is used to generate the donut graph on the front-end to depict the distribution of the colloquias with respect to date. Below is an example of the month\_total.tsv file:

month count

2016-2 4

2016-3 12

2016-4 24

2016-5 27

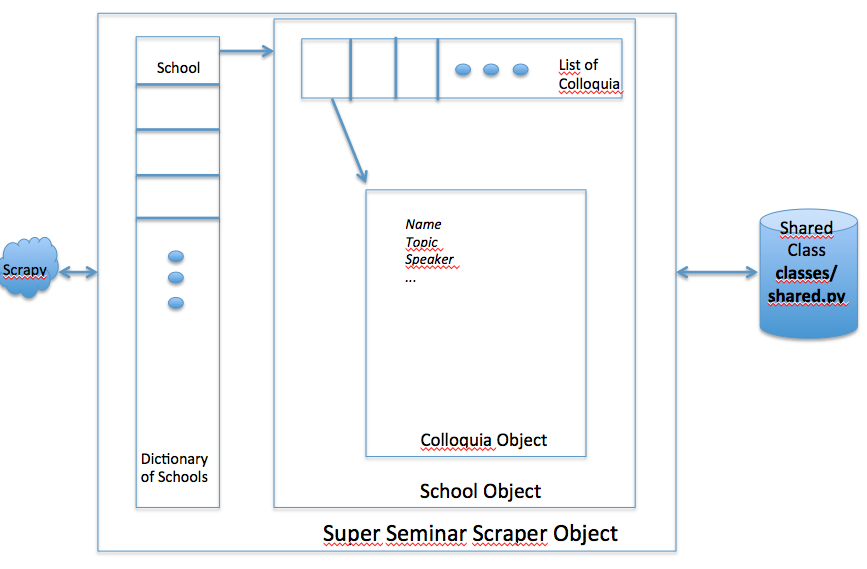
2016-6 24

# Classes and Libraries

In order to utilize the API it is essential to understand the structure of how the classes and libraries are linked with one another to form a unified system. The overall structure is that, in the main we create a SSS (Super Seminar Scraper) Object, which holds a dictionary of School Objects that hold a list of Colloquia Objects. This structure allows the code to store data in a much more elegant, organized and robust than other possible solutions.

The library named “shared.py” is where all of the common functionality is implemented. It is a library made up of functions written with respect to common themes and patterns across colloquia websites. Thus, all of our run-time created objects communicate with the shared library. It is very important to understand the functions within the shared class for writing your own school executor class as well.

Due to time limitations we have unfortunately not been able to document the classes and libraries. In order to understand the functions and their usage please refer to the comments on the source on code.



*Figure 3: Represents the class structure of the system. All classes, including the school executors, utilize the Shared class library. The Scrapy module is only called by the SSS object.*

# How to Use the System

## System Requirements

In order to run the project please have the following libraries installed on your system:

1. The latest Python
2. Scrapy Framework4
3. iCalendar package5
4. (Optional) AWS Command Line Interface6

## Running Locally

The source code for the project is located under a Github repository1. It is a public library and can be simply cloned to your local machine.

In order to run the system locally, from your terminal cd into the talkstream/courseWebPageCrawler directory and execute the command python main.py. This will run the project, which will crawl the colloquia sites and create files in the talkstream/courseWebPageCrawler/data folder. Please refer to the requirements of the system in before trying to execute the project.

To run the front-end locally please refer to the documentations of Jiankun Liu.

## Setting Up and Running on AWS

As the cost and maintenance of an Amazon Web Services Application is fairly insignificant, we recommend that you setup the project on an AWS EC2 instance. As the process requires significant amount of tedious work to setup, we have created scripts to ease the process into as simple as changing a few parameters and running the scripts. The steps to run the script are as follows:

1. Setup an AWS account.
2. Create a Access Key ID and Secret Access Key for your AWS account2.
3. Create an Amazon EC2 Key Pair3
4. From your terminal, cd into the talkstream/serverBot directory. Open up the files install.sh and launch.sh, replace all the $XYZ tags with your credentials from steps 2 and 3.
5. Simply run the script launch.sh

Once the script is run a single EC2 instance running the code on the github Repo is created. To find the DNS of the page view your AWS EC2 console.

Note that since the system is setup on AWS we have configured our script to download and setup the necessary libraries in the EC2 environment such that it meets all the requirements of the project.

## Cron Job Setup

As the SSS needs to be run from time to time to keep up to date we recommend setting up simple Cron Jobs. The Cron Job is as simple as executing the command “python main.py” in the talkstream/courrseWebPageCrawler folder in a time interval you have determined. Configuring Cron Jobs either locally or on AWS is a fairly simple task and should not take more than half an hour for the programmer to setup. Please consult the web for setting up cron jobs.

# How to Add a New School to the System

Follow the steps below to add a new school to the system:

1. In the beginning of the main.py file define the URL of the new school’s colloquia website.
2. In the main.py file create a new School Object for the school and add it to the defined SSS object
3. Add a new <school\_name>.py file to schools folder and write a custom executor of the form *<school\_name>\_executor* (see details for School Executors above)
4. In the main.py file import the new custom executor of form *“from courseWebPageCrawler.schools.<school\_name> import <school\_name>\_executor”* and call the imported function by *<school\_name>\_executor(my\_SSS)*
5. For the new school to be updated into the system, either; Re-run the code by “*python main.py”* or wait for your cron job to take place.

# Statefulness and Search

## Storage – Persistency

In order for the system to be able to maintain a reliable and persistent state we faced two challenges and solved them in the following way:

1. The size of the data was not significant enough to be kept in a database.
   1. Our solution was to simply store the data within a JSON file (refer to Output Files for format of the file).
2. Since we wanted to keep the information from past years in our JSON file, but did not want to have duplicate colloquia in the case that; the same colloquia entry existed on the web page in different runs of the SSS.
   1. The solution was to first check the data folder if <name\_of\_SSS>\_all\_records.jsonexists. If it does, de-serialize it. While de-serializing, create a map with a key generated from each colloquium’s speaker date pair. We assume that the same person cannot be giving multiple talks on the same day.
   2. While serializing the newly scraped data, we first add the initially de-serialized list of colloquia from the previously generated version of <name\_of\_SSS>\_all\_records.json. Then while adding the colloquia information scraped from the current up-to-date page, we check whether the colloquia already exists or not in our hash map. If it doesn’t we add it, otherwise we drop it.
   3. Once the new state of the system is established <name\_of\_SSS>\_all\_records.json is overwritten by the updated data.

## Search Based on Tags

The search in the front-end’s main page is generated by the tags given to each colloquia in the JSON file. We assign tags for every colloquia based on the metadata that is inserted in the colloquia object. Tags, search parameters, for a colloquia is based on the speaker’s name, the words in the title (topic), university where the colloquia is held and the date of the colloquia. Thus “Tags” is a list of key words (tokens) that is appended to every time a new metadata is set in the colloquia object. (For the logic of creating Tag Tokens refer to the set\_metadata() function of the Colloquia Class located in classes/colloquia.py file.

# Recommended Future Work

Below are some recommendations that a future MEng Student can perform, to improve the system:

1. One of the contributors, Jing Jing has created an API-like Python module that utilizes Machine-Learning techniques such as k-means clustering, Logistic Regression with Gradient Descent and Latent Dirichlet Allocation to identify the latent structures within a list of seminars or colloquiums through document clustering and topic modeling. The integration of the library would differentiate colloquiums from seminars, cluster seminars into different groups based on their subareas and extract the common topic words among colloquiums or seminars with the same subarea.

Since tags are already defined for every colloquium it should be fairly easy to integrate the Machine Learning Library to the created SSS API or even utilize the sckit-learn library7 This would enable the backend to generate all the files required to generate the cool graphs in the front-end.

Since this is a trivial task the programmer may wish to develop the backend in the field of ML. For instance writing algorithms to predict future colloquia topics, providing the user with how highly rated a professor is likely to be based on his/her current colloquia area and so forth. Basically any idea that incorporates making sense of the data in a way that leads to useful information.

1. As it was also one of the initial goals of the project, more schools can be added to the system by writing custom school executors. According to new themes/patterns from the new schools, the shared library can be made more robust.

In fact if time had permitted it was our goal to divide shared.py into finer grained set of classes. Having the library divided up into smaller parts would allow for future development to be more flexible.

# References

1. *Inal, B., & Lu, J. (n.d.). Jl-git/talkstream. Retrieved May 11, 2016, from https://github.com/jl-git/talkstream*
2. *Getting Your Access Key ID and Secret Access Key. (n.d.). Retrieved May 11, 2016, from http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSGettingStartedGuide/AWSCredentials.html*
3. *Amazon EC2 Key Pairs. (n.d.). Retrieved May 11, 2016, from* [*http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html*](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html)

1. *Installation guide¶. (n.d.). Retrieved May 11, 2016, from http://doc.scrapy.org/en/latest/intro/install.html*
2. *Icalendar 3.9.2 : Python Package Index. (n.d.). Retrieved May 11, 2016, from https://pypi.python.org/pypi/icalendar*
3. *Installing the AWS Command Line Interface. (n.d.). Retrieved May 11, 2016, from http://docs.aws.amazon.com/cli/latest/userguide/installing.html*
4. *Scikit-learn. (n.d.). Retrieved May 11, 2016, from http://scikit-learn.org/stable/*