Axios toolkit for WordPress API: Delete, Update and Create posts

In the last post we covered the basic retrieval methods for posts and pages from a WordPress REST API.

The `updatePost()` function take the post ID that we want to edit as the parameter.

Put and post request are different than the get requests we've used so far. These requests use the body of the request to send the data to the server.

The `put` request is broken into three pieces.

1. The path to the endpoint as a literal string containing the post ID
2. The second parameter is the body of the request. In this case, it's the content we're updating
3. A configuration object. In this case it contains the Authorization header containing our JWT token

Once the request has been submitted we check if the response status is 200 (OK), and if it is, we log the result to console. If not we log an error message to the console. This is not the same type of error that will trigger the catch block on the promise chain... an HTTP 404 is not considered an error so we have to take this type of response into account.

```
async function updatePost(postID = 790634) {
  let theURL = `wp-json/wp/v2/posts/${postID}`; // 1

  await axios
    .put(
      theURL, // 1
      {// 2
        title: 'Change the example title, version 1',
        body: 'Sample post update content to make sure it works',
```

```
      },
      { 'Change the example title, version 1'//3zation: `Bearer ${access_t
        },
      }
    )
    .then((res) => {
      if (res.status === 200) {
        console.log('Post updated successfully');
      } else {
        console.log('There was a problem updating the post');
      }
    })
    .catch((err) => {
      console.log('There was a problem completing the request','Post updat
    });
}
```

The most complicated, to me, of all the functions I've created is the one that will create new posts.

> Like the update request, the post request is broken into three parts:

1. The first parameter is the path for the endpoint we want to use without the domain name
2. The second parameter is the body of the request. In this case, it's the content we're updating
3. A configuration object. In this case it contains the Authorization header containing our JWT token

One other thing that tripped me is that the successful response when creating a post is not 200, but 201 to represent that the object was created.

```
async function createPost() {
  await axios
    .post( // 1
      'wp-json/wp/v2/posts/',
      { 'wp-json/wp/v2/posts/'// 2ish',
        title: 'Creating a post with Axios, version 1',
```

```
        content: 'Sample post create content',
      },
      {
        params: {},'Creating a post with Axios, version 1'ori// 3on: `Bear
        },
      }
    )
    .then((res) => {
      if (res.status === 201) {
        console.log('Post created successfully');
      }
    })
    .catch((err) => {
      console.log('There was a problem completing the request', err.messag
    });
}
'Post created successfully'`Bearer ${access_token}`
```

The final method that we'll cover in this post is delete. The `deletePost` function takes a post ID as a parameter and uses `axios.delete` to remove the associated post from the server.

This method requires a lot of care. Once the post is removed from the database there's no way of getting back so you need to confirm that deleting the post is the intended behavior.

Because this method is a one-time only method we don't provide a default value for the `postID` parameter because it makes no sense.

After the successful removal of a post, subsequent delete requests will return a status code 410 indicating that the resource was removed from the server.

```
async function deletePost(postID) {
  let theURL = `wp-json/wp/v2/posts/${postID}`;
  await axios
    .delete(theURL, {
      headers: {
        Authorization: `Bearer ${access_token}`,
```

```
      },
    })
    .then((res) => {
      if (res.status === 200) {
        console.log('it worked!');
      }
    })
    .catch((err) => {
      console.log('there was a problem', err.message);
    });
}
'it worked!'
```

With the functions in the last two posts we have the beginning of a full CRUD application using WordPress CMS as the backend.

To summarize, the following table illustrates the portions of the CRUD architecture, the corredponding HTTP Method and the function of our API that implements it

CRUD	HTTP Verb	Function
create	POST	createPost()
read	GET	getPosts(numberOfPosts) **(paginated list)** getPost(postID) **(single post)**
update	PUT	updatePost(postID)
delete	DELETE	deletePost()

We could use these methods as the starting point when working with custom post types