TADAPI

Design Specifications v0.1
March 7th, 2013

Novacoast
Nibble

Disclaimer

Novacoast™, Inc. makes no representations or warranties with respect to the contents or use of this document, and specifically disclaims any expressed or implied warranties of merchantability or fitness for any particular purpose.

Trademarks

The Novacoast name and logo are registered trademarks of Novacoast, Inc. in the United States and other countries. The Novacoast Symbol is a trademark of Novacoast, Inc.

All third-party trademarks are property of their respective owner.

Copyright

Copyright © 2010 Novacoast, Inc. All rights reserved.

Change Control Process

The Change Control Process governs changes to the scope of this project throughout the project's duration. It applies to new components and to enhancements of existing components. A written Change Request communicates any desired changes to this project. It describes the proposed change, the reason for the change, and the effect the change might have on the project. The Novacoast project manager supplies the appropriate Change Management documents.

Both Novacoast and the customer review the Change Request and approve or reject it. Both parties must sign the approval portion of the Change Request to authorize the implementation of any change that affects the project’s scope, schedule, or fee.

Document Change Tracking

Contributors: Andrew Duong, Tong Wu, Jimmy Le, Mark Oparowski, Eron Howard, Renato Untalan, David Parker

Table of Contents

1. Introduction

1.1 Product Summary

TADAPI (The Automated Discovery of Application Programming Interfaces) is a simple to use and intuitive tool to discover API functionality. TADAPI works by sniffing traffic over the network as an API is being navigated and captures the API requests as well as the server responses in order to produce deliverable documentation as well as other services.

1.2 Definitions, Acronyms, Abbreviations

API: Application Programming Interface
PCAP: Packet Capture
TADAPI: The name of the tool being specified in this design specification document

1.3 References

LibPCAP: http://www.tcpdump.org/
CouchDB: http://couchdb.apache.org/
Ruby: http://www.ruby-lang.org/en/
RubyOnRails: http://rubyonrails.org/

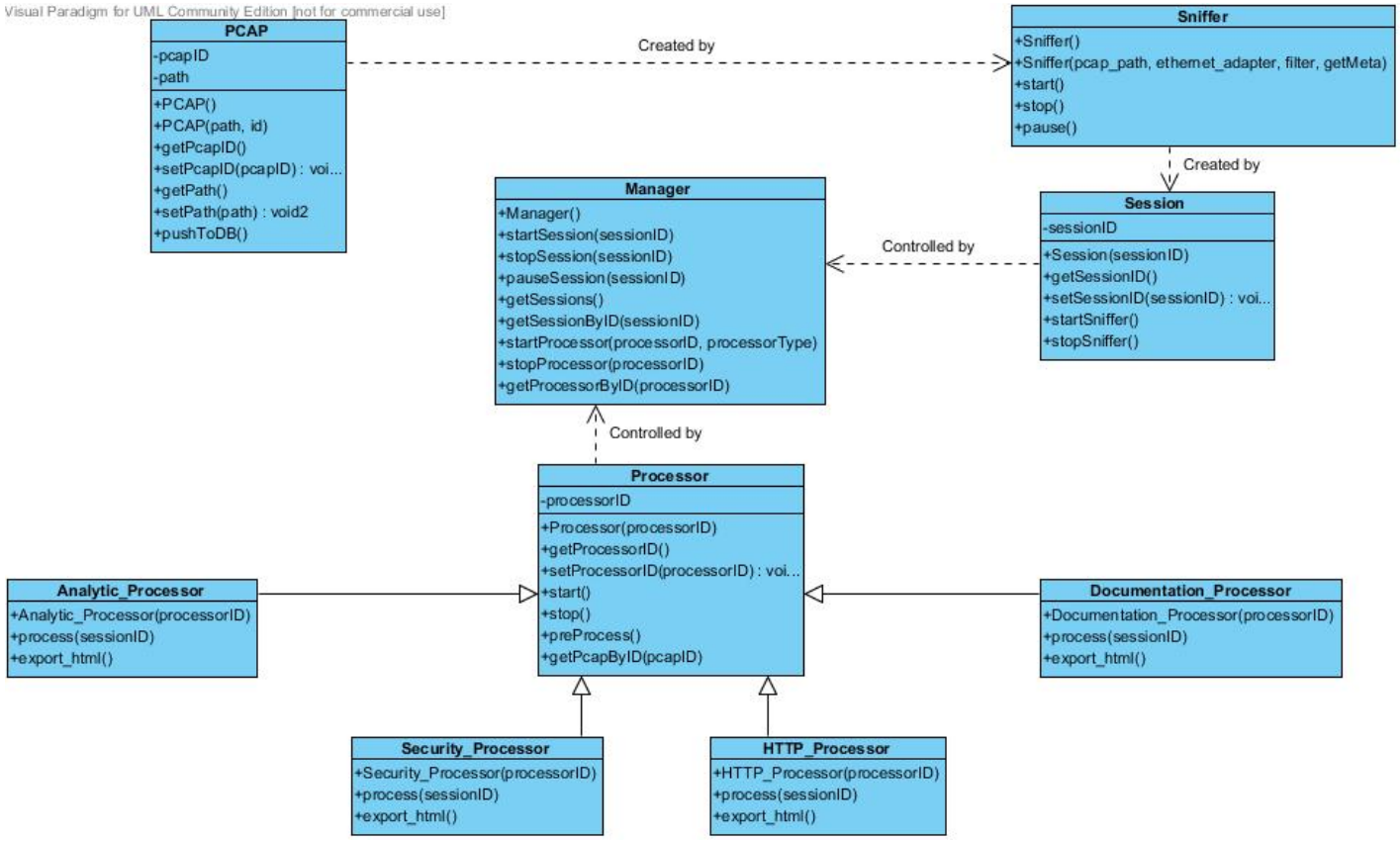
1.4 Document Overview

This document provides detailed descriptions for the design specifications of TADAPI, which includes the design of the front end website, backend, as well as database. Each class will be detailed with their protocols and functionality which will be associated with their respective UML, dataflow or sequence diagrams.

2. Design Specification

2.1 UML

The following illustrates the proposed layout of TADAPI in terms of its classes, functions, member variables, and inheritance models.



2.2 Class Specifications

2.2.1 Manager

The Manager class acts as the controller for the entire application. It is controlled by the user using the play and stop buttons which correspond to the startSession(sessionID) and stopSession(sessionID). The Manager creates a Sniffer object which returns a PCAP file that the Manager sends to the processor object for processing.

2.2.1.1 startSession(sessionID)

Start a session with a specified sessionID. This creates a new sniffer and link to a PCAP file. This function takes a sessionID as its input.

2.2.1.2 stopSesson(sessionID)

Stops a session with a specified sessionID. This deletes the Sniffer object created by startSession(sessionID) by calling the stop function in the sniffer class. This function takes a sessionID as its input and returns the PCAP file.

2.2.1.3 pauseSession(sessionID)

Pause the session with specific sessionID. This functions takes a sessionID as an input and then calls the pause function in the sniffer causing all traffic sniffing to stop temporarily by calling the pause function in the sniffer class. The pauseSession function allows the user to temporarily stop the sniffer so that packets are not captured while the user does other tasks not related to API discovery(email, web surfing, etc.).

2.2.1.4 getSessions()

Returns a list of sessions created by the Manager. This outputs a list of sessions created by the Manager when the user executed the application using the play button. The getSessions function allows the user to see which session's API they want to process.

2.2.1.5 getSessionByID(sessionID)

Gain access to a specific session. This function takes the sessionID as an input and results in a reference to the session specified. This function will allow for management of multiple sniffing tabs and captures concurrently.

2.2.1.6 startProcessor(processorID, processorType)

Creates and starts a processor. This function will take a unique processorID and a processorType as input and depending on the processor type, the function will create a specific "*_Processor" where * is determined by the processorType variable by calling *_Processor in the *_Processor class .

2.2.1.7 stopProcessor(processorID)

Stops the processor and deletes the processor object. The stopProcessor function takes in a processorID parameter.

2.2.1.8 getProcessorByID(processorID)

Returns the processorID. This is used in the event that multiple processors are used to processes multiple parsed PCAP files.

2.2.2 PCAP

The PCAP class holds all the information for the PCAP file created by the sniffer object. The PCAP file contains a unique ID set by the setPcapID(pcapID) function and a path to the PCAP file that is set by the setPath(path) function.

2.2.2.1 getPcapID()

Returns the value for the PCAP object. This function returns the specific PCAP object's ID so that the processor knows which PCAP

2.2.2.2 setPcapID(pcapID)

Sets the specific PCAP to the value given by the pcapID parameter. Having multiple pcapID's and the ability to set them individually allows for multiple sessions to run concurrently.

2.2.2.3 getPath()

Returns the relative path to the PCAP. This function allows for the user to link to a previously captured PCAP. This also allows for TADAPI to store PCAP captured locally before analyzing the data.

2.2.2.4 setPath(path)

Set the path to the PCAP file to use in the processor and sniffer. This takes a path string as input and sets that path as the location of the local PCAP file.

2.2.2.5 pushToDB()

Pushes path to the PCAP file on disk into the database to be accessed by the manager or processor.

2.2.3 Sniffer

The sniffer is the content capturing service in the application. The sniffer takes the incoming network traffic and pushes that data into a PCAP file. The sniffer also acts as a parsing class in which it takes the captured data and formats the data into a json output file. The Sniffer creates output to be used by the Processor.

2.2.3.1 start()

Starts sniffing packets using LIBPCAP and pushes the network traffic sniffed into a PCAP file.

2.2.3.2 stop()

Stops sniffing packets using LIBPCAP.

2.2.3.3 pause()

Pauses the sniffer and halts traffic from being sniffed. This is primarily useful if the user plans to do other tasks which require network access, but does not wish for the traffic to be picked up by the sniffer.

2.2.4 Processor

This is the base class for the function-specific processors. This class includes all the functionalities and member variables that are absolutely required by all processors regardless of their functionality. For function-specific processors, refer to section 2.2.5.

2.2.4.1 getProcessorID()

Get the unique processor ID that is assigned to a specific processor. This allows for easier management of processors when there are multiple sniffing instances happening.

2.2.4.2 setProcessorID(processorID)

Set the unique processorID for a processor by taking the processorID as the input. Having the ability to set the processorID opens up the possibility for multiple processors running simultaneously.

2.2.4.3 start()

Begins the processing of data by calling the “process” function. This allows for better control for processing large sets of data.

2.2.4.4 stop()

Stop will interrupt the “process” function. This allows the user to stop the “process” function in case of extremely large data sets or very calculation intensive processors. Stop is more like “cancel” since you can’t call start again on the same “process” function.

2.2.4.5 preProcess(sessionID)

Parses the PCAP file it created for useful information, typically by the name of the application or API name, as well as by gets, pushes and other requests. This function returns a JSON file of the parsed PCAP file.

2.2.4.6 getPcapByID(pcapID)

Obtain the PCAP file associated with pcapID by searching the database for the session containing it, linking the PCAP file to the current session for processing.

2.2.5 *_Processor

The processor takes in a session ID, and parses the associated PCAP file. The processor analyzes the PCAP for any meaningful patterns, or produces reports and output. Each processor has a well-defined function or purpose. TADAPI's architecture allows for a modular plugin system so that third party developers can create processors in addition to those that ship with TADAPI.

2.2.5.1 process(sessionID)

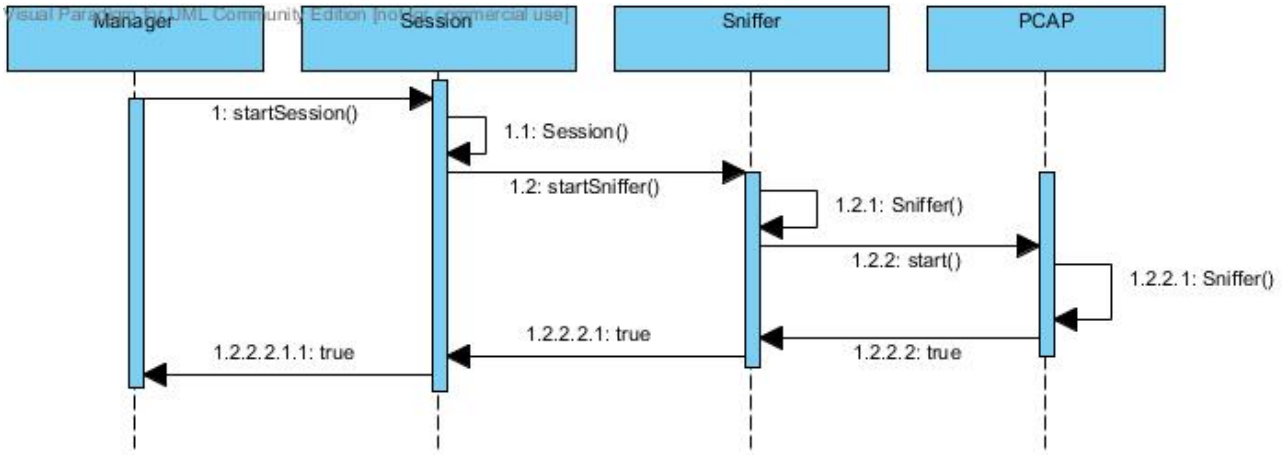
This will take the Session ID and parse the associated PCAP file. The process function will analyze the PCAP file for meaningful patterns and pushes it to the database.

2.2.5.2 export_html()

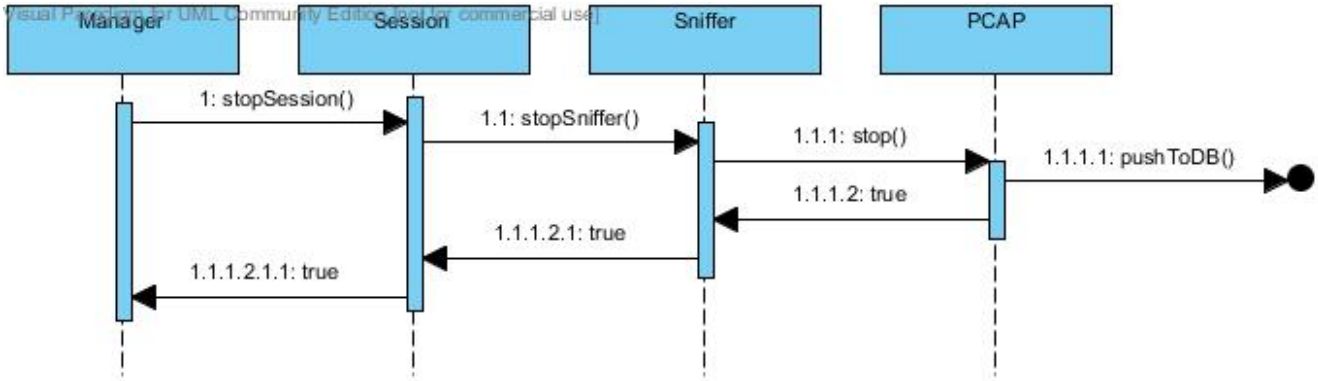
Export the data found by the process function into HTML.
Export the data into HTML format for easy viewing of the API. This function will produce a specifically formatted set of HTML files in an archive so that it can be easily pushed online for others to view. Developers who create their own processors are able to create custom HTML templates as they see fit.

2.3 Sequence Diagram

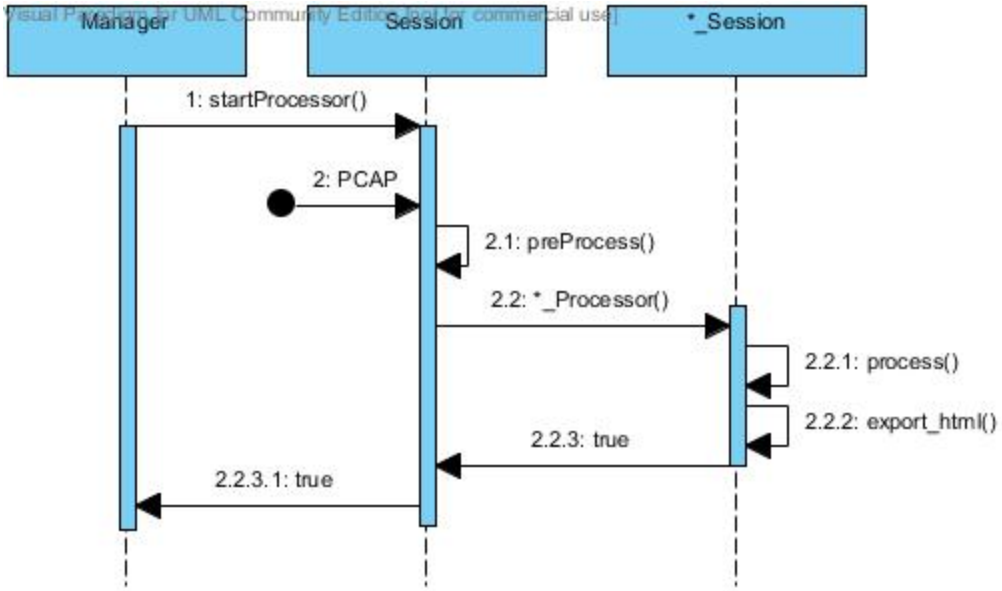
Pressing start session button



Pressing stop session

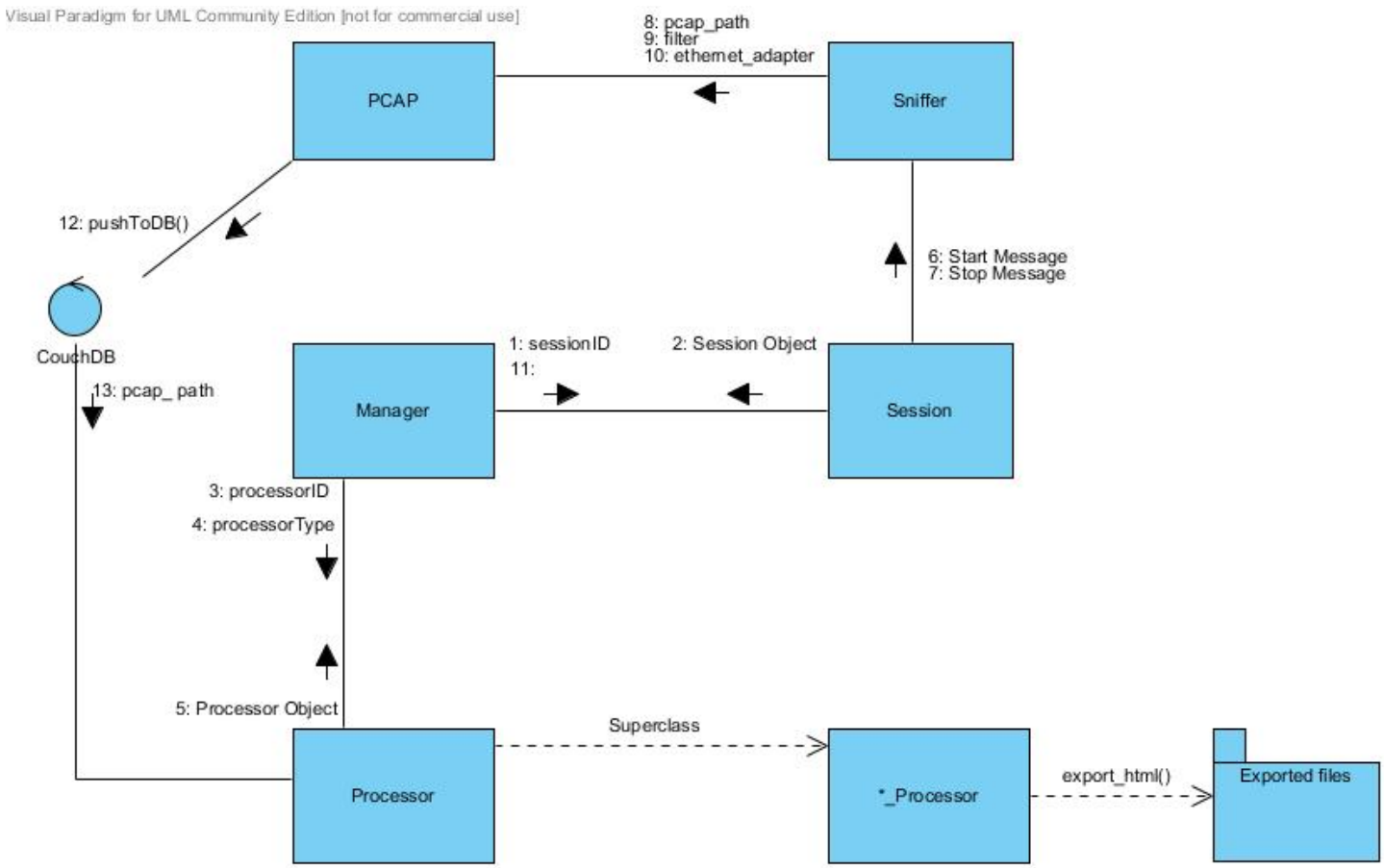


Starting processor and processing information



2.4 Dataflow Diagram

The dataflow diagram illustrates the flow of essential data between each class of this application and how inputs are acquired as well as where outputs are sent to.



2.5 Prototype Tests

The following are prototypes of tests Nibble will be using to execute their test driven development. Ruby tests will be used for back end verification while Selenium tests will be used for front end testing.

2.4.1 Ruby Tests

```
def TestSniffer()
  s = Sniffer.new("pcapfile.pcap", adapter[0], filter[0])
  result = File.exist?(filename)
  assert_equal(result, true)
end

def TestSnifferStart()
  s = Sniffer.new("pcapfile.pcap", adapter[0], filter[0])
  s.start()
  assert_equal_(s.isActive(), true)
end

def TestSnifferStop()
  s = Sniffer.new("pcapfile.pcap", adapter[0], filter[0])
  s.start()
  assert_equal_(s.isActive(), false)
end

def TestManagerStartProcessor()
  m = Manager.new()
  m.startProcessor(1)
  assert_equal(m.isProcessActive(1), true)
end

def TestManagerDBConnectivity()
  m = Manager.new()
  result = m.connectToDB()
  assert_equal(result, true)
end

def TestManagerCreateSession()
  m = Manager.new()
  result = m.createSession()
  assert_equal(result, true)
end

def TestGetSessionByID()
  m = Manager.new()
  id = m.createSession()
```

```
  result m.getSessionById(id)
  assert_equal(result, true)
end

def TestGetProcessors()
  m = Manager.new()
  processors = m.getProcessors()
  result = processors.count() > 0
  assert_equal(result, true)
end
```

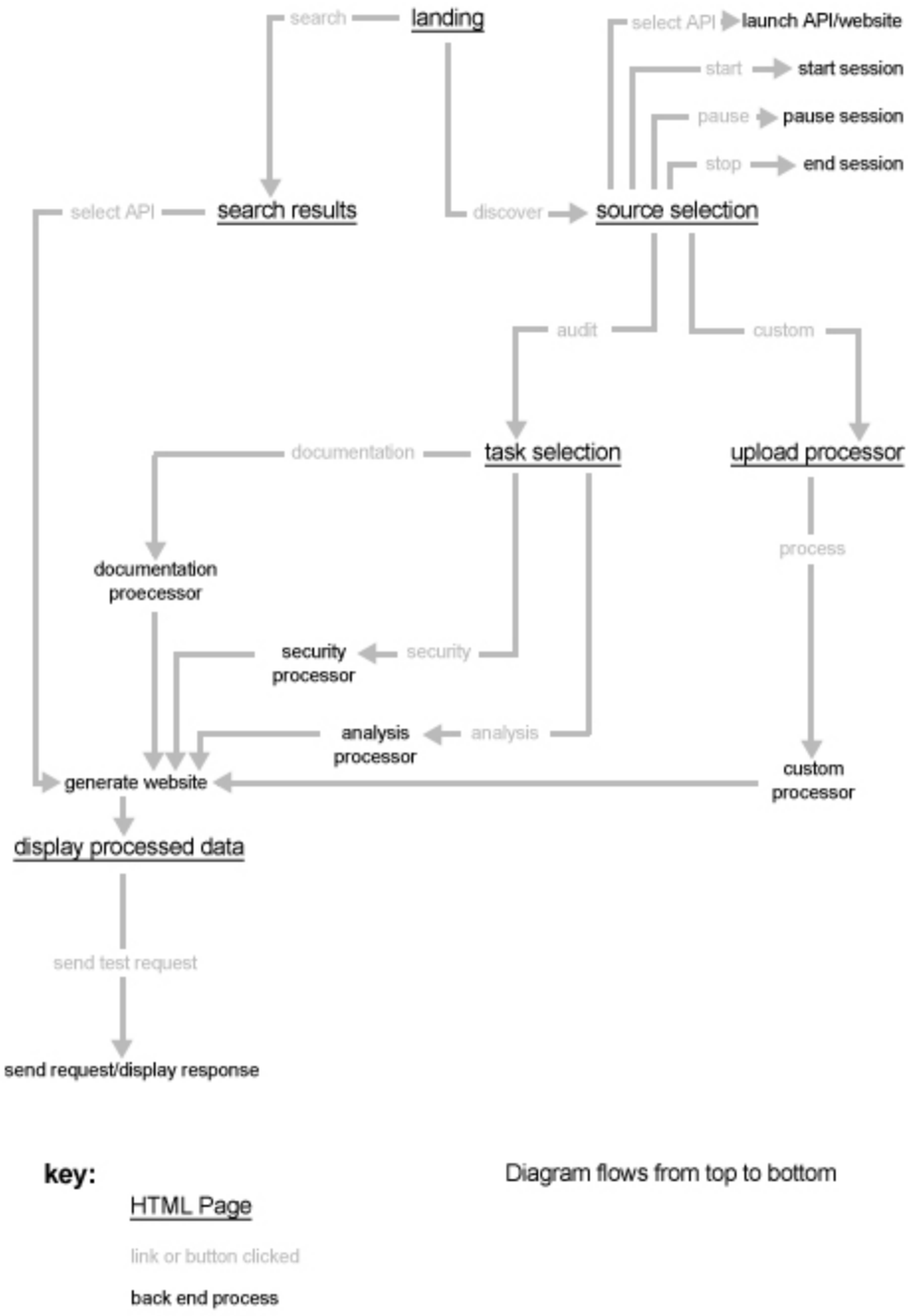
2.4.2 Selenium Tests

```
def testSearch()
  driver = Selenium::WebDriver.for :ie
  driver.navigate.to('TADAPI');
  driver.find_element(:id, "discoverButton).click;
  url = driver.getCurrentUrl();
  assert_equal(url, "TADAPI/discover");
  searchTextbox = driver.find_element(:id, "searchTextbox");
  searchTextbox.sendKeys("Twitter”);
  searchButton = driver.find_element(:id, "searchButton");
  searchButton.click;
  searchList = driver.find_element(:id, "searchList");
  text = searchList.getText();
  assert_equal(true, text != "");
end

def testSelectAPI()
  driver = Selenium::WebDriver.for :ie
  driver.click(“css=a:contains(‘Search Results’):nth-child(0)”)
  confirmSelectionButton = driver.find_element(:id, “confirmSelectionButton”);
  confirmSelectionButton.click;
  url = driver.getCurrentUrl();
  assert_equal(url, ”TADAPI/audit”);
end

def testAPIResponse()
  driver = Selenium::WebDriver.for :ie
  sampleGETTextbox = driver.find_element(:id, “sampleGETTextbox”);
  sampleGETTextbox.sendKeys(“https://api.twitter.com/1/statuses/user
    _timeline.json?include_entities=true&include_rts=true&screen_name=
    twitterapi&count=2”);
  driver.find_element(:id, “getResponseButton”).click;
  actual_response = driver.find_element(:id, “resultsDiv”).getText();
```

```
 expected_response = "[
  {
    "coordinates": null,
    "truncated": false,
    "favorited": false,
    "created_at": "Mon Jun 27 19:32:19 +0000 2011",
    "id_str": "85430275915526144",
    "entities": {
      "urls": [
        {
          "expanded_url": "http://tumblr.com/xnr37hf0yz",
          "url": "http://t.co/cCIWIwg",
          "indices": [
            107,
            126
          ],
          "display_url": "tumblr.com/xnr37hf0yz"
        }
      ],
      "hashtags": [

      ],
   }
   ]";
  assert_equal (actual_response, expected_response);
end

testStartSessionButton()
  driver = Selenium::WebDriver.for :ie
  driver.find_element(:id, "startSession").click
  m = Manager.getManager();
  assert_equal(m.sessionIsActive(), true);
end

testStopSessionButton()
  driver = Selenium::WebDriver.for :ie
  driver.find_element(:id, "endSession").click
  m = Manager.getManager();
  assert_equal(m.sessionIsActive(), false);
end
```

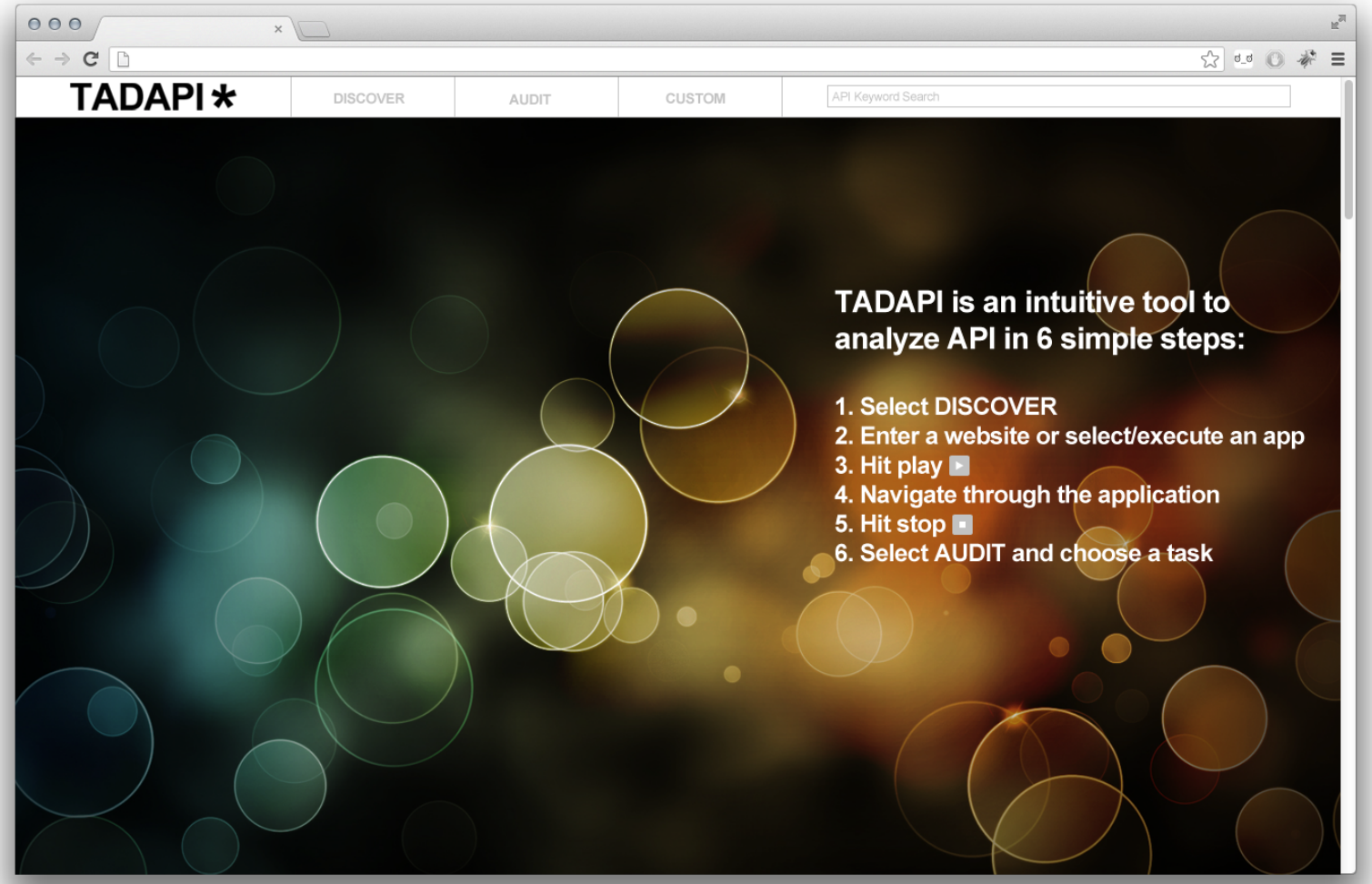
2.6 State Diagram

The following diagram is a state diagram given user inputs. The states are underlined and represent specific webpages documented in the UI Mockup. User input is represented as the grey fonts and lines, and the smaller black font represents a backend process as a result of user input.

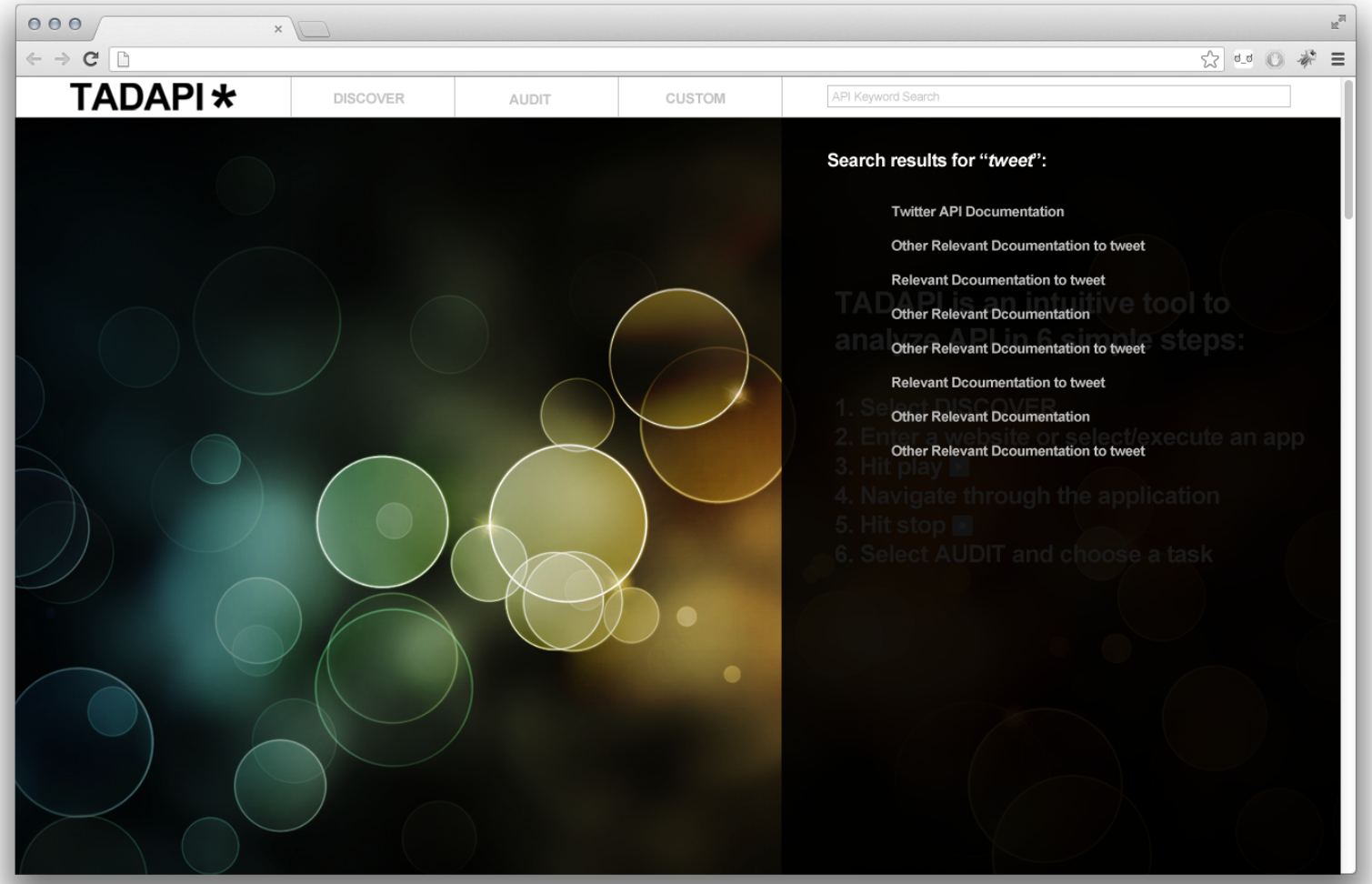


2.7 UI Mockups

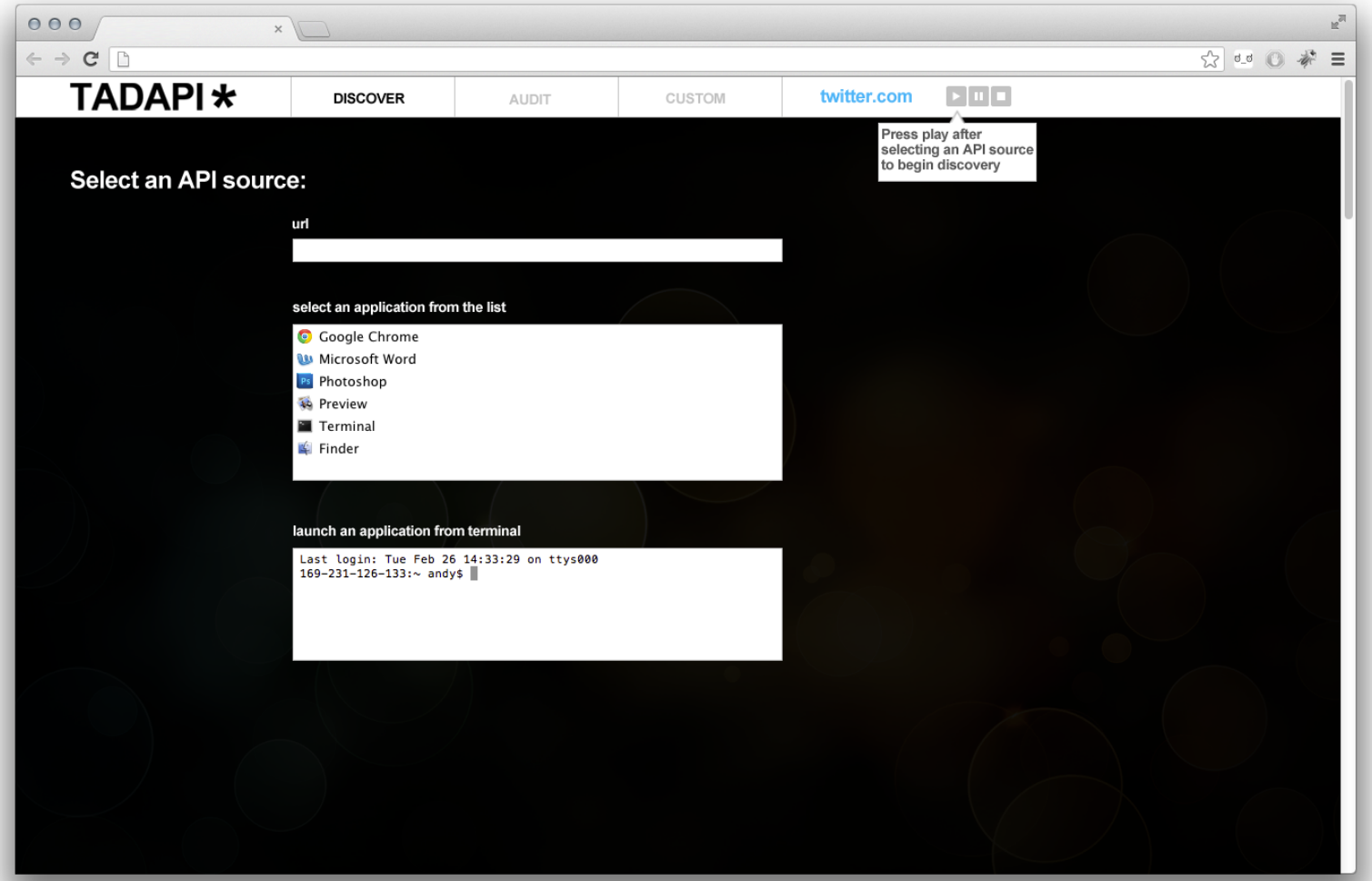
2.7.1 Landing



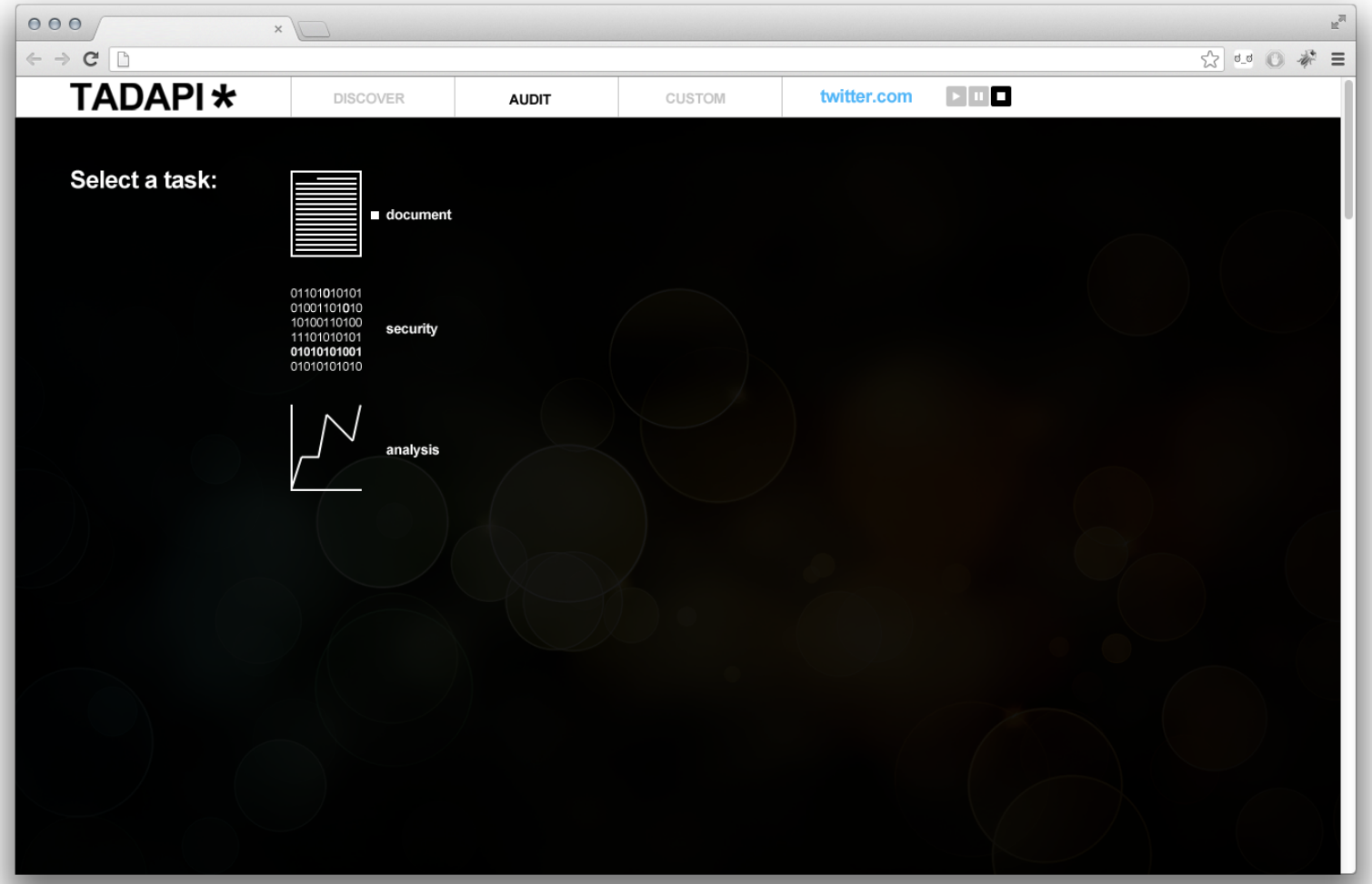
2.7.2 Search Results



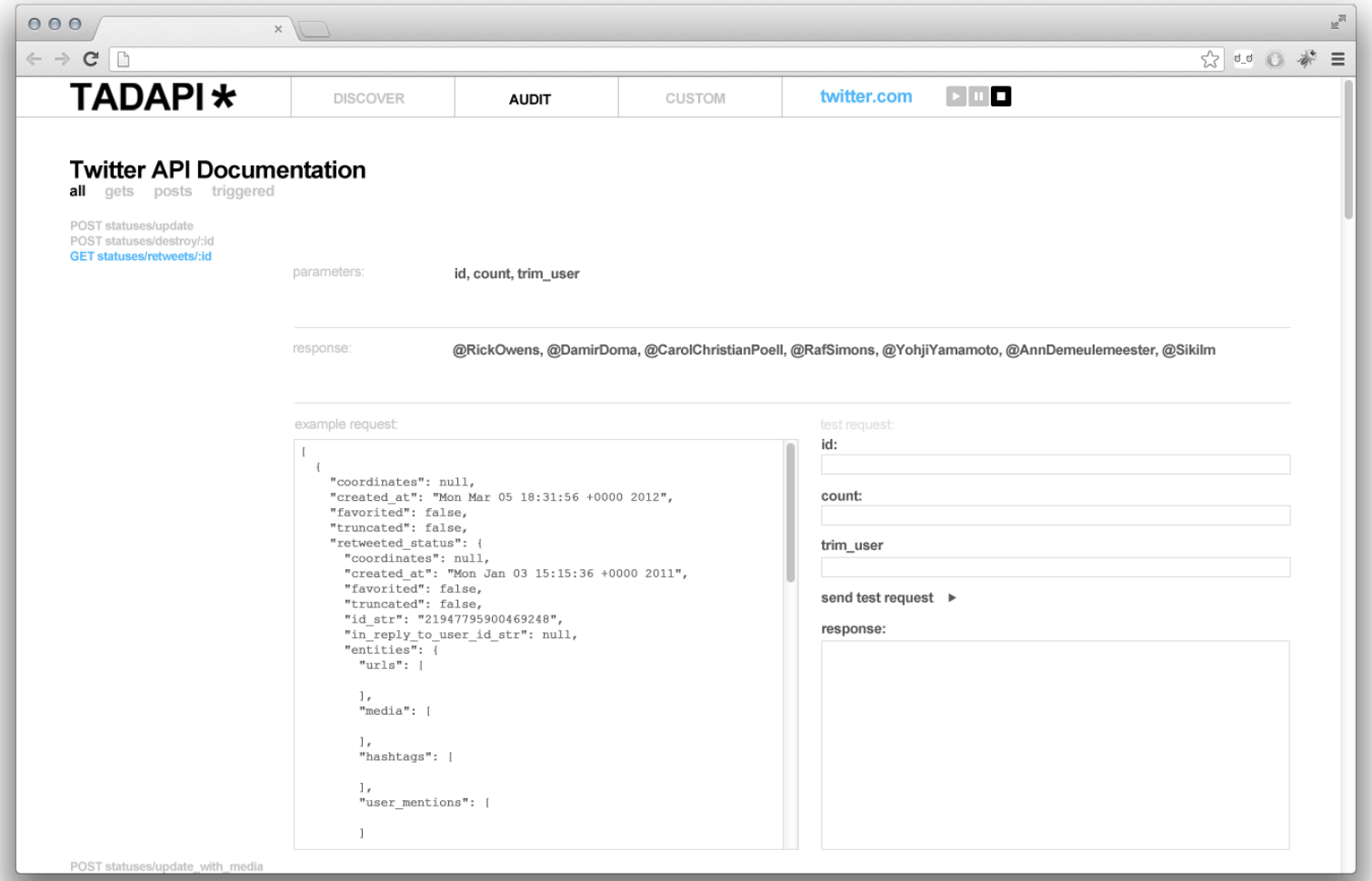
2.7.3 Source Selection



2.7.4 Task Selection



2.7.5 Display Processed Data


TADAPI*
DISCOVER
AUDIT
CUSTOM
twitter.com
Twitter API Documentation
all gets posts triggered
POST statuses/update
POST statuses/destroy/:id
GET statuses/retweets/:id
parameters:
id, count, trim_user
response:
@RickOwens, @DamirDoma, @CarolChristianPoell, @RafSimons, @YohjiYamamoto, @AnnDemeulemeester, @Sikilm
example request:
[
{
"coordinates": null,
"created_at": "Mon Mar 05 18:31:56 +0000 2012",
"favorited": false,
"truncated": false,
"retweeted_status": {
"coordinates": null,
"created_at": "Mon Jan 03 15:15:36 +0000 2011",
"favorited": false,
"truncated": false,
"id_str": "21947795900469248",
"in_reply_to_user_id_str": null,
"entities": {
"urls": [
],
"media": [
],
"hashtags": [
],
"user_mentions": [
]
test request:
id:
count:
trim_user
send test request
response:
POST statuses/update_with_media

I. Change Log

Date	Version	Author	Changes
3/7/2013	v0.1	NIBBLE	First Draft