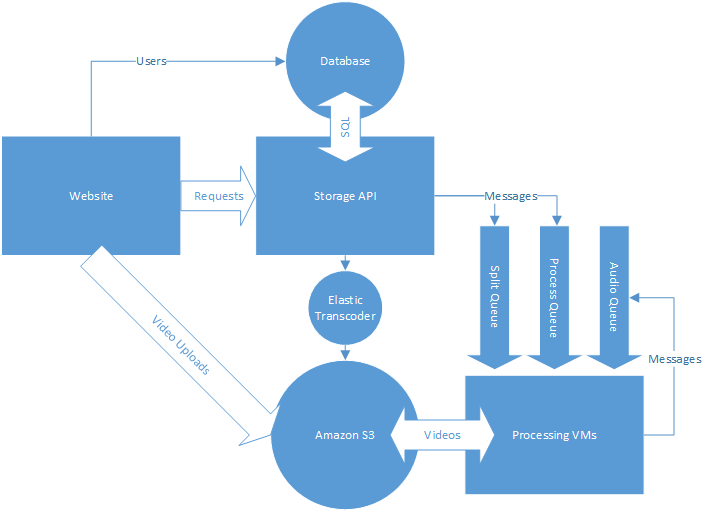
# Mario Kart 64 Analytics Suite

Software Report

# Systems Overview



There are three primary components to this system, the Website, the Storage API, and the Processing VMs. The website communicates with S3 to upload videos and with the Storage API to request complex data to display. The storage API accepts requests from both the website and the processing VMs (the latter is not shown) to manipulate the database. The storage API also deals with dispatching work to the processing VMs by way of feeding work to be done into message queues. The processing VMs accept work to be done via the queues and run the video and audio analysis suite on the videos they’re instructed to. These pieces are documented more fully in the rest of this document.

# Amazon Web Services — Instructions

This project uses Amazon Web services for basically everything. Here are some setup instructions.

* First you should create an account, then use IAM to create a couple of roles. These will be used for authentication by the VMs. Create these roles with permissions.
  + video-processing: S3 full access, SQS full access, EC2 full access
  + storage-api: SQS full access, elastic transcoder full access, EC2 full access
  + A user with s3 full access.
  + You should also go ahead and create a keypair.
  + Set up personal user account too
* Now we’ll setup up the Database
  + Open up Amazon RDS and create a new instance.
  + We want Postgres, pick the latest version, choose an instance size based on how big a db you want. Pick a size and some credentials.
  + Name the database something cool, and figure out where you want to put it
* Let’s get some buckets
  + Go to S3
  + create a bucket for session videos, race videos, and race audio, and an additional bucket for the processing VMs to get code from.
  + Add a CORS config to the session bucket with the content of the session\_bucket\_cors.xml file.
  + In the misc files bucket upload a zip of the n64\_img\_processing files.
* Now we’ll set up the API
  + Go to the n64\_storage\_flask git repository
  + run “eb init” (make sure you have elasticbeanstalk tools installed)
  + Fill out it’s prompts
  + And run “git aws.push”
  + Now go to the elastic beanstalk website, add an environment variable called CONFIG\_MODULE with the value “n64\_storage.config.AWSConfig”.
  + Set the instance profile to be the role you set up with sqs, ec2, and elastic transcoder access (the storage-api role).
  + ssh into the instance and run
    - source /opt/python/run/venv/bin/activate
    - /opt/python/current/app/manage.py db upgrade
* And now the website
  + Go to the website git repository
  + run “eb init”
  + Fill out the prompts
  + Run “git aws.push”
  + On the ElasticBeanstalk website you should update the config for the appplication to have a static path of “n64/staticfiles” and a DJANGO\_SETTINGS\_MODULE of n64.settings.
  + You should also paste you AWS access keys and secret keys for the user with s3 full privileges to their respective fields.
  + ssh to the website and run
    - source /opt/python/run/venv/bin/activate
    - python /opt/python/current/app/manage.py syncdb
* Next we’ll setup autoscaling
  + Create a launch configuration with AMI ami-cbddc4a2, (this has all of the software you need installed), you could also build this manually, by installing all of the dependencies of n64\_img\_processing into a VM and creating an image of the VM.
  + Have it launch spot instances of size c3.xlarge, have the user data be the startup script included on this CD. You should modify the startup script to reference your buckets.
  + Have the VM have an IAM role of the role you set up with ec2, s3, and sqs access (the video-processing role).
  + Add this to an autoscale group called ‘video-processing-group’ and don’t set any scaling criteria.
* Now we’ll set up message queues
  + Create 3 queues in Amazon SQS named ‘audio-queue’, ‘split-queue’, and ‘process-queue’.
  + Set the visibility timeout of audio and split to be 15 minutes, and the visibility timeout of process to be 45 minutes.
  + Add a CloudWatch alarm to monitor the number of visible messages in process and have it scale up your autoscale group when the alarm is set.
* Finally we’ll set up Elastic Transcoder
  + Open up elastic transcoder and create 2 pipelines
  + The first will be for transcoding session videos
    - Name: encode-sessions
    - Input Bucket: your session bucket
    - Transcode Bucket: your session bucket
    - You can use reduced redundancy storage if you want
  + The second will transcode races
    - Name: encode-races
    - Input bucket: your race bucket
    - Transcode bucket: your race bucket
    - Reduced redundancy is acceptable
  + Figure out the ids of these and update the storage api configs with them.
  + If you want a different preset for transcoded videos, set the id of that preset in the config file as well. The included one should work for transcoding for web

# 

# Video and Audio Processing

## Quick Setup

On some Linux distribution, install pip and virtualenv. Inside the video processing directory (n64\_image\_processing), run the command virtualenv env to generate a virtual environment. Activate this using source env/bin/activate. Now you can install all Python dependencies by running pip install -r requirements.txt. Note that gcc must be installed on the system for this to succeed. Once completed, proceed to install OpenCV and ffmpeg.

## Dependencies

ffmpeg

Manipulation of video and audio files in this project is done with the ffmpeg utility. We use it to convert, split, and read video files in addition to extracting audio data. In addition to the main utility, we also used a few of the more common video encoding libraries.

**Version Used:** 2.2

**Developer:** FFmpeg team

**Contact:** ffmpeg-user@ffmpeg.org (mailing list)

**Additional Libraries:** libx264, libaacplus

OpenCV

The video processing component of this project uses the Open Computer Vision (OpenCV) library for manipulating the video and recognizing objects. Specifically, you will need to build OpenCV with ffmpeg support.

**Version Used:** 2.4.9

**Developers Team:** itseez

**Contact:** info@itseez.com

Boto

The Amazon Web Services library. This drives all of the AWS integration. For the video processing component, we use the simple queueing service (SQS), the elastic compute cloud (EC2), and the simple storage service (S3). Installation of this is outlined in the Quick Setup section.

**Version Used:** 2.27.0

**Developer:** Amazon

**Contact:** AWS Support Center (https://**aws**.amazon.com/contact-us/‎)

## Software Modules

dispatcher.py

This is the main driver and front end of the entire processing suite. It uses Amazon Web Services (AWS) components in order to dispatch jobs for processing uploaded videos. Listening on all AWS SQS job queues in a while-loop, the script determines the type of job to dispatch and downloads the corresponding data. As this script is meant to be run at the launch of every new VM in Amazon’s Elastic Compute Cloud (EC2), it can also be run with a --daemon flag to force it to run in the background.

api.py

The job dispatcher utilizes the methods defined in this file for communicating with the various AWS components and the database. Specifically, it contains classes for representing SQS, EC2, S3, and database connections. Wrapped by these classes are methods for downloading to and uploading from S3 buckets; reading, writing, and deleting SQS messages; terminating EC2 instances; and posting event and race data to the database.

const.py

Container file for AWS-related constant fields. Specifically, it holds the SQS queue names, the S3 bucket names, and the base URLs for the S3 buckets.

phase\_0.py

Wrapper module for the task of splitting a Mario Kart 64 session video into individual race videos. It initializes the detectors necessary for said task and the video processing engine. After initialization, it begins processing. Upon completion, it returns a dictionary of session variables which will be processed by the job dispatcher.

phase\_1.py

Wrapper module for the task of detecting events in Mario Kart 64 races. This module is quite similar to phase\_0.py, with the main difference being the types of detectors being initialize. Once the event detectors have been initialized, they get passed into the video processing engine and start processing the video source. The module returns a list of “Event” dictionaries which will be processed by the job dispatcher.

audiodetect.py

Wrapper module for the task of detecting exciting moments and key phrases in Mario Kart 64 races. It prepares the audio file and the audio processing API, processes the data, and returns a list of “Event” dictionaries which will be processed by the job dispatcher.

run\_tests.py

Wrapper module for unit tests contained within the testing directory. It generates test suites for each and runs a test runner from Python’s native unittest module.

detection/config.py

Contains configuration variables and objects used during processing. The DEBUG\_LEVEL global variable is used to control the verbosity of the suite. The race and player objects represent dictionary templates to store important data during the two phases of processing. Both of them are derived from Python’s multiprocessing module as they are used by child processes during detection.

detection/generic.py

This file contains all classes shared between the two phases: the abstract Detector class, the class determining how black a frame is, and the video processing Engine class. The heart of the image recognition used in this project is in the Detector.process() function. The Engine class is what controls the data flow shared with the subprocesses, populating a frame buffer and alerting the worker processes of the new data.

detection/splitting.py

Contains the core classes used in phase 0. Each one is derived from the abstract Detector class and defines detector-specific handling functions for extracting important race information such as map type, characters selected, number of players, etc.

detection/processing.py

Contains the core classes used in phase 1. Similar to splitting.py, the classes in this file are derived from the abstract Detector class and define specific handling mechanisms for generating a list of “Events” occurring in a race.

detection/audioprocessing.py

Contains the class consisting of functions used for the processing of race audio. This includes everything from processing the signal obtained from an audio file to sending requests to google for speech recognition.

detection/parallel.py

Core module used for parallelizing the work done by the Detectors in each phase.

It outlines the Worker, derived from Python’s multiprocessing.Process class, which has access to a read-only shared frame buffer. The video processing Engine described earlier alerts each Worker that this buffer has been filled, which triggers the Worker to apply Detectors on the video. Additionally, this file contains a ProcessManager class used by Engine for handling multiple Workers.

detection/utility.py

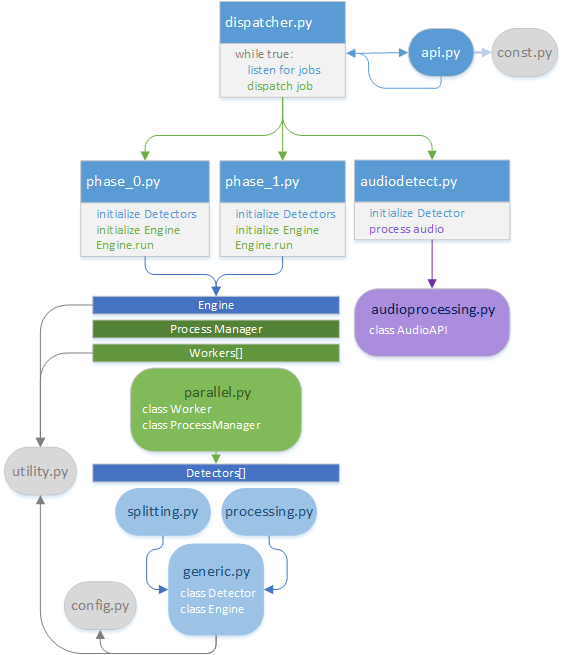
Utility module containing objects used frequently in splitting and processing, the most important being the Barrier class. This class is a multiprocessing.Barrier implementation. Within this class, we provide an implementation of a semaphore, as well as its corresponding release. Once the number of allotted frames has been buffered in shared memory, each worker will process those frames. Once processing completes, the Worker will wait at the barrier for all other Workers to finish processing. In addition to the multiprocessing.Barrier implementation, we provide several useful functions and classes for all Detectors to utilize. The RingBuffer class is an implementation of a circularly linked list, using a deque as the backend. The scaleImage() function scales an image from one size to another. The in\_range() function simply returns whether a value is in a certain specified range. The find\_unique() function is used to find unique elements in a container. This is mainly used by the Character detector class.

testing/all\_tests.py

Wrapping script for generating test suites for all unique tests within the testing directory. This gets launched by the run\_tests.py script.

testing/test\_\*.py

These glob files represent all the unit tests used during development. The tests themselves test common cases and uncommon cases, asserting that the generated outputs are correct.



# Storage API

## Quick Setup

You should be on some type of Linux. First install pip and virtualenv. Inside the n64\_storage\_flask folder run “virtualenv env”. Then active the virtualenv by running “source env/bin/activate”. Then you can install the dependencies by running “pip install -r requirements.txt”. Make sure you have gcc and the postgresql-devel packages installed or this will fail. From here you can run “python manage.py runserver -t 0.0.0.0:5000” to run a server on port 5000 that will answer to any computer on the internet.

## Dependencies

The full length of dependencies is documented in the requirements.txt file. Some more important dependencies are explained here.

[SQLAlchemy](http://www.sqlalchemy.org/)

SQLAlchemy is probably the best ORM software that exists. It uses the “mapper” pattern which makes it a slightly lower level interface to the database as compared to something like DJango’s ORM framework. It still beats writing SQL though, and this slight increase in complexity makes for a much more flexible and performant library.

[Flask](http://flask.pocoo.org/)

Django was overkill for this piece of the project. We use Flask as the framework behind the REST API.

[Flask-Restful](http://flask-restful.readthedocs.org/)

Small polish on top of Flask to make it easier to make REST APIs. In particular, it makes it possible to write the URL endpoints nicely.

[Flask-SQLAlchemy](http://pythonhosted.org/Flask-SQLAlchemy/)

Small library to better integrate SQLAlchemy into the Flask request/response cycle.

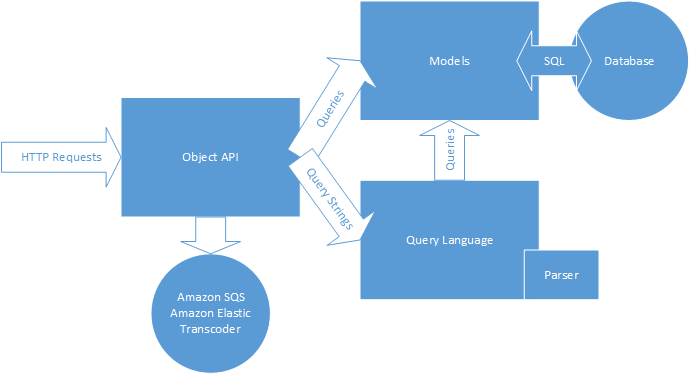
[Boto](http://boto.readthedocs.org/)

The Amazon Web Services library. This drives all of the AWS integration. We use the ec2 autoscaling module, the sqs module, and the elastic transcoder module. In the future we could use the s3 module.

[Pyparsing](http://pyparsing.wikispaces.com/)

An object-oriented parser library. This is used to parse queries that users input.

## Block Diagram



## Files

manage.py

The manage contains a number of different functions that allow one to manage the database. This includes things like running a development server, upgrading/migrating the database, and running tests.

application.py

Application.py is a WSGI entry point for running this on AWS Elastic Beanstalk. This file allows you to use EB to deploy the code using git.

migrations/versions/

This directory documents the changes in the db schema and includes scripts for how to get from one version to another. These are used in conjunction with the manage command.

n64\_storage/models.py

Database models. These get mapped to/from the SQL database and allow one to use rows from a database like python objects. These use SQLAlchemy’s declarative base toe describe what the database columns should look like.

n64\_storage/object\_api.py

This is where the REST API gets made. contains a number of classes which represent URL endpoints. These endpoints call one of the HTTP method functions defined on the class. The specific endpoints are further documented in the README. The various endpoints basically amount to doing pre-defined queries using SQLAlchemy and returning JSON of the query.

n64\_storage/query.py

The query generator part of the query language. The primary class is EventQuery, which takes a query string as an argument. It parses this and walks the parse elements, modifying its query as it goes.

n64\_storage/parser.py

The parser uses the pyparsing API to create a parser for the query language. It works by building up new objects out of smaller parser objects using a simple object oriented style. Parsers are defined by what they match, so combining parsers with operators lets you build complicated languages quickly.

n64\_storage/video\_api.py

This isn’t really a video\_api, it’s name is leftover from when it was. Now it submits jobs to Amazon elastic transcoder. So it *talks* to a video api I guess.

n64\_storage/tests/\*

There are a couple of test suites. The api test suite is supposed to check that the the API functions as desired. It doesn’t do this like it should though, since it tries to use SQLite which doesn’t have all the column types we need. This should be fixed someday. The query tests test both the query language itself as well as the parser. The query tests are kind of in a weird place because they test that queries with obvious results return those results. This means that they test the data in the DB as well as the actual query language.

# Mario Kart Simulator

## Quick Usage

If you have requests installed you can run this just by running “python simulator.py <nplayer> <nraces>” This wil fill the database with simulated race data. You’ll probably have to change the DB endpoint so do that by editing the line in the simulator.py file.

## Dependencies

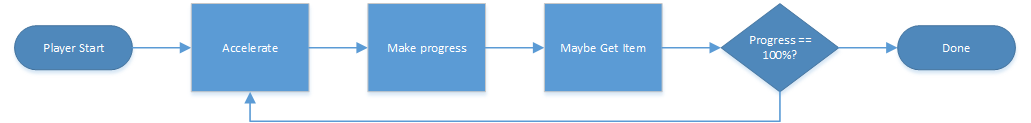
The only dependency for the simulator is the python requests framework. This can be retrieved by “pip install requests.”

## Files

simulator.py

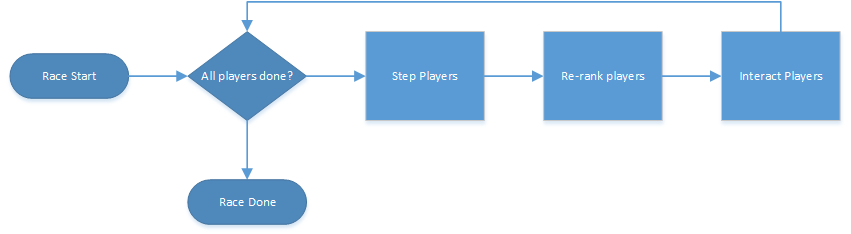
This file simulates a Mario Kart 64 race and puts the resulting sessions, races, and events into the database for usage in test queries. The simulator works by simulating the players in a race as Finite State machines. Each step they make progress through the course (in the form of a % complete) and accelerate. They can do things like use items on people near them, randomly slow down, or stop completely. This adds some randomness to the results. That said, the results are very regular and you will not get anything like accurate data out of this. It’s mainly for simulating so you have data to test queries against.

## Player Simulation Flow



The race runs each player from accelerate to the progress check every step.

## Race Simulation Flow



# 

# 

# 

# 

# The Karty Klub Website

## Quick Usage

This should be deployed on some flavor of Linux. First, install apache, postgresql-devel, gcc, pip, and virtualenv. Inside the kartlytics\_site folder, run “virtualenv env”. Then active the virtualenv by running “source env/bin/activate”. Then you can install the dependencies by running “pip install -r requirements.txt”. Next, run “python manage.py runserver 8080” to run the website on port 8080.

## Dependencies

Django

Python web development framework. This is the skeleton of the website. The Django files that control the website are outlined below.

**Django website:** [**https://www.djangoproject.com/**](https://www.djangoproject.com/)

**Django book:** [**https://github.com/jacobian/djangobook.com**](https://github.com/jacobian/djangobook.com)

Requests

This is a package for sending REST based calls to other servers. It is being used to query the database once the data is properly formatted.

**Requests website:** [**http://docs.python-requests.org/en/latest/**](http://docs.python-requests.org/en/latest/)

Wsgiref

This is the package that talks between python and apache. Wsgi.py uses it behind the scenes to point apache to the python application.

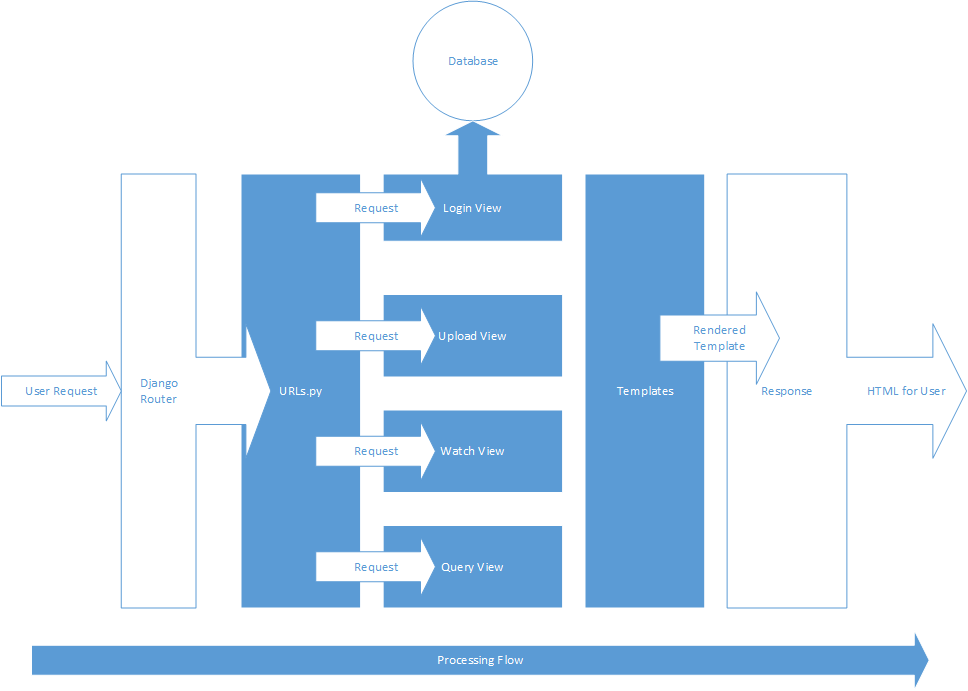
**Wsgiref website:** [**https://docs.python.org/2/library/wsgiref.html**](https://docs.python.org/2/library/wsgiref.html)

Psycopg2

This is the package that communicates between python and postgre-sql. In combination with Django, it does all of the configuration / maintenance of the databases for you.

**Psycopg2 website:** [**http://initd.org/psycopg/**](http://initd.org/psycopg/)

## Architecture Overview



## Files

manage.py

This is the file that runs or tests the server. It is the default file that comes with all django projects.

settings.py

This file stores the parameters for your website. This includes the installed applications, database specifics (currently pointed at the storage flask on amazon), location of the static folder, and middleware.

urls.py

This is the URL resolver for the website. It matches the urls based on regular expressions in the file, and sends the request to the appropriate view.

views.py

This file does all the heavy lifting with the data. It reads post requests, and queries the database for the required information. There is an individual view for each web page.

forms.py

This file defines two forms that are passed through the views to the html templates.

wsgi.py

This file is the web server gateway interface. It does not need to be touched, just sets the application to the project in the current folder when manage.py is run.

## Folders

templates/

This folder holds all of the html template files. The templates are based around the django template model. All templates are derived from base.html, which serves as the home page. There is a template for query, watch, and upload as well. Some code has been separated out of base.html into templates/include/, including the navbar, footer, account (login), scripts, and stylesheets.

staticfiles/

This is the folder that holds all of the static files for the website, including the javascript, bootstrap css, karty klub css, and images populating the website.