* Why REST
  + Simple, easy to use/understand
* Horizontally-Scalable Web Application
  + Horizontally-Scalable
    - Multiple instances of your code will be running on seperate machine
  + REST provides a way to balance the load of requests between machines
  + Capacity can be dynamically added to handle increased load
  + Servers are data-less, they communicate with a single database
* REST Background
  + Good for
    - Minimal documentation
    - Ease of testing
    - Designing a system by contract
  + REST is not a standard
    - Certain things you can do with REST do not have standard methodologies
* Nouns
  + Just like in OO, figure out your nouns which will have database representations
* Verbs
  + Use the same format (like json). POST and GET strings could look identical for a single entry.
  + Accessed from a url with something like webaddress.com/api/tableName/id
  + GET
    - Get information from the database (whole collections or single items)
  + PUT
    - Modifies a database entry by receiving a whole database object, modifying a field, and sending it back.
  + POST
    - Add an entry to the database
  + PATCH
    - Modifies a database entry by simply sending the particular field the user wishes to modify
    - Not supported by all frameworks
  + DELETE
    - Removes an entry from the database
* Example, logging in:
  + POST /api/login
    - username
    - hashed password
  + Get back some local modification representing your account or an error code
* HTTP Error codes
  + 2XX
    - All good
    - 200
      * Generic OK
  + 3XX
    - Redirects
  + 4XX
    - You did something wrong (user)
    - 401 Unauthenticated
      * Not a valid username/pw
    - 403 Unauthorized
      * Authenticated user does not have permission to access/modify this data
    - 404 Resource not found
  + 5XX
    - I did something wrong (server)
* Possible Problems
  + Returning huge lists in json format slows down the entire system
  + Pagination
    - Data can be separated into multiple pages
      * /api/users?page=0[size=50]
        + Or something like that
  + Make sure your data supports pagination
  + Make sure your paging strategy is good
    - Don’t let the user pass in size 500000000 (if you allow them to pass in a size)
  + Sorting
    - Sometimes JS objects can let you sort a webpage object
    - If everything is loaded into the webpage, the user’s browser hangs
    - If you use pagination, sorting doesn’t make sense
    - Sorting should happen on the backend
  + Try to make everything you do be easier on the frontend
    - Don’t get JS-happy
  + You sometimes have to do complicated tasks with only a limited number of REST verbs
    - How do you handle things that aren’t nouns?
    - How do you handle related database objects?
    - What if you send a request that takes a very long time to execute?
  + Jobs (a solution)
    - /api/job\_descriptors
      * Like typing in ‘help’ into the command line
    - /api/jobs
      * See what jobs are running (‘ps’)
    - Posting to jobs starts a new job
    - Getting a single job gets the status of that job
  + Database should be indexed
    - Large databases require a full look-through for the service to find the correct entry, indexing helps with fast lookup
* Deletion
  + When something is deleted, you also have to make sure to have a policy about deleting related data.
  + You also may not want to delete data if you want to have some kind of audit trail
    - So you could keep an attribute ‘deleted’
* API Versioning
  + /api/v1/users
  + Supporting multiple versions of an API is very hard
  + You have to do everything on a single database
  + Try to avoid breaking your API, but it’s better than keeping different versions
* Things to think about
  + Discoverability
    - You should be able to visit /api/ and get a page detailing all the places you can visit