**Server Client Communication**

Let's talk a little about the client and the server are expected to interact and what that implies for our implementation

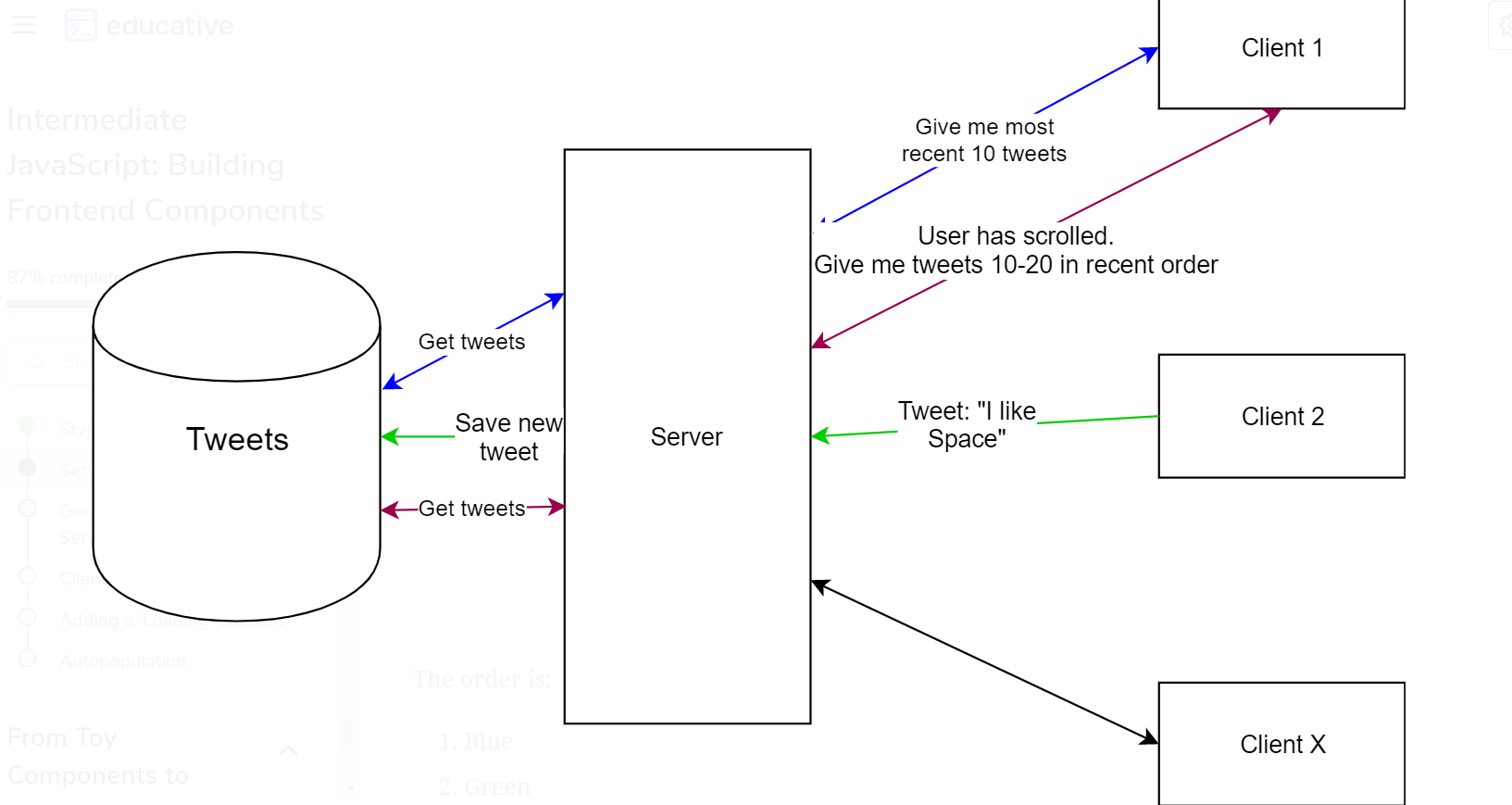
**We'll cover the following**

* + [The protocol](https://www.educative.io/courses/intermediate-javascript/3jOYJ2L9XDp#the-protocol)
  + [Requirements](https://www.educative.io/courses/intermediate-javascript/3jOYJ2L9XDp#requirements)

**The protocol**[#](https://www.educative.io/courses/intermediate-javascript/3jOYJ2L9XDp#the-protocol)

The last time we saw the mock server was when we implemented the autocomplete search component, and it’s going to play an important role in this component. Let’s talk a little more about client-server communication in regards to how the client consumes APIs.

This is a simplified view of the main issue:



The order is:

1. Blue
2. Green
3. Red

An instance of our web app is running on a client, and it has requested to get the ten most recent tweets with “space” hashtag. The user reads them happily while scrolling. During this time, another client has posted a space-related tweet. When the user in client one scrolls to the point where more tweets should be loaded, how does it reconcile that more tweets have come in? If it does a simple lookup of “get 10th to 20th most recent tweets”, it’ll get duplicates that client one had received in its original request.

Client X is just there to represent that we have potentially many more clients posting and requesting.

This is a design decision you have to make. For your particular infinite list, would putting something out of order confuse the user?

## Requirements [#](https://www.educative.io/courses/intermediate-javascript/3jOYJ2L9XDp#requirements)

We’re going to step away from how Twitter does things and impose our own set of requirements:

* Everything in our list must be ordered by the date and time at which the Tweet was posted.​
* We do **not** want a client to receive duplicate tweets
* Tweets are loaded ten at a time

Things to keep in mind:

* If there are fewer​ tweets to serve than requested, the server gives a list of tweets available, if any
* We can keep **any** information about tweets that we want, e.g., the date created, unique ID, etc.

The server must use the information given by the client to figure out the right set of tweets to give it. As a baseline, let’s say each client makes a request with the following:

{  
  pageSize: 10,   
  sortOrder: 'recent'  
}

# Generating Tweets from Server

We need to bring back the server to generate varied data again!

###### We'll cover the following

* + [Revisiting random text server](https://www.educative.io/courses/intermediate-javascript/m220RR1ZNVp#revisiting-random-text-server)
  + [Observations](https://www.educative.io/courses/intermediate-javascript/m220RR1ZNVp#observations)
  + [Unit tests](https://www.educative.io/courses/intermediate-javascript/m220RR1ZNVp#unit-tests)

## Revisiting random text server [#](https://www.educative.io/courses/intermediate-javascript/m220RR1ZNVp#revisiting-random-text-server)

Let’s bring back our mock server from the autocomplete search and see what we can reuse.

// Server

function getRandomString({length}) {

  const characterChoices = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 ";

  const characters = [];

  while (characters.length < length) {

    const randomIndex = Math.floor(Math.random() \* characterChoices.length);

    characters.push(characterChoices[randomIndex]);

  }

  return characters.join('');

}

function getRandomInteger({min, max}) {

  return Math.floor((Math.random() + min) \* (max - min));

}

function generateSuggestion(prefix) {

  const RATIO\_EXACT\_MATCH = 0.3;

  const RATIO\_AUTOCORRECT = 0.1;

  if (Math.random() < RATIO\_AUTOCORRECT) {

    return getRandomString({ length: getRandomInteger({min: 1, max: prefix.length}) })

  }

  if (Math.random() < RATIO\_EXACT\_MATCH) {

    return prefix;

  }

  return prefix + getRandomString({ length: getRandomInteger({min: 1, max: 10}) })

}

function getAutocompleteHandler(data) {

  const MAX\_CHARS = 10;

  const NUM\_AUTOCOMPLETE\_RESULTS = 10;

  const RATIO\_AUXILLERY\_DATA = 0.1;

  if (data.length > MAX\_CHARS) {

    return [];

  }

  const results = [];

  while (results.length < NUM\_AUTOCOMPLETE\_RESULTS) {

    const suggestion = generateSuggestion(data)

    if (results.find(result => result.suggestion === suggestion)) {

      continue;

    }

    if (Math.random() < RATIO\_AUXILLERY\_DATA) {

      for (let i = 0; i < 2; i++) {

        results.push({

          suggestion,

          auxillery: getRandomString({ length: getRandomInteger({min: 5, max: 10}) })

        });

      }

    } else {

      results.push({ suggestion, auxillery: "" });

    }

  }

  return results;

}

const endpoints = {

  "/": {

    "get": () => "hello world"

  },

  "/autocomplete": {

    "get": getAutocompleteHandler

  }

}

// API library

function getFunction(url, data, callback) {

  const domain = url.substring(0, url.indexOf("/"));

  const endpoint = url.substring(url.indexOf("/"), url.length);

  callback(endpoints[endpoint]["get"](data));

}

const api = {

  get: getFunction

};

## Observations [#](https://www.educative.io/courses/intermediate-javascript/m220RR1ZNVp#observations)

* We can reuse getRandomString to generate gibberish tweets
* We can reuse getRandomInteger to generate random IDs
* We’ll need to add an endpoint to allow clients to request tweets
* The handler for such a method will need to accept new parameters
* We need to mock a database
* class Database {
* constructor() {
* this.tweets = [];
* }
* query({lastTweetId, pageSize}) {
* // TODO
* }
* insert(tweet) {
* // TODO
* }
* }
* const database = new Database();
* function getTweetsHandler(data) {
* const pageSize = data.pageSize;
* const sortOrder = data.sortOrder;
* const lastTweetId = data.lastTweetId;
* if (sortOrder !== 'recent') {
* throw new Error('I dont know how to handle that');
* }
* return database.query({lastTweetId, pageSize});
* }
* function postTweetHandler(data) {
* database.insert(data.tweet);
* }
* const endpoints = {
* "/tweets": {
* "get": getTweetsHandler,
* "post": postTweetHandler
* }
* }
* We’re going to ignore the strictly server-side concerns here like querying databases, distrbuted servers, etc. Our database will just be an array, and querying will be for-looping.

## Unit tests [#](https://www.educative.io/courses/intermediate-javascript/m220RR1ZNVp#unit-tests)

* Let’s actually write some unit tests first. Our initial get should return an empty list. We’ll post one tweet, and check that the following get includes that tweet. We’ll post another, and check that the get returns the tweets in order of most recent (based on when the server received it).
* The implementation should be very short. We’re not taking into consideration lastTweetId or pageSize yet.
* You can view the actual tests on the “test” tab.

index.js

class Database {

  constructor() {

    this.tweets = [];

  }

  query({lastTweetId, pageSize}) {

    // TODO

  }

  insert(tweet) {

    // TODO

  }

}

const database = new Database();

function getTweetsHandler(data) {

  const pageSize = data.pageSize;

  const sortOrder = data.sortOrder;

  const lastTweetId = data.lastTweetId;

  if (sortOrder !== 'recent') {

    throw new Error('I dont know how to handle that');

  }

  return database.query({lastTweetId, pageSize});

}

function postTweetHandler(data) {

  database.insert(data.tweet);

}

Tests

var TestResult = function() {

this.succeeded = false;

this.reason = "";

this.input = "";

this.expected\_output = "";

this.actual\_output = "";

}

var executeTests = function(){

var results = [];

const result = new TestResult();

result.input = "getTweetsHandler({pageSize: 10, sortOrder: 'recent'})";

result.expected\_output = '[]';

let output = getTweetsHandler({pageSize: 10, sortOrder: 'recent'});

result.actual\_output = String(output);

if (output instanceof Array && output.length === 0) {

result.succeeded = true;

result.reason = "Succeeded";

} else {

result.succeeded = false;

result.reason = "Expected an empty list";

}

results.push(result);

const firstTweet = {

message: 'I like space',

name: 'Space Man',

handle: 'spaceman'

};

postTweetHandler({tweet: firstTweet});

const afterPostResult = new TestResult();

afterPostResult.input = "getTweetsHandler({pageSize: 10, sortOrder: 'recent'})";

afterPostResult.expected\_output = `[{message: ${firstTweet.message}, name: ${firstTweet.name}, handle: ${firstTweet.handle}}]`;

output = getTweetsHandler({pageSize: 10, sortOrder: 'recent'});

afterPostResult.actual\_output = String(output);

if (output instanceof Array && output.length === 1 && output[0] === firstTweet) {

afterPostResult.succeeded = true;

afterPostResult.reason = "Succeeded";

} else {

afterPostResult.succeeded = false;

afterPostResult.reason = "Expected the first tweet";

}

results.push(afterPostResult);

const secondTweet = {

message: 'I like space 2',

name: 'Space Man 2',

handle: 'spaceman2'

};

postTweetHandler({tweet: secondTweet});

const afterSecondPostResult = new TestResult();

afterSecondPostResult.input = "getTweetsHandler({pageSize: 10, sortOrder: 'recent'})";

afterSecondPostResult.expected\_output = `

[{

message: ${firstTweet.message},

name: ${firstTweet.name},

handle: ${firstTweet.handle}

},

{message: ${secondTweet.message},

name: ${secondTweet.name},

handle: ${secondTweet.handle}

}]

`;

output = getTweetsHandler({pageSize: 10, sortOrder: 'recent'});

afterSecondPostResult.actual\_output = String(output);

if (output instanceof Array && output.length === 1 && output[0] === firstTweet && output[1] === secondTweet) {

afterSecondPostResult.succeeded = true;

afterSecondPostResult.reason = "Succeeded";

} else {

afterSecondPostResult.succeeded = false;

afterSecondPostResult.reason = "Expected the tweets in order";

}

results.push(afterPostResult);

return results;

}

It’s not important for us to go over the full implementation of the server. Let’s assume we now have a working server that’ll handle requests correctly.

# Client Side Hydration

When the user first loads the page, we need to show some list items right away. We'll learn how to in this lesson!

###### We'll cover the following

* + [Server side](https://www.educative.io/courses/intermediate-javascript/7AzwNx50kM1#server-side)
  + [Fetching and rendering tweets](https://www.educative.io/courses/intermediate-javascript/7AzwNx50kM1#fetching-and-rendering-tweets)
  + [Line breaking long tweets](https://www.educative.io/courses/intermediate-javascript/7AzwNx50kM1#line-breaking-long-tweets)

## Server side [#](https://www.educative.io/courses/intermediate-javascript/7AzwNx50kM1#server-side)

We first want to get to a point where we have our initial set of data. As soon as the window loads, let’s pull data from the server and display ​it as a series of tweets.

To know if we’ve succeeded, the server should have some data.

We’ll write a short script to do so:

function loadTestData() {

  const sampleData = [];

  const sampleDataSize = 20;

  for (let i = 0; i < sampleDataSize; i++) {

    const message = getRandomString({

      length: getRandomInteger({min: 10, max: 150}),

      includeSpaces: true

    });

    const firstName = getRandomString({

      length: getRandomInteger({min: 3, max: 7}),

      includeSpaces: false

    });

    const lastName = getRandomString({

      length: getRandomInteger({min: 3, max: 7}),

      includeSpaces: false

    });

    const handle = '@' + getRandomString({

      length: getRandomInteger({min: 4, max: 8}),

      includeSpaces: false

    });

    sampleData.push({

      tweet: {

        name: `${firstName} ${lastName}`,

        message, handle

      }

    });

  }

  for (const data of sampleData) {

    // Do nothing with result

    api.post(HOST + 'tweets', data, () => {});

  }

}

Hopefully, this gives a good set of data. I’ve changed the getRandomString function slightly to include spaces only if specified​, since we don’t want things like handles to have spaces. We do want messages to, however.

If you’re interested in the server changes, here they are:

class Database {

  constructor() {

    this.tweets = [];

  }

  query({lastTweetId, pageSize}) {

    if (!lastTweetId) {

      return this.tweets.slice(0, pageSize);

    }

    for (let i = 0; i < this.tweets.length; i++) {

      const currentTweet = this.tweets[i];

      if (currentTweet.id === lastTweetId) {

        return this.tweets.slice(i + 1, i + 1 + pageSize);

      }

    }

    return [];

  }

  insert(tweet) {

    this.tweets.push({

      tweet,

      id: getRandomString({length: 50}),

      timestamp: (new Date()).getTime()

    });

  }

}

The only thing I want you to note is that the insert function adds data to the request instead of putting in the database directly. Namely, we give it a unique ID (this has to be done on the server to avoid conflicts with others) and the current time as received by the server.

## Fetching and rendering tweets [#](https://www.educative.io/courses/intermediate-javascript/7AzwNx50kM1#fetching-and-rendering-tweets)

Now, for the actual hydration on the client, we just call get with a pageSize, and then pass the relevant fields to a template we created.

The hydration looks like this:

const DEFAULT\_PAGE\_SIZE = 5;

const DEFAULT\_SORT\_ORDER = 'recent';

function onNewTweets(data) {

  // TODO create tweet and render it

}

function hydrate() {

  const params = {

    pageSize: DEFAULT\_PAGE\_SIZE,

    sortOrder: DEFAULT\_SORT\_ORDER

  }

  api.get(HOST + 'tweets', params, onNewTweets);

}

loadTestData();

hydrate();

Pretty simple.

As for the template, we have to move that whole chunk of HTML to JavaScript to populate it dynamically. Some people are extremely opposed to this trend in the web – and it is a trend now. One of the biggest complaints about a popular framework, AngularJS, was that it coupled the HTML and JavaScript too much. React went one step further and just had all the HTML reside within JavaScript files. So many websites these days are dynamically populated that there’s little static HTML on them (maybe just the footer with links to “about,” “career,” etc.​), in which case it can make sense for convenient templating.

In any case, I don’t want to take up space with the template in the JavaScript file. Just know that there’s a function createTweet that returns a string, which represents the HTML we wrote earlier with the values set to the parameters.

function createTweet({name, handle, message}) {

  const template = `

    <div class="tweet">

      ...

      <div class="tweet\_\_main\_\_message">

        ${message}

      </div>

      ...

    </div>

  `;

  return template;

}

## Line breaking long tweets [#](https://www.educative.io/courses/intermediate-javascript/7AzwNx50kM1#line-breaking-long-tweets)

So, I’m not sure about what you see, but if you refresh enough, you’ll get long outputs and the component styling breaks. We didn’t rigorously test our CSS when we first applied styles – but that’s okay. That’s what generated test data is for! It looks like we’re not breaking text that’s too long. Thankfully, we have an easy fix for this!

# Adding a 'Loading' Element

Showing the rest of the list items -- infinitely. (Or almost)

###### We'll cover the following

* + [Changelog](https://www.educative.io/courses/intermediate-javascript/qVAr5AzMm6y#changelog)
  + [Client-side state handling](https://www.educative.io/courses/intermediate-javascript/qVAr5AzMm6y#client-side-state-handling)

## Changelog [#](https://www.educative.io/courses/intermediate-javascript/qVAr5AzMm6y#changelog)

The next thing we need to do is to fetch more tweets dynamically. This will involve a few different changes:

* We need to make some HTML elements that shows a loading state. To simplify our use case, we will once again use the beautiful stock dog instead of a spinning animation.
* We need to introduce a delay into our server, otherwise we can’t test the loading state. We’ll add a two-second pause.
* We need to detect when to load. There are global variables that let us access the user’s current scroll position at any given time. We can fetch more when it’s near the bottom of the page.
* We need to keep track of the last tweet ID so that we can start using it in subsequent calls.

## Client-side state handling [#](https://www.educative.io/courses/intermediate-javascript/qVAr5AzMm6y#client-side-state-handling)

Our API is already set up to handle asynchronicity with callbacks, so the introduction of a delay can be implemented with minimal changes:

function getFunction(url, data, callback) {  
  const domain = url.substring(0, url.indexOf("/"));  
  const endpoint = url.substring(url.indexOf("/"), url.length);  
   
  setTimeout(() => callback(endpoints[endpoint]["get"](data)), 2000);  
}

During the time in which the client has made the request, ​but the callback that handles the response has not yet been invoked, we should show the user something to let them know that more is coming.

Our component will now have to keep state. When the API request is made, the state is immediately set to ‘loading,’ and the loading element should show. In the callback function, we reset the state and replace the loading element with the content just fetched.

Because setting state has side effects, we want to have an abstraction like setPending() instead of doing both state = 'loading' and addLoadingElement(). That way, everywhere that we set the pending state, we just need to call one function, and any changes to actions that happen upon the pending state are restricted to one place.

const States = {

  PENDING: 'pending',

  READY: 'ready'

};

let componentState = States.READY;

const loadingElement = document.createElement('div');

// Give it the same style

loadingElement.classList.add('tweet');

loadingElement.innerHTML = `

  Here I am... Loading...

  <img class="loading\_\_image" src="http://educative.io/udata/1m5lkL7p9Q0/dog.jpeg" />

`;

function setPending() {

  componentState = States.PENDING;

  document.body.appendChild(loadingElement);

}

function setReady() {

  componentState = States.READY;

  document.body.removeChild(loadingElement);

}

The states aren’t exactly being used yet – but I know they’ll be useful!

Here’s our component so far. It’s got a slight delay at the beginning, during which time it shows the loading element. If you missed it, just press the “HTML” tab and press back to “Output.”

# Autopopulation

In this lesson we learn how to dynamically load more tweets.

###### We'll cover the following

* + [Scroll action listener](https://www.educative.io/courses/intermediate-javascript/gxxDKQExRD3#scroll-action-listener)

## Scroll action listener [#](https://www.educative.io/courses/intermediate-javascript/gxxDKQExRD3#scroll-action-listener)

​Let’s add an event listener to the scroll action, and when it reaches near the bottom of the page (say, 100px above), we’ll use that as a trigger to load. Here’s where the states will be used. If the state is in pending, we know not to trigger another request.

function onScroll(event) {

  const scrolledTo = window.innerHeight + window.pageYOffset;

  const scrollLimit = document.body.offsetHeight;

  const scrollThreshold = 30;

​

  if (scrollLimit - scrolledTo <= scrollThreshold) {

    // TODO update

  }

}

​

window.addEventListener('scroll', onScroll);

We can’t quite yet do the update, since we don’t have the lastTweetId.

In the loop for processing API responses, we’ll add this line.

lastTweetId = tweetResponse.id;

Processing for new tweets now needs to be additive, so we switch

document.body.innerHTML = tweetsHTML;

to

document.body.innerHTML += tweetsHTML;

And if the component is pending, we do nothing. This gives us an onScroll function that looks like this:

function onScroll(event) {

  if (isComponentPending()) {

    return;

  }

  const scrolledTo = window.innerHeight + window.pageYOffset;

  const scrollLimit = document.body.offsetHeight;

  const scrollThreshold = 30;

​

  if (scrollLimit - scrolledTo <= scrollThreshold) {

    const params = {

      pageSize: DEFAULT\_PAGE\_SIZE,

      sortOrder: DEFAULT\_SORT\_ORDER,

      lastTweetId

    }

    api.get(HOST + 'tweets', params, onNewTweets);

    setPending();

  }

}