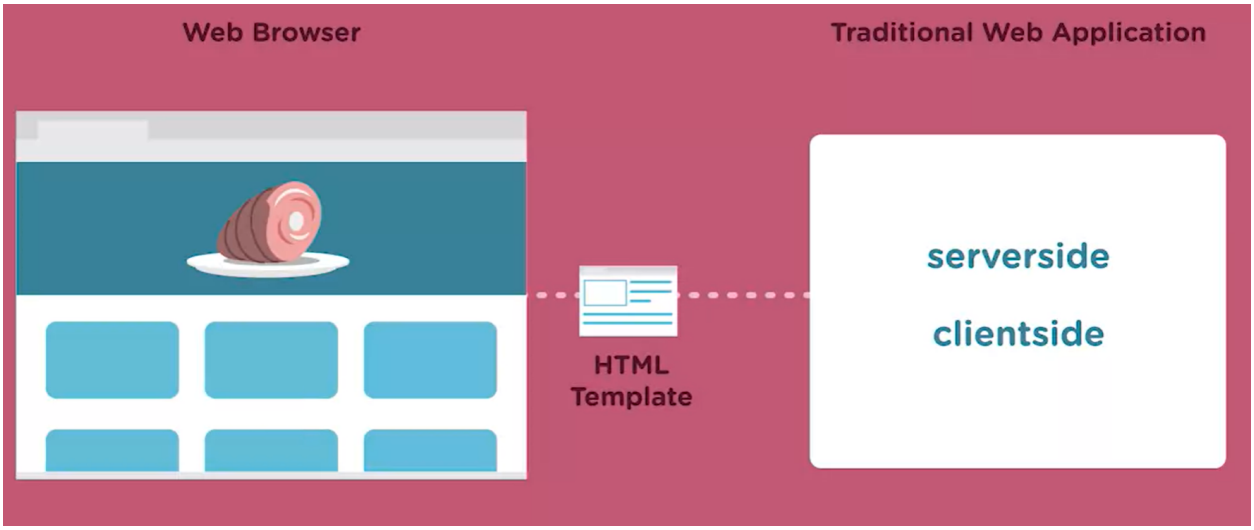
Introduction to REST APIs

APIs are one of the most commonly used interfaces for sharing information across both internal products and third party data sources, such as Twitter and Google. Understanding the fundamentals will give you a strong foundation before getting started with language specific tools.

REST is really just another layer on top of HTTP. When you build a website or app, you´re building a user interface for your app's logic and data model. The point of an API though, is not to create a traditional user interface, but to provide a programmatic interface. The major difference is that the burden of creating the interface is on the users of the API and not the creator.

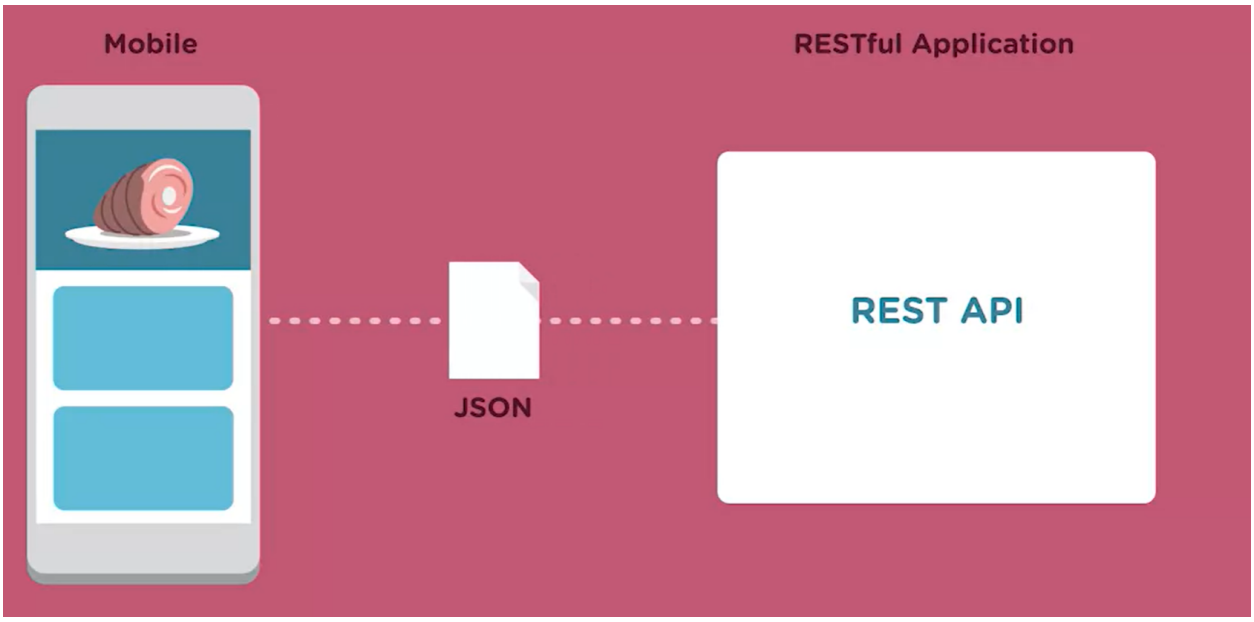
API stands for Application Programming Interface which is a way of talking about code that makes it easier for things outside of an application to interact with that application. Quite often, the code is running on a server, like a Java Virtual Machine, and we need to talk to it from somewhere outside.

Maybe we have a server running for handling the scores of our game from all across the globe and we want to be able to update those scores, whether our players are using a PC, a mobile device, or a tablet. We could write some sort of bridge or connection layer, but we might have to do that for every new device. Building a REST API that works with any external client, anywhere on the Internet, saves us a lot of headaches.



Traditional web applications handle both serverside and clientside concerns. Say you build a web application to keep a record of your favorite recipes. To view a certain recipe, you'd click on that recipe's URL, and your browser would request that recipe from a server. A traditional server side application would respond to that request by finding the recipe data in a database, assembling that data into HTML templates and ending that HTML back to the browser to be displayed.

Seems reasonable but what if you wanted to use that same recipe information to build a mobile app or an entirely different application? That's where REST API comes in, when you request a specific recipe in a RESTful application, the application responds only with the recipe data, typically in the form of JSON.



Sending JSON data rather than HTML means the back end only has to be built once. While any number of front end applications can consume and display the data. When designing an application this way, you can manipulate the same data in endless ways.

REST stands for Representational State Transfer. The web is, by design, stateless. This means that every request that you make to a website is like meeting that site for the first time. Imagine if your friends were like this. You'd have to introduce yourself each time you spoke in a conversation. REST puts all of the work of remembering state on the client, which is your computer program.

Example:

https://randomuser.me/

https://randomuser.me/api/

https://api.randomuser.me/

Endpoints

After each request, the server forgets your client entirely. In fact, you might not even be talking to the same server each time. Your client though, can, and will hold on, to whatever state information it needs, like authentication keys or previous endpoints.

In a REST API, we have these somethings called resources. A resource usually it's a model in our application. If you're holding onto players' scores for your game, a player would be a resource, and so would a score. You might have other things too, like a match if they're playing against other players, or a seed if your game levels were randomly generated. These resources are things we want to be able to retrieve, create, or modify through our API.

We do that retrieving, creating, and modifying, and even deleting, at specific URLs which are called endpoints. Endpoints represent either a single record or a collection of records. For example, these two endpoints represent a collection of games and a single game.

```
/api/v1/games
/api/v1/games/1234
```

Examples:

https://api.spacexdata.com/v3

https://api.spacexdata.com/v3/launches

In RESTful API design, there are actions you're going to take on resources. But instead of being in your URL, they're represented by the type of request the client makes to the API. We have four main verbs, or HTTP methods, that we use for Rest APIs.

HTTP Methods Used in REST API

- GET
- POST
- PUT
- DELETE

GET is used for fetching either a collection of resources, or a single resource. All of our previous URLs would be GET-able.

POST is used to add a new resource to a collection. For example, we wouldn't POST to /players/567 or /games/1234, because they aren't collections. We would, however, post to /players or /games to create a new player or a new game.

PUT is the HTTP method that we use when we want to update a record. We wouldn't use PUT on a collection or list URL.

And finally, we have the DELETE method. Sending a DELETE request to a detail record, a URL for a single record, should delete just that record. Sending delete to an entire collection would delete the whole collection. But that's usually not implemented, with good reason.

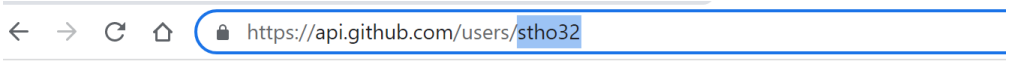
Requests

List of HTTP header fields

The request that a user sends in, can provide a lot more information than just the end points and HTTP can. Different aspects of the request can be used to change the format of the response. The version of the API and more.

Let's open PostMan and take a look at an actual request. In the address bar, much like a web browser, we can enter an endpoint url. Enter a url with an endpoint (e.g., github/userid) to receive an HTML page as a user profile.

Example: https://api.github.com/ or https://api.github.com/users/stho32




```
{
    login: "stho32",
    id: 9053027,
    node_id: "MDQ6VXNlcjkwNTMwMjc=",
    avatar_url: "https://avatars.githubusercontent.com/u/9053027?v=4",
    gravatar_id: "",
    url: "https://api.github.com/users/stho32",
    html_url: "https://github.com/stho32",
    followers_url: "https://api.github.com/users/stho32/followers",
    following_url: "https://api.github.com/users/stho32/following{/other_user}",
    gists_url: "https://api.github.com/users/stho32/gists{/gist_id}",
    starred_url: "https://api.github.com/users/stho32/starred{/owner}{/repo}",
    subscriptions_url: "https://api.github.com/users/stho32/subscriptions",
    organizations_url: "https://api.github.com/users/stho32/orgs",
    repos_url: "https://api.github.com/users/stho32/repos",
    events_url: "https://api.github.com/users/stho32/events{/privacy}",
    received_events_url: "https://api.github.com/users/stho32/received_events",
    type: "User",
    site_admin: false,
    name: null,
    company: null,
    blog: "",
    location: null,
    email: null,
    hireable: null,
    bio: "The most important thing is to be effective in creating value for the customer.",
    twitter_username: null,
    public_repos: 179,
    public_gists: 0,
    followers: 11,
    following: 25,
    created_at: "2014-10-07T15:23:23Z",
    updated_at: "2023-02-26T09:18:34Z"
}
```


We can send more information to the API through parameters. By adding topic as the key and security as the value, we can see these properties are added to the URL.

This is known as a query or **query string**. When we send this request, we will only see repositories.

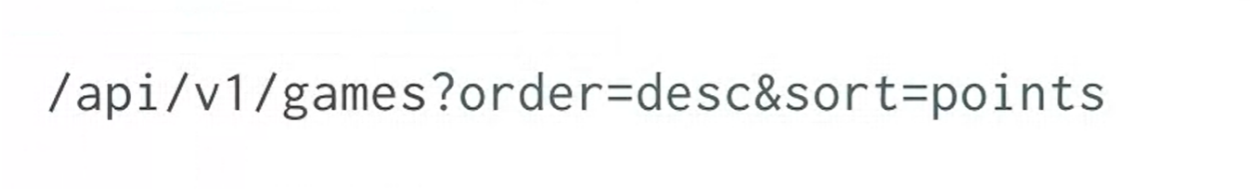
Example: https://api.github.com/search/repositories?q=javascript

This searches github repos that have javascript in their name.

Or another example:

https://api.github.com/search/topics?q=Quant

Everything after the question mark is treated as a set of key and value pairs. For example, this query string gives us two keys.

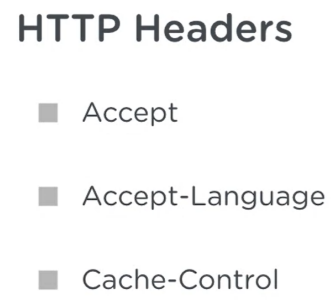


Order and sort, and two values, desc for descending and points. Likely, this would cause us to send back a collection of games sorted by their points in descending order.

Many APIs handle their requests in this way, but it's not always the best approach.

We could also send this information using the HTTP headers. If the API accepts the format in a header, you could send something like this, **Accept as the key and application/json as the value**.

The HTTP spec provides a very thorough list of headers that can and should be accepted by an API. You don't have to accept all of them. let's look at a few that you should probably always pay attention to.



Accept, specifies the file format the requester wants.

Accept-Language specifies the human-readable language, like English, Spanish, or French.

Cache-Control specifies whether the response can be generated from a cache or a quick-to-access memory bank of data or not. You don't have to implement all of the headers.

You don't have to implement any of them. But smarter clients and smarter APIs take advantage of of the http spec, to make transactions cleaner and more explicit.

When consuming a third party API make sure you check the documentation to understand which headers that API accepts.

You might be thinking that your user will send all of their request's content to you in the query string. Most git endpoints accept parameters through the query string where they are URL encoded.

Post requests on the other hand encode the content as either www form URLencoded, or form data and typically send it through as the body of the request. This hides the data from the URL but still makes it available to the API.

You can use Postman to test sending query parameters, headers, and body data to any API you´re using.

Notice the v1 in the URL below:

Responses

After sending a request, we should expect a response.

Like an HTTP request, an HTTP response also has headers. In Postman, we can also view the response Headers.vThe Content-Type header is used to specify what type of response is returned.

This should match the type that was requested. When requesting the JSON format in the URL, or the Accept header, the results are JSON. When we remove this from the header and the URL, the default Content-Type is HTML. In the response Headers, we can see Access-Controls, Cache-Controls, and many more.

Looking at the Status header, we come to another very important part of an API. If you've been on the Internet for very long, you've probably come across a couple of status codes like 200, 500, and the fairly common 404.

These status codes tells us a lot about the state of the bit of content, and we can use them in an API to communicate the state of a request. Generally, status codes are broken up into four chunks, each 100 numbers apart.

Status codes in the 200 to 299 range say that content is good and everything is okay. Some of these reference the fact that an action has been taken but no error has happened. For example, if a client posts a new record to a collection, you'd want to use the 201 status code in your response to let them know that the resource has been created.

Codes in the 300 to 399 range mean the request was understood, but the requested resource is now located somewhere else. These status codes are most often used to perform redirects to new URLs.

The third section, status codes from 400 to 499 are errors typically generated by the client. Maybe the request is wrongly constructed, which would be a 400. Or it's for a resource that no longer exists, 404. Or maybe there's a resource that only be read with GET and no one is allowed to post, put or delete to it. This should return a 405.

The 400 block is about Errors and is the largest of the HTTP status code blocks and it's well worth looking over.

Errors in the 500 to 599 range are all errors on the server's end. The most used of these and the least descriptive is status code 500, which just means that there's some sort of error on the server. It's the equivalent of your server throwing up its hands and just quitting. Look at the rest of the block though. Because a lot of these are way better error messages than just sending back a 500 to your client.

A lot of clients might handle all 500 block errors the same way. But giving them better responses, which will show up in their logs, makes you a better Internet citizen.

List of HTTP status codes

Useful headers, correct status codes, appropriate format, and useful data all combine to make a great HTTP response for an API.

Security

An API should be available all the time otherwise it won't be useful. The first step towards making sure that your API is available for clients is usually caching. A cache is usually a service that runs in memory to hold recently requested results, like a newly created record, or a large data set. This helps to prevent database calls and even costly calculations on your data.

Maybe your data is spread across several databases, or tables in your database, and gathering up all of that information, sorting it, and presenting it to the user, takes several seconds. Putting that final calculated data into a cache means that subsequent lookups only take as long as required for your cache to find and return the right key.

Memcached, and TimesTen are very popular cache, but there are many different pieces of software available to handle this. You may have to experiment to find the one that's right for your API. Also keep in mind that third party APIs that you may be using are probably implementing some sort of caching. This can sometimes cause unexpected results if the caches is not working properly.

The second step on our path to having a resilient API is rate limiting. Rate limiting is a pretty simple idea. Each user is allowed a certain number of requests to your API in a given time period. Once a user exhausts their allotment, they'll have to wait until the timer runs out so they can get more. This helps to prevent users from just flooding you with requests. And also helps to prevent distributed denial of service or DDOS attacks.

And out final step is authentication. It's a little hard to rate limit users when you don't have any idea which request is from which user. And what if we need to restrict certain information to certain users.

We need some way to verify that a user is who they say they are. Different APIs handle authentication differently. When building an API, how your user gets accounts is up to you, and whatever tools you're using. Some few ways that authentication requests can be handled are described here.

The most common way you'll encounter is the use of API tokens. When setting up an API account, a user is given a token and a secret pair. The user will pass those credentials when making a request to the server. This allows the API's server to verify the communication. The server takes the pair of credentials and checks that they're active, proper users in the database. It's like including a user name and password when you log into a site. Users need to include their token with every request because of the statelessness of HTTP. Which means authentication happens with each request. Most of the time, the token and secret are included as keys in the JSON or XML data that a client will send. But it is possible for them to be included in the authentication headers in the HTTP request.

There are other methods of handling authentication, like cross realm authentication, HTTP Digest and others. But a lot of them will be specific to the API or tools that you're using. Postman, the tool that we've been using to test out our API, should allow you to send any type of authentication request that you encounter.

Coding REST API with Node

We will be using NodeJS platform and Express Framework to develop API.

REST APIs can provide data and content for rich web applications, mobile apps, and other serverside applications, even those applications written in other programming languages.

In this course, we'll use Node and Express to build out a simple API that provides data about famous quotes. When we're finished, users will be able to request famous quotes from our API, as well as add new quotes, edit and delete existing quotes, and request a random quote.

A Simple API

Let's create a simple API that returns a JSON object back.

To follow along: open **S1-1_initial_start** and add the following code in the app.js that ads a GET method to an end node "/greetings":

```
app.get ('/greetings', (req,res) => {
    res.json({greeting: "Hello World!!!"})
});
```

You can test this on localhost:3000/greetings

This is a very simple API. Let's expand on this basic API in the following sections.

Planning out a REST API

It is a good idea to have a plan for the API including the request types, end nodes, and resources. The following code is added to the S1-2_planning_complete files so we have a guideline.

```
// Send a GET request to /quotes to READ a list of quotes
// Send a GET request to quotes/:id to READ (view) a quote
// Send a POST request to /quotes to CREATE a new quote
// Send a PUT request to /quotes/:id to UPDATE (edit) a quote
// Send a DELETE request to /quotes/:id DELETE a quote Il Send a GET request to
/quotes/quote/random to READ (view) a random quote
```

GET a Quote, GET all Quotes

Open S1-3_get_quote_all_quotes_complete folder.

In this practice, let's get the data from the data.json file with the following code:

```
const data = require ('./data.json')
```

The following code is used to GET request all quotes or a quote based on an id.

```
// Send a GET request to /quotes to READ a list of quotes
app.get('/quotes', (req, res)=>{
  res.json(data);
});
// Send a GET request to /quotes/:id to READ(view) a quote
app.get('/quotes/:id', (req, res)=>{
  const quote = data.quotes.find(quote => quote.id == req.params.id);
  res.json(quote);
});
```

Generally, we use a database such as MySQL or MongoDB to manage the data. In this tutorial we didn't use a database. When using a database, normally an ORM is used to manage the queries. In this tutorial, In order to be familiar with how data and queries are managed using a real database, we use **data.json** and **records.js** in this tutorial to practice with building an API. In the future tutorials we will switch to a real database.

To continue with this tutorial, use S22 folder. In the folder, **records.js** file contains functions that we need to manage our data and queries. The functions provided include:

- getQuotes()
- getQuote(id)
- getRandomQuote()
- createQuote(newRecord)
- updateQuote(newQuote)
- deleteQuote(record)

For now, the focus is not on the functions, as we will use databases later.

The first step is to getQuote using the provided functions. The code for getting data is shown below:

```
// Send a GET request to /quotes to READ a list of quotes
app.get('/quotes', (req, res) => {
  const quotes = records.getQuotes();
  res.json(quotes);
});
```

However, this doesn't work as the res.json sends quotes before records are retrieved due to asynchronous nature of **getQuotes** function in node.js. We can address this with **async/await** functions, as shown below:

```
// Send a GET request to /quotes to READ a list of quotes
app.get('/quotes', async (req, res) => {
  const quotes = await records.getQuotes();
  res.json(quotes);
});
```

Refactor the Get One Quote Route

Using, **S2-1_refactor_get_all_complete** file, to perform a query for a specific record with an id, the following code is added:

```
app.get('/quotes/:id', async (req, res)=>{
    const quote = await records.getQuote(req.params.id);
    res.json(quote);
```

Pay attention to the async function setup.

Using API Testers

GET functions is easy to be tested using browsers. However, when we get to other API methods such as POST and DELETE, we need a tool to test. There are software such as Postman, Insomnia, Thunder, and others that can be used for such purpose.

The rest of this document describes POST, PUT, and DELETE and we use a tester software to make sure our code works.

Create a New Quote

To create new records using our api, we use POST method, as shown in the following code:

```
//Send a POST request to /quotes to  CREATE a new quote
app.post('/quotes', async (req, res) => {
  const quote = await records.createQuote({
    quote: req.body.quote,
    author: req.body.author,
  });
  res.json(quote);
});
```

As using a database, an ID is automatically created, we do not need to add a key for ID explicitly. Also, pay attention to async/await.

But this alone won't work. We need to use a middleware on the top to make sure express expects to receive information as JSON. The code is:

```
app.use(express.json());
```

We can test on Postman. Remember, we do not need to add ID, as it is automatically generated.

Add a JSON like below, and see it is added to the **data.json** file.

```
{
      "quote": "quote 1",
      "author": "author 1",
    }
```

Using Try/Catch with Async/Await

To use async and await with Express, we need some way to capture any errors that might occur in our await function, like if something goes wrong with the server when we try to create a new quote.

We can use **JS Try/Catch** block to address this. See an example in S33 folder.

To try, let's add a manual error using:

```
    throw new Error ("something wrong");
```

We can test the error now.

```
app.post('/quotes', async (req,res) =>{

    throw new Error("something went wrong. Error ###")
    try {
        const quote = await records.createQuote({
            quote: req.body.quote,
            author: req.body.author
        });
        res.json(quote);
    } catch(err) {
        res.json({message: err.message});
    }
});
```

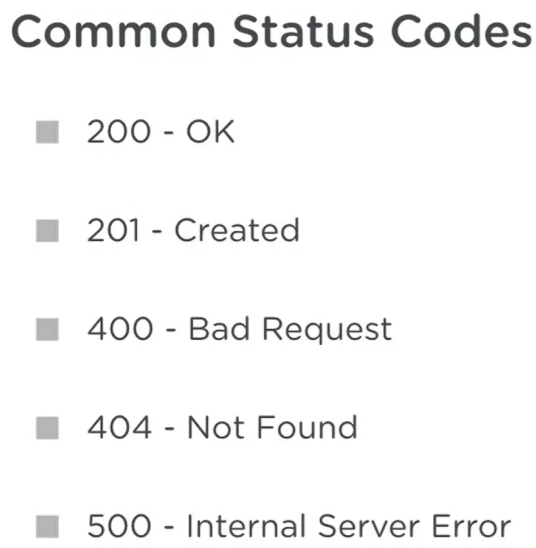
Now delete new Error, and use the rest for try catch. (code in S33)

HTTP Status Codes

Status codes were summarized in the previous sections.

We can use **res.status(???)** to return a specific status code as desired. See **S34** folder.

Some common status codes are shown below:



Edit A Quote

PUT is used for Editing a quote. See **S3-4_edit_quote_complete** folder.

The following code is the main code for updating a record:

```
// Send a PUT request to /quotes/:id to UPDATE (edit) a quote
app.put('/quotes/:id', async(req,res) => {
    try {
        const quote = await records.getQuote(req.params.id);
        if(quote){
            quote.quote = req.body.quote;
            quote.author = req.body.author;

            await records.updateQuote(quote);
            res.status(204).end();
        } else {
            res.status(404).json({message: "Quote Not Found"});
        }

    } catch(err){
        res.status(500).json({message: err.message});
    }
});
```

Delete a Quote

DELETE request is used for deleting a record. See **S3-5_delete_quote_complete** folder.

The main code for the DELETE is as follows:

```
// Send a DELETE request to /quotes/:id DELETE a quote
app.delete("/quotes/:id", async(req,res) => {
    try {
        const quote = await records.getQuote(req.params.id);
        await records.deleteQuote(quote);
        res.status(204).end();
    } catch(err){
        res.status(500).json({ message: err.message });
    }
});
```

Error Handling

The last two middleware represent the logic for handling errors:

```
app.use((req, res, next) => {
    const err = new Error("Not Found");
    err.status = 404;
    next(err);
});

app.use((err, req, res, next) => {
    res.status(err.status || 500);
    res.json({
        error: {
            message: err.message
        }
    })
});
```

S4-1_error_handler_complete folder represent error handling. Pay attention to **next** in delete function. This **next** is required to go to the error handling functions (above) in case the user didn't do any of the request types as well.

```
// Send a DELETE request to /quotes/:id DELETE a quote
app.delete("/quotes/:id", async(req,res, next) => {
    try {
        const quote = await records.getQuote(req.params.id);
        await records.deleteQuote(quote);
        res.status(204).end();
    } catch(err){
        next(err);
    }
});
```

A summarized code is provided in **Complete_Simple_API** folder.

Class Exercise:

Exercise: Add the getRandomQuote to the route “/quotes/randomquote