

Learning Objectives

- Define how the `fetch` function retrieves data
- Transform data from `fetch` to JSON
- Handle exceptions when using `fetch`
- Define the three most common React lifecycle methods
- Implement state in a component
- Fetch data when the component mounts
- Pass variables and dictionaries from Django to JavaScript
- Understand how to make the table respond to pagination and sorting the table

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the blango repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a blango directory appear in the file tree.

You are now ready for the next assignment.

Fetch

Fetch Intro

`fetch()` is a function built into all modern browsers (released in the past 5 or so years) that performs HTTP requests. It uses a promise-based system to call back when results are received. It takes two arguments, the first is the URL. The second is not mandatory, and contains all the options for the request. For example, to make a GET request, we just need one argument:

```
fetch('/api/v1/posts/')
```

If we wanted to specify a different method, like a POST, and provide some data, that's all part of the options argument:

```
fetch('/api/v1/posts/', {  
  method: 'POST',  
  body: data  
})
```

In this course we're only going to look at using GET, to provide data for our PostTable. If you want to know more about the options available to `fetch()`, and how to upload data, then the [MDN Fetch Documentation](#) is an excellent resource.

Since `fetch()` is promise based, we need to provide a callback function to the `then()` method to deal with the response. Most of the time, we just need to retrieve the JSON data from the response, and this is done by calling the `json()` method on the response object:

```
fetch('/api/v1/posts/').then(response => {  
  return response.json()  
})
```

Is the JSON data now returned from the function? No, the `json()` method actually returns another Promise. Luckily, we can chain promises together so that our source code doesn't get too unwieldy:

```
fetch('/api/v1/posts/').then(response => {
  return response.json()
}).then(data => {
  // do something with data, for example
  console.log(data)
})
```

Before going much further, we need to discuss exceptions in JavaScript.

Exceptions

Like Python, JavaScript has try/catch/finally error handling. In Python, we could do something like this:

```
try:
    raise Exception("Something went wrong")
except TypeError as e:
    print("Got type error", e)
except Exception as e:
    print("Got Exception", e)
finally:
    print("This is always called")
```

The JavaScript equivalent is like this:

```
try {
    throw new Error('Something went wrong')
} catch(e) {
    if (e instanceof TypeError) {
        console.log('Got type error')
        console.log(e)
    } else {
        console.log('Got Exception')
        console.log(e)
    }
} finally {
    console.log('This is always called')
}
```

You can see they are fairly similar, mostly just with different keywords: throw instead of raise, catch instead of except, and different exception classes. Also JavaScript doesn't have the ability to have different handlers for different exceptions. You have just one handler and need to check the exception class to decide how to handle it.

Handling exceptions in promise code is a bit different. Instead of wrapping the promise in a `try/catch` block, we use a new method on the `Promise` class called `catch()`. To this, we'll pass a callback which is called when an exception is raised anywhere along the promise chain.

To add exception handling to a `fetch()` and JSON decode chain, we'd just call `catch()` at the end:

```
fetch('/api/v1/posts/').then(response => {  
  return response.json()  
}).then(data => {  
  // do something with data, for example  
  console.log(data)  
}).catch(e => {  
  console.error(e)  
})
```

This will catch exceptions raised either during the response handling or JSON decode. In its current state, we'll always attempt to decode JSON from the response, even if the response is an error status. If we got a `500 Internal Server Error` response back from the API, we wouldn't know that this had happened until the JSON was decoded and failed. This would hide the real cause from us.

What we should do is throw our own exception after checking the `response.status` value, and we'll get some indication that the server failed, rather than a generic "invalid JSON" message. Let's assume that if the server doesn't return a `200` status that something's gone wrong. Note that Django Rest Framework does return valid JSON in some error cases, like `404 Not Found` or `401 Unauthorized`, so we could handle those cases better. For our simple example though, if the JSON isn't in the format we expect we can't do anything about it, so we'll just go to the error case.

Here's how we could add an exception on a non-200 response:

```

fetch('/api/v1/posts/').then(response => {
  if (response.status !== 200) {
    throw new Error('Invalid status from server: ' +
      response.statusText)
  }

  return response.json()
}).then(data => {
  // do something with data, for example
  console.log(data)
}).catch(e => {
  console.error(e)
})

```

▼ Testing inequality

Note the use of `!==`. This is the type checking equivalent of `!=`, so `200 != '200'` is false, but `200 !== '200'` is true.

The `catch()` handler will be triggered if an exception occurs in either the response handling callback or in the data handling callback - it applies to all callbacks along the promise chain.

Try It Out

To see our good and bad test cases in action, you'll try to fetch three URLs and see the valid or invalid responses that are output.

Copy and paste this code to the start of `blog/static/blog/blog.js`. Don't remove the existing code in the file:

```

['/api/v1/posts/', '', '/abadurl/'].forEach(url => {
  fetch(url).then(response => {
    if (response.status !== 200) {
      throw new Error('Invalid status from server: ' +
        response.statusText)
    }

    return response.json()
  }).then(data => {
    // do something with data, for example
    console.log(data)
  }).catch(e => {
    console.error(e)
  })
})

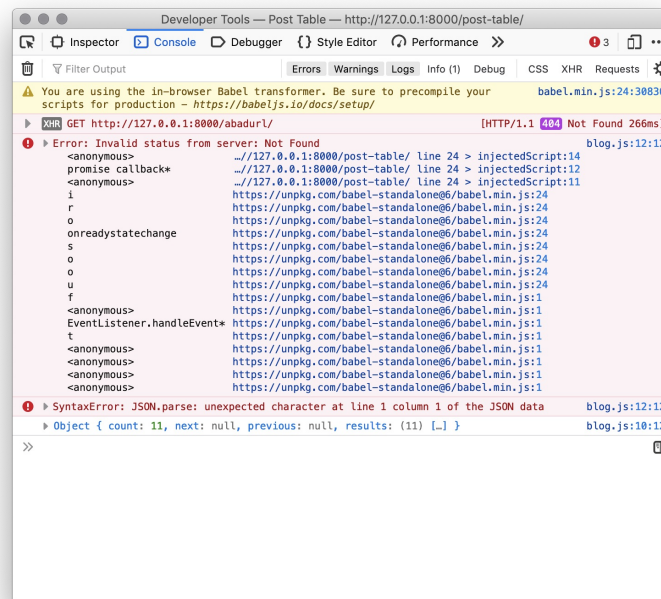
```

This will fetch three URLs:

- `/api/v1/posts/`, which will output valid data to the console.
- `/`, which will return a 200 status code since it exists, but will fail JSON decode.
- `/abadurl/`, which will return a 404 and trigger our exception.

Load up `/post-table/` in your browser and check the browser console. You should see the three errors output, similar to the next screenshot.

[View Blog](#)



fetch errors

Note that because of the asynchronous nature of `fetch()` and promises, the responses may not be returned in the same order in which they were sent. You can see in the screenshot, the third request finished first, the second request finished second, and the first request finished last. Your results may vary.

You can now remove the code you just added from `blog.js`, make sure you leave `PostTable`, `PostRow` and the rest of the React methods untouched.

Next, we'll add data fetching to our `PostTable` component using React component events.

React Component Events

React Component Events

Throughout the lifecycle of a React component, the React library will call a few methods on it as events take place. You can implement these methods on your Component classes and hook into these events. The three most common methods are, `componentDidMount()`, `componentDidUpdate()` and `componentWillUnmount()`.

`componentDidMount`

This takes no arguments, and is called right after the component is inserted into the DOM. You could use this to perform any setup of the component that's not appropriate to do in the `constructor()` method. For example, we're going to use it to start the fetch from the Post API.

`componentDidUpdate`

This method is called when the properties being passed to a component change. This would happen when the parent component re-renders and passes a different value for a property to the child component. For example, if the `PostTable` component re-renders and passes a different `post` property to a `PostRow`, then `componentDidUpdate` will be called on the `PostRow` component.

`componentDidUpdate` is passed three arguments:

- `prevProps`: The props object that the component had prior to being updated, i.e. the new properties are in `this.props` and the ones being used before are in `prevProps`.
- `prevState`: The state object that the component had prior to being updated, i.e. `this.state` is the new state and `prevState` the previous one.
- `snapshot`: This is the result of the `getSnapshotBeforeUpdate()` method, which is invoked just before the rendered component is updated in the DOM. Normally you wouldn't implement `getSnapshotBeforeUpdate()` as you can get the changes by comparing using `prevProps` and `prevState`. For this reason, `snapshot` will be `null`.

You can use `componentDidUpdate()` to check if properties you care about have changed, and then prevent extra function calls or network requests from taking place if they haven't.

For example, we might have a `PostEdit` component that accepts multiple properties, including a `postId`. We only want to fetch the `Post` detail data if the `postId` changes, not if another property changes. We could write a `componentDidUpdate()` to do the check like this:

```
componentDidUpdate (prevProps, prevState) {  
  if (prevProps.postId !== this.props.postId) {  
    this.fetchPostData()  
  }  
}
```

`componentWillUnmount`

This is called directly before the component unmounts (is removed from the DOM). When called, the component is still attached to the DOM. You could use this method to cancel any promises that are outstanding, for example, stopping an ongoing network request.

Other Methods

There are also a few other lifecycle methods that are called, which are rarely used, such as `getSnapshotBeforeUpdate()`, which we already mentioned. If you find that the three methods above don't cater to all your needs, you can read about the others at the [Rarely Used Lifecycle Methods](#) documentation.

Try It Out

Now you'll make your `Post Table` on `Blango` work with real data, by implementing `fetch()` inside the `componentDidMount()` method on your `PostTable` component.

Open `blog/static/blog/blog.js`. The first step is to replace the state in `PostTable` with the real state we want to use, like this:

```
state = {  
  dataLoaded: false,  
  data: null  
}
```

Then we'll implement `componentDidMount()`, with `fetch()` and error handling that we saw earlier. Once we receive the data, from the API, we'll set update the state like this:

```
this.setState({
  dataLoaded: true,
  data: data
})
```

If an error occurs, we'll `catch()` it, log it to the error console, then set `dataLoaded` to `false` and put in placeholder data with an empty results list.

```
console.error(e)
this.setState({
  dataLoaded: true,
  data: {
    results: []
  }
})
```


Putting it all together, add this method to `PostTable`:

```
componentDidMount () {
  fetch('/api/v1/posts/').then(response => {
    if (response.status !== 200) {
      throw new Error('Invalid status from server: ' +
        response.statusText)
    }

    return response.json()
  }).then(data => {
    this.setState({
      dataLoaded: true,
      data: data
    })
  }).catch(e => {
    console.error(e)
    this.setState({
      dataLoaded: true,
      data: {
        results: []
      }
    })
  })
}
```

Refresh `/post-table/` in your browser and you should see all the real `Post` objects that your user has access to.

[View Blog](#)

Title	Image	Tags	Slug	Summary	Link
Django vs Python		django, test	django-vs-python	Which is better, Django or Python? We investigate.	View
Breaking News	-		breaking-news	The latest breaking news story from across the globe.	View
A Real Post	-	example, test	a-real-post	Finally, some real content? We're using a real post to illustrate having multiple posts in the system.	View
This is now a real post!	-	test	yet-another-test-post	Are we taking testing too far?	View
An Example Post	-	django, example	an-example-post	A short example post.	View

post table with real data

`fetch()` will automatically send your browser cookies, so since we have session authentication enabled in DRF, we know the user for the API is automatically set to whichever user is logged into the site, so the list of Posts will be correct for the user.

The PostTable has one final issue that we should fix: the URL is hardcoded. This means we can't filter or go through multiple changes, since this means changing the URL. Let's fix that, and talk about passing variables from Django

Passing Variables from Django

Passing Variables from Django

To pass variables through from Django, we need to just render them in the template. However it's not safe to just render them as is, because Django might not be escaped in the correct way by default. Django provides a couple of template filters to help with this.

For simple values, use the `escapejs` tag. For example, if we had passed the variable `url` to the template context:

```
const url = '{{ url|escapejs }}'
```

The `url` variable in JavaScript now contains the `url` value passed from Python, with single quotes and other characters safely escaped.

If you need to pass a whole Python dictionary to JavaScript, you should use the `json_script` filter. This encodes the Python dictionary using JSON and outputs it to the template, including surrounding `<script>` tags. The filter takes a single argument, which is the `id` attribute to set on the `<script>` tag so that it can be located inside JavaScript code.

For example, if we had the Python dictionary `{"url": "/api/v1/posts/"}`, and we set it in the context variable `config`, we would render it like this:

```
{{ config|json_script:"config-data" }}
```

The output on the page would be this:

```
<script id="config-data" type="application/json">{"url":  
  "/api/v1/posts/"}</script>
```

Then to parse the data back out, we need to fetch the script element using `document.getElementById()`, then parse its `textContent` with `JSON.parse()`:

```
const configElement = document.getElementById('config-data')  
const encodedConfig = configElement.textContent  
const config = JSON.parse(encodedConfig)  
  
// use values from the config object  
const url = config.url
```

Now you can make Blango more dynamic by passing the URL into PostTable as a property.

Try It Out

The journey to pass the URL into the PostTable dynamically starts in the `post_table` view, in `blog/views.py`. Open this up, and import the Django URL `reverse()` function:

```
from django.urls import reverse
```

Then update the `post_table` view to include the reversed `post-list` URL in the context when rendering. The whole function should end up looking like this:

```
def post_table(request):
    return render(
        request, "blog/post-table.html", {"post_list_url":
        reverse("post-list")}
    )
```

Then you'll need to add this variable to the `post-table.html` template. Add another `<script>` element, which must come before the one that includes `blog.js`. It should look like this:

[Open post-table.html](#)

```
<script>
    const postListUrl = '{{ post_list_url|escapejs }}'
</script>
```

This means the `post_list_url` context variable is available for use in the `postListUrl` variable in JavaScript, using the `escapejs` filter to make sure it's safe.

The final two changes to make use of the `postListUrl` variable are done in `blog.js`. Find the line that starts the fetch process:

[Open blog.js](#)

```
fetch('/api/v1/posts/').then(response => {
```

And change it to use `url` from `this.props`:

```
fetch(this.props.url).then(response => {
```

Since we're not actually rendering the `PostTable` like an HTML element we need to pass the `url` property in with the `React.createElement()` call at the bottom of the script file. Make this change:

```
ReactDOM.render(  
  React.createElement(  
    PostTable,  
    {url: postListUrl}  
  ),  
  domContainer  
)
```

Remember that we have access to `postListUrl` as it's a global variable that was defined on the page above where the current script was included.

Refresh `/post-table/` in your browser and you should see no change if everything was done correctly. However now you have a properly dynamic component.

[View Blog](#)

Further Enhancements

Further Enhancements

Right now our React post table doesn't have any advantages over just building a table in plain HTML. The next features to add would be sorting of results by clicking on the table headers, as well as going through the results page by page. Since we have a paginated response, we're only actually showing the first 100 posts in the table.

We won't be covering in detail how to make these changes, as you already have the knowledge to do it if you're interested. But here's a brief overview of a suggested method:

- Create a "pagination" component that renders a previous button, next button, and a list of pages numbers. Using a [Bootstrap Pagination Component](#) would help.
- Use the next, previous and count values of the Post list response to control how the pagination component is rendered. The previous and next buttons would only be shown if the values are not `null` in the response.
- Add `onClick` handler to the table headers to set the result ordering.
- Create a wrapper component that contains the `PostTable` and new pagination component. The wrapper component would do the `fetch()` instead of the `PostTable`, and then pass the relevant values to its child components.
- In order for the child components to trigger a refresh on the parent, they'll need to call parent methods. Parent methods can actually be passed to the child components as properties. Here's a small example, a button that when clicked executes a parent method:

```

class ChildButton extends React.Component {
  render () {
    return <button onClick={ () => {
      this.props.parentCallback('foo') } }>Click Me</button>
  }
}

class ParentContainer extends React.Component {
  aCallback(val) {
    // will log 'foo' when child is clicked
    console.log(val)
  }

  render () {
    return <div>
      <ChildButton parentCallback={ (arg) => {
        this.aCallback(arg) } } />
    </div>
  }
}

```

So if you feel like a challenge and want to make these modifications, you can, but they won't be assessed in any way.

Wrap Up

That brings us to the end of another module and course. We started with how to test Django Rest Framework, followed by some enhancements and ended with JavaScript and a quick intro to the React framework.

In the next Course, we'll start off by looking at how to integrate external APIs into a Django project.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .  
git commit -m "Finish fetch and React hooks"
```

- Push to GitHub:

```
git push
```