API

Data

- We now have a database that stores app data
- Users have to control data
  - Manage their profile/setting
  - Make posts
  - Use a shopping cart
  - etc.
- How should users interact with stored data?

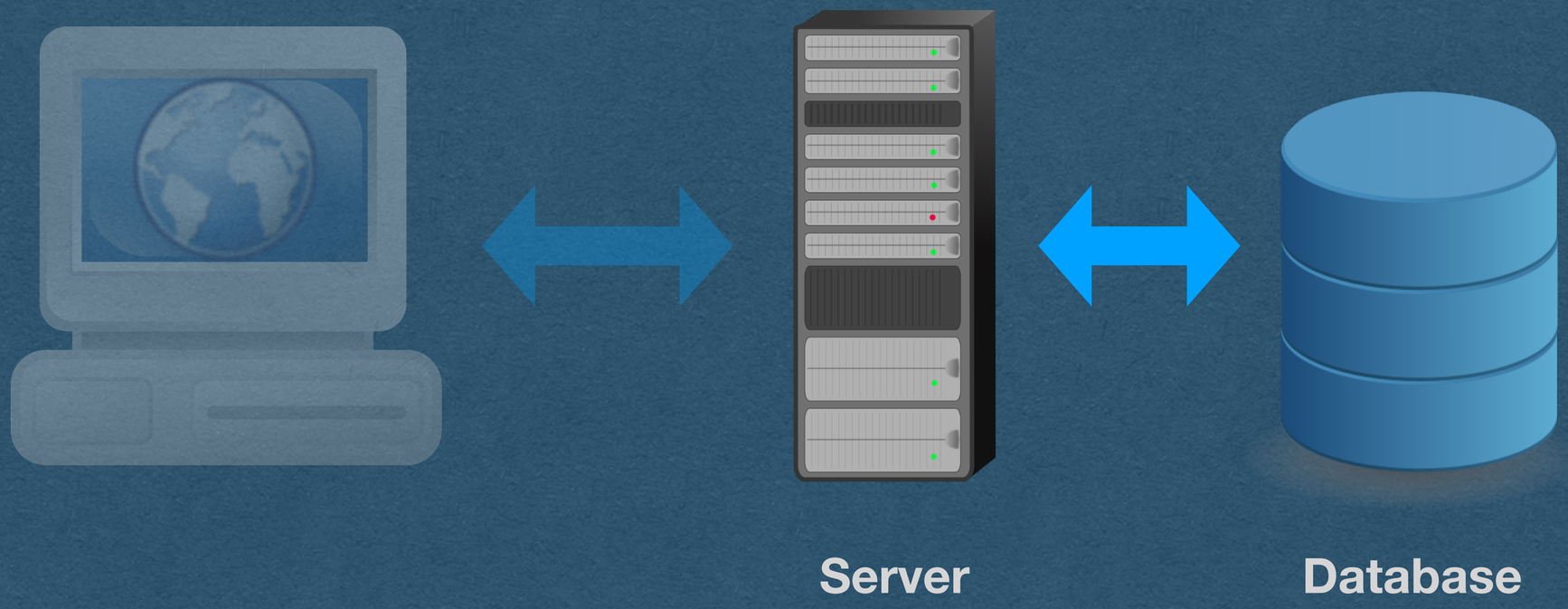
Data

- How do users interact with stored data?

User/Client **Server** **Database**

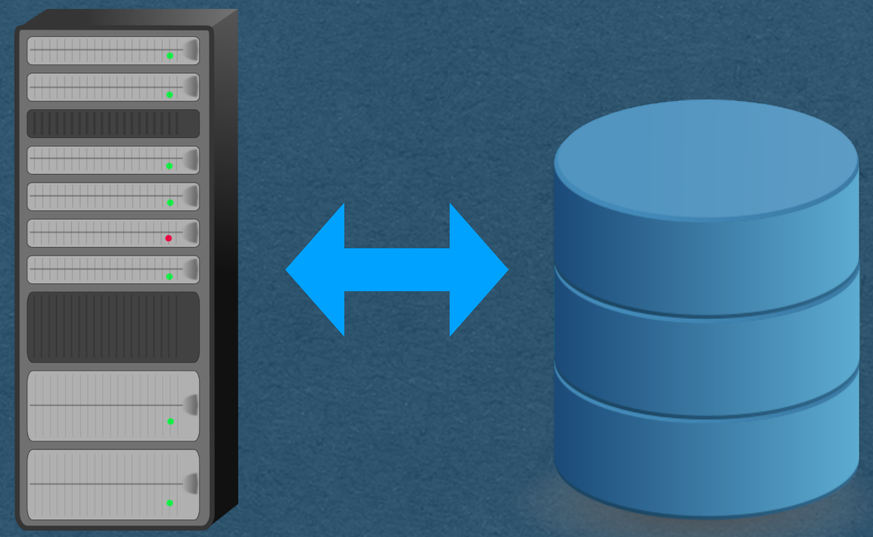
Data

- How does our server interact with stored data?



CRUD

- CRUD is an acronym for the 4 basic operation used to control data
  - **C**reate
  - **R**etrive
  - **U**pdate
  - **D**elete



CRUD - Create

- Create a new record
- INSERT INTO user (?, ?)
- userCollection.insert({"email":"...", "username": "..."})

CRUD - Create

- When a record is created, it should be assigned a unique id
  - This id will be used to identify the created record
  - The id is typically an auto-incrementing integer
    - First record had id==1, second has id==2, etc
  - Let your database generate the ids
    - CREATE TABLE user (id int AUTO_INCREMENT, ...)

CRUD - Create

- MongoDB does not have an auto-increment feature
- You must manage your own ids as integers
  - Maintain a collection that remembers the last used id
  - Increment the id each time a record is created
- Do not use the default "_id"
  - It is not an integer and not allowed on the HW

CRUD - Retrieve/List

- Retrieve all records
  - SELECT * FROM user
  - userCollection.find({})
- Retrieving all records is often called **List**
  - Technically, the acronym is **CRUDL** when list operations are allowed

CRUD - Retrieve

- Retrieve a single existing record
- SELECT * FROM user WHERE id=3
- userCollection.find({"id":3})

CRUD - Update

- Update an existing record

- UPDATE user SET email=?, username=? WHERE id=5

- userCollection.update({"id":5}, {"$set": {"email":"...", "username":"..."}})

CRUD - Update

- Can update all fields except the id
  - The id *can* change, but you should never change it

CRUD - Delete

- Delete an existing record
- DELETE FROM user WHERE id=2
- userCollection.delete({"id":2})

CRUD - Delete

- In practice, common to "soft delete"
  - Don't actually delete the data
  - Instead, mark it as deleted
  - Do not allow retrieve/update operations on data marked as deleted
- Soft deletion allows sys admins to perform additional operations
  - eg. User requests to undo an accidental delete
  - Preserves history

Data

- How do users interact with our server?

User/Client **Server**

HTTP Requests

- GET
  - Request data from the server (Retrieve)
- POST
  - Send data to the server (Create)
- PUT
  - Create or update a resource (Update)
- DELETE
  - Delete a resource (Delete)

HTTP - POST v. PUT

- Both POST and PUT are used to send data to the server
- POST
  - Requires the server to process the data
  - eg. Generating the id for a created record
- PUT
  - Directly create/update a record
  - Server does not process the data of the request
  - Must be idempotent

HTTP - Idempotent

- When multiple identical HTTP requests are sent
  - If the requests are idempotent, they will have the same effect as sending a single request
- The additional requests will not change the data of the API

HTTP - Idempotent

- GET is idempotent
  - Only retrieve data
  - GET should not change the data of the API

HTTP - Idempotent

- PUT and DELETE requests must be **idempotent**
- eg. A second identical PUT doesn't change anything since the change was already made
- eg. Deleting a record twice has the same effect on the API as deleting the record once

HTTP - Idempotent

- POST is **not** idempotent
- Since the server is processing the data, there is no implied idempotent property
- eg. Sending 2 identical POST requests to create a record will result in 2 records being created with different ids

RESTful API

- REST -> **RE**presentational **S**tate **T**ransfer
- We'll use HTTP requests to interact with data

- Designed to simplify the way data is used
- Improve reliability and scalability

RESTful API

- REST is fairly loosely defined (No RFC)
  - Or *loosely understood*

- Typically measured on a spectrum
  - An API can be more/less RESTful
  - The API for the HW is *mostly* RESTful

REST Constraints

- **Client-Server architecture** and **statelessness**
  - Both constraints are implicit when using HTTP

- The use of cookies in a RESTful API would be a violation of statelessness
  - Usually accepted in practice (API tokens)

REST Constraints

- **Cacheablility**
  - Each response must contain caching information
  - Requests should be cached if possible
  - Must avoid stale data from being cached

REST Constraints

- **Layered-System**
  - The API should have the ability to add additional layers between it and the client
  - Ex: Client interacts with a load balancer that delegates to many instances of your API
  - Ex. A Proxy server is added that encrypts all traffic (HTTPS)

REST Constraints

- **Uniform Interface**
  - Resources are defined in the requests
  - The user is given, in a response, enough information to update/delete the resource
  - A request contains all information needed to handle that request
  - The API should be self-contained (No reliance on documentation that cannot be accessed from an API path)

Data

- Users interact with our RESTful API
- API requests correlate to CRUD operations