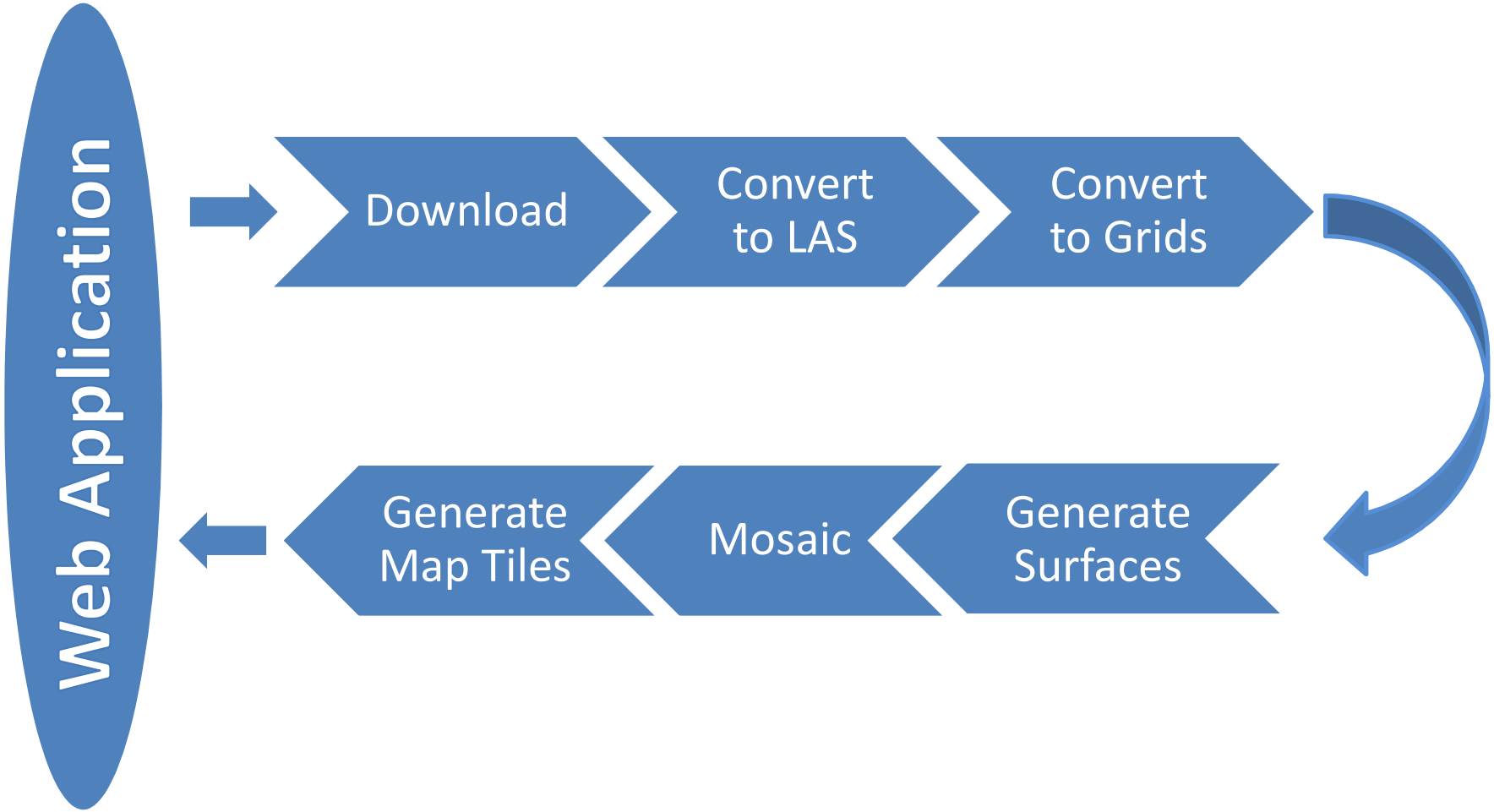
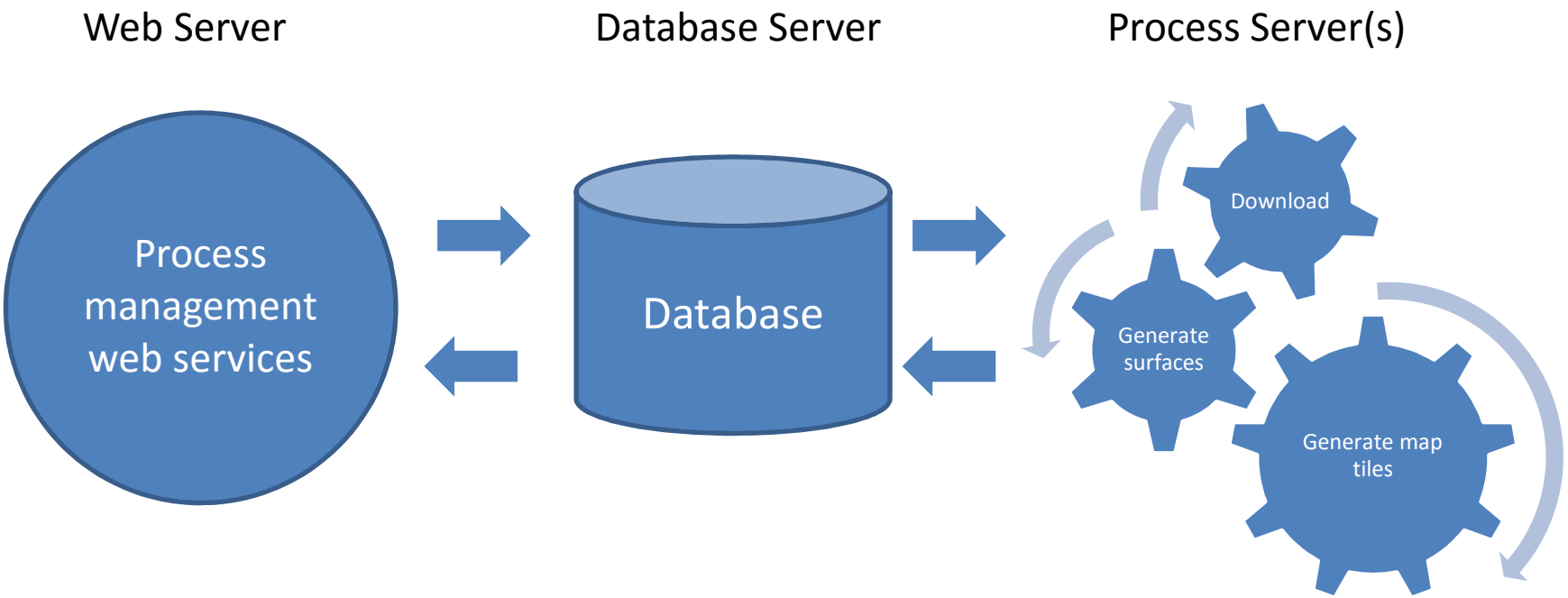
Job Pipeline

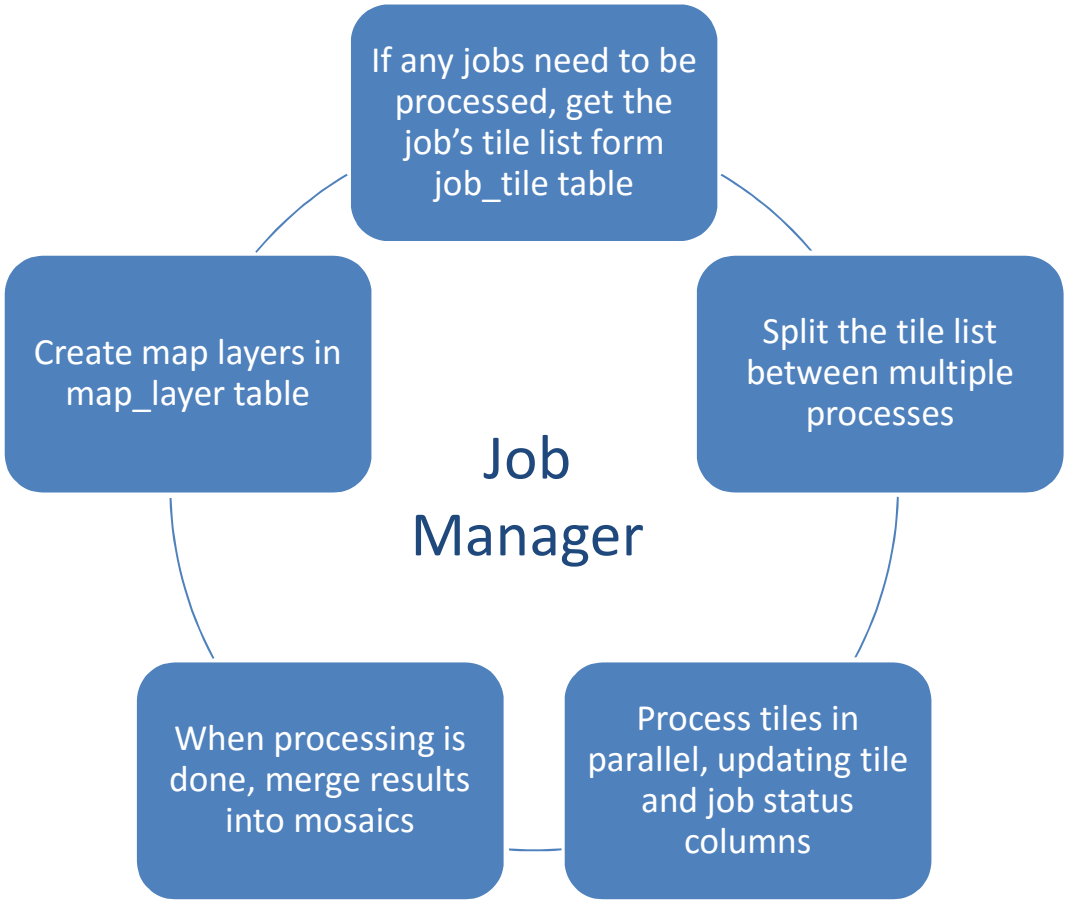


Job Control Mechanism

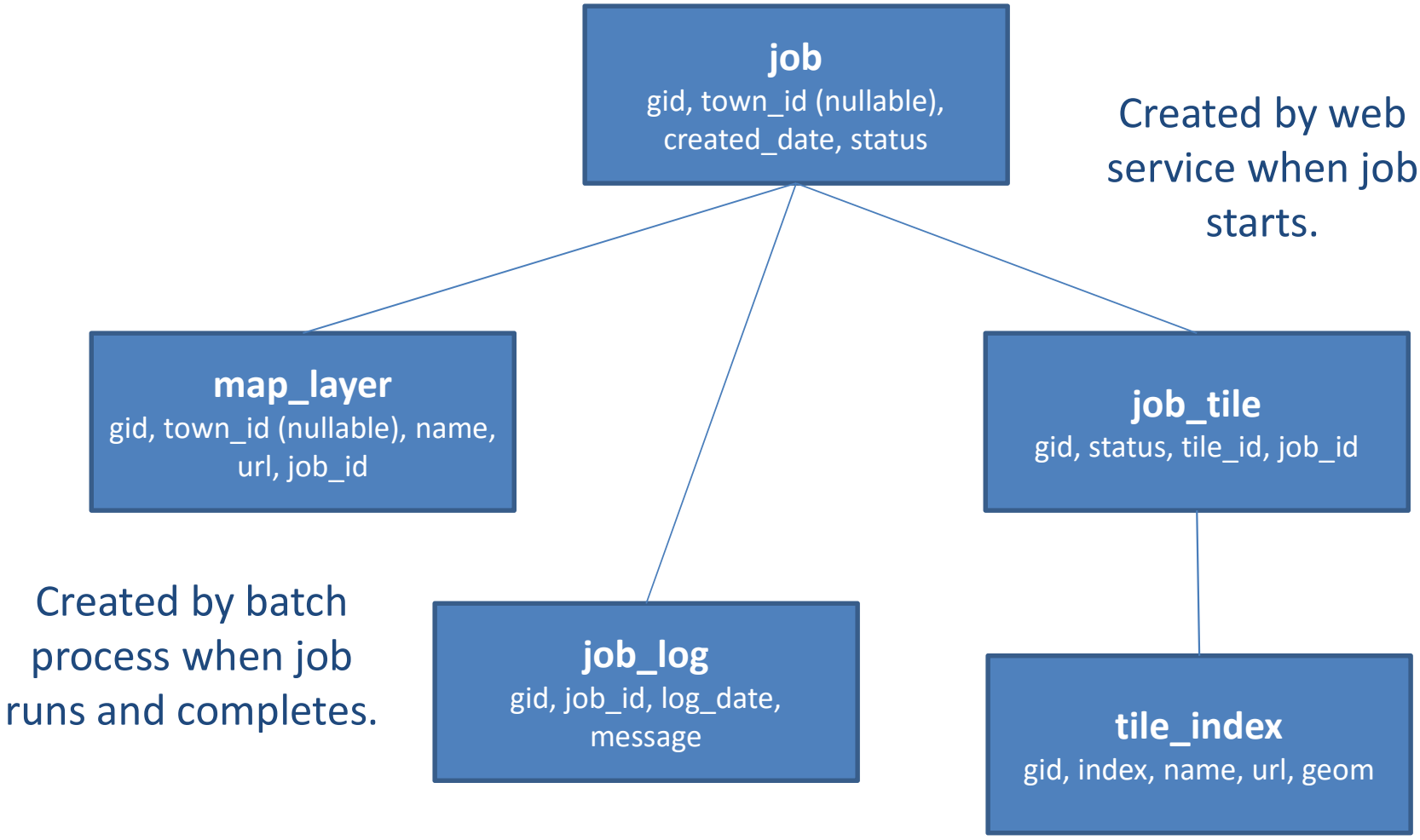


/start_job?tile_ids=4315,4283,4318,4286
/start_job?town_id=11850
/job_status?job_id=29
/cancel_job?job_id=29

Job Manager


If any jobs need to be processed, get the job's tile list form job_tile table
Split the tile list between multiple processes
Process tiles in parallel, updating tile and job status columns
When processing is done, merge results into mosaics
Create map layers in map_layer table
Job Manager

Job Database



Configuration

For Windows:

subprocess and **multiprocess** modules are built in.

GIS libraries need to be installed:
- Install Anaconda Python 2.7 64-bit
- Install SAGA GIS
- Install GDAL code and GDAL python bindings according to these instructions:
  https://sandbox.idre.ucla.edu/sandbox/tutorials/installing-gdal-for-windows

Set up config file paths to point to the install directories
Start service from /services using **python server.py**
Start batch process from /process using **python main.py**
Process will loop until a job becomes available in the job table from a call to the /start_job endpoint

Database Details

Job is parent table to **job_tile** and **job_log**. Foreign key constraints set to delete child records when parent is deleted, and to prevent orphan records from being left behind. Default values for date in log table and primary key from sequence cut down on what has to be managed by code:

```
CREATE TABLE public.job_log
(
    gid bigint NOT NULL DEFAULT nextval('job_log_gid_seq'::regclass),
    job_id integer NOT NULL,
    log_date date NOT NULL DEFAULT ('now'::text)::date,
    message character varying(1000) COLLATE "default".pg_catalog NOT NULL,
    CONSTRAINT job_log_pkey PRIMARY KEY (gid),
    CONSTRAINT fk_job FOREIGN KEY (job_id)
        REFERENCES public.job (gid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;
```

Selecting from sequence guarantees that job ids are unique:

```
select nextval('job_gid_seq')
```

Code Details

Multiprocess creates a separate processes, each with their own environment (like separate command-line consoles). This seems to be more stable than the threading module, which creates multiple threads within the same single process. Note that it requires that the process be a function, not an object:

Database.py wrapper object provides centralized db utilities – it can be easier to add logging, debug, etc., when all the db access is in one place. But watch out for closing connections – can run out of system resources if connections are not closed.

To run in a multi-threaded environment like Cherrypy, you have to be careful that unique values are maintained. Using Postgres' **returning** statement ensures that the operations are atomic, for example:

```
sql = """update job set status = 'in progress'
   where gid in (select min(gid) from job where status = 'ready')
   returning gid"""
```

Code Details

Job_manager object runs all the component tasks using multiprocess to run in parallel, and these create command-line instances using the process module.

Per-tile processing can be run in parallel. With the multiprocess module, this is easy:

```
args = []
for i in range(self.num_threads):
    tiles_this_thread = tile_list[tile_count_step * i:tile_count_step * (i+1)]
    args.append(tiles_this_thread)
pool = Pool(self.num_threads)
pool.map(process_tiles, args)
pool.close()
pool.join()
```

Mosaicking and generating tile must happen in a single thread, so these happen after the tile-based processes are complete.

Code Details

Cherrypy server setup:

```
class Server(object):

    def get_response_wrapper(self):
        # Gets a wrapper for service reponses
        return {
            "status": 200,
            "data": dict(),
            "message": "Success"
        }

    def set_response_headers(self):
        # Sets the response headers to url can be accessed from any web
        # site, and so the browser recognizes the content as json text.
        cherrypy.response.headers["Access-Control-Allow-Origin"] = "*"
        cherrypy.response.headers['Content-Type'] = 'application/json'

    def encode_results(self, result):
        # Encodes strings in utf-8 format.
        self.set_response_headers()
        return json.dumps(result).encode('utf-8')

    @cherrypy.expose
    def index(self):
        # Index runs when no web document or "route" is specified
        output = self.get_response_wrapper()
        output["data"] = "running!"
        return self.encode_results(output)
```

Wrapper output:

```
{
    status: 200,
    message: "Success",
    data: "running!"
}
```