
Embrace Opportunity

Software Engineering Day 9

Stat

Today's Overview

- CRUD
- Bring together React and Express for a Full Stack Application

- Cheat Sheet:
  - iX Cheat Sheet
  - iX Cheat Sheet - Day 9

Full Stack

Connecting frontend and backend apps.

Project Requirements - Reminder

- Now that we have built out our application's frontend and backend, we will update our project in order to utilize our created: UI, API and Database.
- Requirements for the project to handle:
    - CRUD (Create, Read, Update, Delete) for:
        - Authentication
        - Categories
        - Blogs

Updating to Full Stack

- Let us focus on connecting the backend and frontend for the blog posts aspect of our application.
- Backend:
  - Routes
    - *createBlog*,
    - *getBlogs*,
    - *getBlogById*,
    - *getBlogsByCategoryId*,
    - *updateBlog*,
    - *deleteBlog*,

Backend - Routes

- POST - /api/blogs/
  - Handles creating new blog
- GET - /api/blogs/
  - Handles getting all blogs
- GET - /api/blogs/:id
  - Handles getting a single blog by ID
- GET - /api/blogs/category/:id
  - Handles getting all blogs by Category ID
- PUT - /api/blogs/:id
  - Handles updating a single blog by ID.
- DELETE - /api/blogs/:id
  - Handles deleting a single blog by ID

Frontend - Service

- We have our routes defined with their corresponding controllers connected to our DB on the backend, from Day 8.
    - Now we will utilize the backend by creating our services to handle all the requests to the backend based on the defined routes.
- Let us focus on our blogs sections.
    - We will create our blogService (*frontend/src/services/blogService.js*)
- We'll define our functions inside our service, then we'll create each async fetch function with the relative request method, to its corresponding route.
    - After which we can update our components to utilize the and response data from the backend.

Frontend - Service

- After creating blogService (*frontend/src/services/blogService.js*)
  - We will define all our functions needed:

```js
const blogService = {
  createBlog,
  fetchBlogs,
  fetchBlogById,
  fetchBlogsByCategoryId,
  fetchBlogsByAuthorId,
  updateBlog,
  deleteBlogsById,
};
```

- Each function will relate to the Backend Routes we defined earlier.

Frontend - Service

- POST - /api/blogs/

```js
const createBlog = async (blog) => {
  const response = await fetch("http://localhost:8000/api/blogs", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: blog,
  });

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};
```

Frontend - Service

- GET - /api/blogs/

```js
const fetchBlogs = async () => {
  const response = await fetch("http://localhost:8000/api/blogs", {
    method: "GET",
    headers: {
      "Content-Type": "application/json",
    },
  });

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};
```

Frontend - Service

- GET - /api/blogs/:id

```js
const fetchBlogById = async (id) => {
  const response = await fetch("http://localhost:8000/api/blogs/" + id, {
    method: "GET",
    headers: {
      "Content-Type": "application/json",
    },
  });

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};
```

Frontend - Service

- GET - /api/blogs/category/:id

```js
const fetchBlogsByCategoryId = async (categoryId) => {
  const response = await fetch(
    "http://localhost:8000/api/blogs/category/" + categoryId,
    {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
      },
    }
  );

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};
```

Frontend - Service

- GET - /api/blogs/author/:id

```js
const fetchBlogsByAuthorId = async (authorId) => {
  const response = await fetch(
    "http://localhost:8000/api/blogs/author/" + authorId,
    {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
      },
    }
  );

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};
```

Frontend - Service

- PUT - /api/blogs/:id

```js
const updateBlog = async (blog) => {
  const response = await fetch(
    "http://localhost:8000/api/blogs/" + blog.get("id"),
    {
      method: "PUT",
      headers: {
        "Content-Type": "application/json",
      },
      body: blog,
    }
  );

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};
```

Frontend - Service

- DELETE - /api/blogs/:id

```js
const deleteBlogsById = async (id) => {
  const response = await fetch("http://localhost:8000/api/blogs/" + id, {
    method: "DELETE",
    headers: {
        "Content-Type": "application/json",
    },
  });

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};
```

Frontend - Service

- Now that all our service functions have been created, linking to our defined routes.
- We'll update our corresponding components to take advantage of this service.
    - Firstly with fetching of blogs:
        - HomePage
            - *fetchBlogs*
        - BlogsPage
            - *fetchBlogsByCategoryId*
        - BlogPage - We will create this component along with the ProfilePage.
            - *fetchBlogById*
        - ProfilePage - This will be handled when we learn JWT authentication.
            - *fetchBlogsByAuthorId*

Frontend - Service

- Secondly with handling modals
  - AddEditBlogModal
    - *createBlog*
    - *updateBlog*
  - DeleteBlogModal
    - *deleteBlogById*
- These two modals will be utilized on:
  - BlogsPage
  - HomePage
  - ProfilePage

Frontend - Components - Loading

- Before utilizing *useEffect* in order to fetch the data required, let us create a Loading component, a Success Toast and Error Toast, to be used during our fetching stages.
- These will be common components used between different pages.
- Create our component *Loading* in *frontend/src/components/Loading/index.jsx*
    - Using the following code snippet:

```
import React from "react";

export default function Loading() {
  return (
    <div className="vh-100 d-flex align-items-center justify-content-center">
      <div className="spinner-border algin-self-center" role="status">
        <span className="visually-hidden">Loading...</span>
      </div>
    </div>
  );
}
```

Frontend - Components - SuccessToast

- For our toast components, you will notice shortly we are utilizing *toast component* from *bootstrap*.
- We'll create our *SuccessToast* in *frontend/src/components/SuccessToast/index.jsx*
- We'll be passing *show, message* and *onClose* props to this component.
    - Create using the following code snippet (Full snippet in vue)
        - We'll first create the component, import the necessary modules, and set up our propTypes.

```
import React, { useEffect, useState } from "react";
import PropTypes from "prop-types";
import { Toast } from "bootstrap";

export default function SuccessToast({ show, message, onClose }) {
  let [successToast, setSuccessToast] = useState(null);

  return (...);

SuccessToast.propTypes = {
  show: PropTypes.bool.isRequired,
  message: PropTypes.string.isRequired,
  onClose: PropTypes.func.isRequired,
};
```

Frontend - Components - SuccessToast

- We'll be using *useEffect* hook in order to subscribe to *show*.
  - This will handle whether to display the modal.

```
useEffect(() => {
  const successEl = document.getElementById("successToast");
  const successToast = successEl
    ? new Toast(successEl, {
        autohide: false,
      })
    : null;

  if (show && successToast) {
    successToast?.show();
    setSuccessToast(successToast);
  }
}, [show]);
```

Frontend - Components - SuccessToast

- Once the above is completed we can create the actual toast element.
- Inside the element we are utilizing the *message* and *onClose* props we will pass in to our component.

```
return (
  <div className="toast-container position-fixed top-0 end-0 p-3">
    <div
      id="successToast"
      className="toast bg-success text-white"
      role="alert"
      aria-live="assertive"
      aria-atomic="true"
    >
      <div className="toast-header">
        <strong className="me-auto">Success</strong>
        <button
          type="button"
          className="btn-close"
          aria-label="Close"
          onClick={() => {
            onClose();
            successToast?.hide();
          }}
        ></button>
      </div>
      <div className="toast-body">{message}</div>
    </div>
  </div>
);
```

Frontend - Components - ErrorToast

- Just like the above *SuccessToast*, we'll follow the same procedure to create our *ErrorToast*.
- Still passing in the props *show, message* and *onClose*.
- We'll create our *ErrorToast* in *frontend/src/components/ErrorToast/index.jsx*
    - First create the component, import the necessary modules, and set up our propTypes.
- Same as *SuccessToast*, we'll utilize *useEffect* in order to display the modal, which can be found on full code snippet.

```
import React, { useEffect, useState } from "react";
import PropTypes from "prop-types";
import { Toast } from "bootstrap";

export default function ErrorToast({ show, message, onClose }) {
  let [errorToast, setErrorToast] = useState(null);

  return (...);
}

ErrorToast.propTypes = {
  show: PropTypes.bool.isRequired,
  message: PropTypes.string.isRequired,
  onClose: PropTypes.func.isRequired,
};
```

Frontend - Components - ErrorToast

- Once the above is completed we can create the actual toast element.
- Inside the element we are utilizing the *message* and *onClose* props we will pass in to our component.

```
return (
  <div className="toast-container position-fixed top-0 end-0 p-3">
    <div
      id="errorToast"
      className="toast bg-danger text-white"
      role="alert"
      aria-live="assertive"
      aria-atomic="true"
    >
      <div className="toast-header">
        <strong className="me-auto">Error</strong>
        <button
          type="button"
          className="btn-close"
          aria-label="Close"
          onClick={() => {
            onClose();
            errorToast?.hide();
          }}
        ></button>
      </div>
      <div className="toast-body">{message}</div>
    </div>
  </div>
);
```

Frontend - Components - HomePage

- We'll add the following code snippet in our HomePage component.
- We'll be fetching all the blog posts, in order to update our "Recent Blog Posts" section of the page.
    - First we'll define our states necessary for operation:

```
const [blogs, setBlogs] = useState([]);
const [categories, setCategories] = useState([]);
const [isError, setIsError] = useState(false);
const [isSuccess, setIsSuccess] = useState(false);
const [isLoading, setIsLoading] = useState(true);
const [message, setMessage] = useState("");
```

    - We'll also add our loading component:

```
if (isLoadingBlogs) {
  return <Loading />;
}
```

Frontend - Components - HomePage

- Next we'll add our asynchronous fetching, utilizing the services we created before.
- We can see we are utilizing our *blogService.fetchBlogs()* service function.
- With the response data updating our *blogs* and setting our success message that will be sent to the toast.
- It can also be seen based on the response from the backend, whether to call on the *success* or *error* toast.

```
useEffect(() => {
  const fetchData = async () => {
    try {
      setIsLoading(true);
      const blogs = await blogService.fetchBlogs();
      setBlogs(blogs.data.reverse());
      setIsSuccess(true);
      setMessage(blogs.message);
      setIsLoading(false);
    } catch (error) {
      setIsError(true);
      setMessage(error.message);
      setIsLoading(false);
    }
  };
  fetchData();
}, []);
```

Frontend - Component - HomePage

- Before handling the return on the component, we'll create two functions in order to reset the states for handling the success or error toasts, that we will pass as a prop when embedding *SuccessToast* and *ErrorToast*.

```
const resetSuccess = () => {
  setIsSuccess(false);
  setMessage("");
}

const resetError = () => {
  setIsError(false);
  setMessage("");
}
```

Frontend - Component - HomePage

- Lastly we can update our *HomePage* return element to incorporate the *SuccessToast* and *ErrorToast* components.
- And setup in the toast components we are passing the necessary props to those components, that have been set in the above fetch.
- We are also passing in a function for the *onClose* prop which will reset the states tied to success or error.

```
return (
  <>
    <Navbar />
    <div className="container">
      <Heading />
      <SubHeading subHeading={"Recent Blog Posts"} />
      <BlogGrid blogPosts={blogs}></BlogGrid>
      <SubHeading subHeading={"Categories"} />
      <CategoryList categories={categories}></CategoryList>
      <Footer />
    </div>
    <SuccessToast
      show={isSuccess}
      message={message}
      onClose={resetSuccess}
    />
    <ErrorToast
      show={isError}
      message={message}
      onClose={resetError}
    />
  </>
);
```

Frontend - Components - BlogsPage

- We'll follow for our next components, the exact same procedure with updating our HomePage, so please see the full code snippet for each component.
- We'll add the following code snippet to our BlogsPage component.
- We'll be fetching all the blog posts by the selected category (if a category is selected) in order to filter the available blog posts.
- Notice something crucial with our *useEffect* hook.
    - Our dependency array isn't empty anymore, which as we've learnt before, will re-run when the dependency changes.
- We'll see we our utilizing our *blogService.fetchBlogsByCategortId(categoryId)* service function.
    - And we are passing in a prop *categoryId* to our service function.

Frontend - Components - BlogsPage

```
useEffect(() => {
  const fetchData = async () => {
    try {
      setIsLoading(true);
      const blogPosts = await blogService.fetchBlogsByCategoryId(categoryId || null);
      setBlogPosts(blogPosts.data.reverse());
      setIsSuccess(true);
      setMessage(blogPosts.message);
      setIsLoading(false);
    } catch (error) {
      setIsError(true);
      setMessage(error.message || error);
      setIsLoading(false);
    }
  };
  fetchData();
}, [categoryId]);
```

Frontend - Components - BlogPage

- We'll add the following code snippet to our *BlogPage* component, even though we'll create this component along side the *ProfilePage*, we will have a look at the use of our service.
- We'll be fetching the blog post selected in order to display the entire blog post for the user to read.
- Same with our *BlogsPage useEffect* hook.
    - Our dependency array isn't empty, which as we've learnt before, will re-run when the dependency changes.
- We'll see we our utilizing our *blogService.fetchBlogById(blogId)* service function.
    - And we are passing in a prop *blogId* to our service function.

Frontend - Components - BlogPage

```
useEffect(() => {
  const fetchData = async () => {
    try {
      setIsLoading(true);
      const blog = await blogService.fetchBlogById(blogId);
      setBlog(blog.data);
      setIsSuccess(true);
      setMessage(blog.message);
      setIsLoading(false);
    } catch (error) {
      setIsError(true);
      setMessage(error.message || error);
      setIsLoading(false);
    }
  };
  fetchData();
}, [blogId]);
```

Frontend - Components - ProfilePage

- Even though we will create our ProfilePage later in the course once we've learnt about JWT authentication, we will have a look how the service is utilized on the page in regards to blogs.
- We'll be fetching all the blog posts created by the author, to display all their created blogs on their page.
- Same with our *BlogsPage useEffect* hook.
    - Our dependency array isn't empty, which as we've learnt before, will re-run when the dependency changes.
- We'll see we are utilizing our *blogService.fetchBlogByAuhtorId(authorId)* service function.
    - And we are passing in a prop *authorId* to our service function.

Frontend - Components - ProfilePage

```
useEffect(() => {
  const getAuthorBlogs = async () => {
    try {
      setIsLoading(true);
      const blogs = await blogService.fetchBlogsByAuthorId(authorId);
      setBlogs(blogs.data);
      setIsSuccess(true);
      setMessage(blogs.message);
      setIsLoading(false);
    } catch (error) {
      setIsError(true);
      setIsLoading(false);
      setMessage(error.message || error);
    }
  };
  getAuthorBlogs();
}, [authorId]);
```

Frontend - Modals

- Now that we have updated our components to utilize fetching data from the backend.
- We'll create our two modals to Create, Update and Delete Blogs, in order to complete our CRUD operation.
  - *AddEditBlogModal*
    - *frontend/src/components/AddEditBlogModal/index.jsx*
  - *DeleteBlogModal*
    - *frontend/src/components/DeleteBlogModal/index.jsx*
- For our modals, we'll be utilizing the *modal module* from bootstrap.
- Please view the full code snippet for the modals.

Frontend - Modal - Add & Edit

- Firstly let us create our AddEditBlogModal (*frontend/src/components/AddEditBlogModal/index.jsx)*
- Define our component:
    - As you can see we are passing in props: *addBlog*, *editBlog*, *categories*, *createBlog*, *updateBlog*

```
export default function AddEditBlogModal({
  addBlog,
  editBlog,
  categories,
  createBlog,
  updateBlog
  }) {
  const modalEl = document.getElementById("addEditBlogModal");
  const addEditBlogModal = useMemo(() => {
    return modalEl ? new Modal(modalEl) : null;
  }, [modalEl]);

  return (...)
}
```

Frontend - Modal - Add & Edit

- Once the component has been defined, let us create all the necessary state variables with their respective defaults needed throughout the component.

```
const [isError, setIsError] = useState(false);
const [isLoading, setIsLoading] = useState(true);
const [message, setMessage] = useState("");
const [blog, setBlog] = useState({
  image: "",
  title: "",
  description: "",
  categories: [],
  content: [],
  authorId: "",
});
```

Frontend - Modal - Add & Edit

- Due to passing in a prop *editBlog* and *addBlog*, this allows us to use a single modal for two purposes of creating or editing a *blog* post.
- So we'll create a *useEffect* in order to subscribe to *addBlog*, *editBlog* and *addEditBlogModal*.
    - Which will set *blog* state (will populate the form fields) and show the modal.

```
useEffect(() => {
  if (addBlog) {
    setBlog(addBlog);
    addEditBlogModal?.show();
  } else if (editBlog) {
    setBlog(editBlog);
    addEditBlogModal?.show();
  }
}, [addBlog, editBlog, addEditBlogModal]);
```

Frontend - Modal - Add & Edit

- Now we'll build a form in order for users to create or edit the required blog fields.
- Fields required:
  - *id*
  - *title*
  - *description*
  - *categoryIds*
  - *content*
  - *authorId*
- Please see the full code snippet for the return in which we create the form, *blogForm*.
- Note: We will not handle the image field right now, we'll learn about file upload on Day 13, and we won't handle authorId till we learn JWT on Day 11.

Frontend - Modal - Add & Edit

- Once the form has been created, we will add some form validation before submitting.
- We will be validating that the user has selected a category for the blog post.

```
const isFormValid = () => {
  const form = document.getElementById("blogForm");
  const hasCategories = blog?.categories?.length > 0;
  form?.elements[1].setCustomValidity(hasCategories ? "" : "Invalid");
  form?.classList?.add("was-validated");
  return form?.checkValidity() && hasCategories;
};
```

Frontend - Modal - Add & Edit

- We will utilize our two function props that will be create on our *BlogsPage*.
- We'll create our onSubmit action.
  - We'll handle which function to call based on if there is an *editBlog* object or not.
    - This will be passed into the modal when it is embedded.

```
const onSubmit = (e) => {
  e.preventDefault();
  if (isFormValid()) {
    if (editBlog) {
      updateBlog(blog);
    } else {
      createBlog(blog);
    }
    resetBlog();
    addEditBlogModal.hide();
  }
};
```

Frontend - Modal - Add & Edit

- We'll also create an *onClose* function in order to reset *blog* state and hide the modal, and we'll create the function *resetBlog* in order to set *blog* state.

```
const onClose = (e) => {
  e.preventDefault();
  resetBlog();
  addEditBlogModal.hide();
};
```

```
const resetBlog = () => {
  setBlog({
    image: "",
    title: "",
    description: "",
    categories: [],
    content: [],
    authorId: ""
  });
};
```

Frontend - Modal - Add & Edit

- Now that we have our modal created, we'll need to create the states and functions on our *BlogsPage* component, in order to pass them as props.
- We'll create our two asynchronous functions that will utilize either our *blogService.createBlog(blog)* or *blogService.updateBlog(blog)* service function.
    - For both we are passing in a prop *blog* to the service functions.

```
const createBlog = async (blog) => {
    try {
      setIsLoading(true);
      const blogRes = await blogService.createBlog(blog);
      setBlog(blogRes.data);
      setIsLoading(false);
    } catch (error) {
      setIsError(true);
      setMessage(error.message || error);
      setIsLoading(false);
    }
};
```

```
const updateBlog = async (blog) => {
  try {
    setIsLoading(true);
    const blogRes = await blogService.updateBlog(blog);
    setBlog(blogRes.data);
    setIsLoading(false);
  } catch (error) {
    setIsError(true);
    setMessage(error.message || error);
    setIsLoading(false);
  }
}
```

Frontend - Modal - Add & Edit

- We'll also need to declare our states on BlogsPage:
  - addBlog
  - editBlog

```
export default function BlogsPage() {
  const [addBlog, setAddBlog] = useState();
  const [editBlog, setEditBlog] = useState();

 return(...)
}
```

Frontend - Modal - Add & Edit

- Now that we have our states declared on our page component, we can add functionality to add a blog.
- Therefore from the code snippet, we will be adding a button on our *BlogsPage* in order to *setAddBlog*.
    - This will on press call the *AddEditBlogModal* to show.

```
const onAddBlog = () => {
  setAddBlog({
    image: "",
    title: "",
    description: "",
    categories: [],
    content: [],
    authorId: "",
  });
};

const AddBlog = () => {
  return (
    <div style={{ display: "flex", justifyContent: "space-between" }}>
      <p className="page-subtitle">Blog Posts</p>
      <button
        style={{ margin: "16px" }}
        type="button"
        className="btn btn-outline-secondary"
        onClick={onAddBlog}
      >
        Add Blog Post
      </button>
    </div>
  );
};
```

Frontend - Modal - Add & Edit

- Lastly with all the pieces created, we can embed *onAddBlog* and *AddEditBlogModal* into *BlogsPage*.
    - Passing in the necessary props created above.

```
<AddBlog />
<AddEditBlogModal
  addBlog={addBlog}
  editBlog={editBlog}
  categories={categories}
  createBlog={createBlog}
  updateBlog={updateBlog}
/>
```

Frontend - Modal - Add & Edit

- To be able to utilize the *AddEditBlogModal*, we'll be updating our BlogItem component to incorporate an edit button, this edit button will update our editBlog state in order to trigger our modal to show and populate the form.
- Creating EditButtons in *frontend/src/components/EditButtons/index.jsx*
    - We'll be passing *onEdit* and *onDelete* (we'll handle this later)

```jsx
import React from 'react'

export default function EditButtons({onEdit, onDelete}) {
  return (
    <>
    <button
      style={{
        position: "absolute",
        top: "10px",
        right: "60px",
        border: "none",
        zIndex: 1,
      }}
      type="button"
      className="btn"
      onClick={onEdit}
    >
      <i className="bi bi-pencil-fill"></i>
    </button>
    <button
      style={{
        position: "absolute",
        top: "10px",
        right: "35px",
        border: "none",
        zIndex: 1,
      }}
      type="button"
      className="btn"
      onClick={onDelete}
    >
      <i className="bi bi-trash-fill"></i>
    </button>
  </>
  )
}
```

Frontend - Modal - Add & Edit

- Once the *EditButtons* component has been created, we can update our *BlogItem* to incorporate these buttons.
- *BlogItem* - *frontend/src/component/BlogItem/index.jsx*
  - We create the container component that utilizes the *EditButtons*, and we are passing in our two function props, *setEditBlog* and *setDeleteBlog*.
  - And lastly embedding this component inside our return.
- Full code snippet - These are just the updates to this component.

```
export default function BlogItem({
  ...,
  setEditBlog,
  setDeleteBlog
}) {

  const EditButtonsContainer = () => {
    <EditButtons
      onEdit={() => setEditBlog(blogPost)}
      onDelete={() => setDeleteBlog(blogPost)}
    />
  };

  return (
      //Full code snippet on vue
      <EditButtonsContainer />
  );
}
```

Frontend - Modal - Add & Edit

- You might notice how we are now passing in two more props into the BlogItem component, setEditBlog and setDeleteBlog.
- This means we have to update the entire tree from *BlogsPage* where we originally call *BlogList* to also pass in these props.
    - This should already give you an idea as to why state management libraries like Redux exist.

```
export default function BlogList({ blogPosts, setEditBlog, setDeleteBlog }) {

 return (
  <BlogItem
    ...
    setEditBlog={setEditBlog}
    setDeleteBlog={setDeleteBlog}
  />
 );

BlogList.prototype = {
    blogPosts: PropTypes.array.isRequired,
    setEditBlog: PropTypes.func.isRequired,
    setDeleteBlog: PropTypes.func.isRequired,
};
```

Frontend - Modal - Add & Edit

- And finally the last step is to update *BlogsPage* component to pass in the additional props necessary from *BlogList* and *BlogItem*.

```
export default function BlogsPage() {
  return (
    ...
    <BlogList
      blogPosts={blogs}
      onEdit={setEditBlog}
      onDelete={setDeleteBlog}
    />
  );
}
```

Frontend - Modal - Delete

- Finally our last modal will be to delete a selected blog post.
- Firstly we'll create our *DeleteBlogModal*.
  - *frontend/src/components/DeleteBlogModal/index.jsx*
  - We'll be passing *deleteBlog* and *removeBlog* as our props

```
export default function DeleteBlogModal({
    deleteBlog,
    removeBlog,
}) {
  const modalEl = document.getElementById("deleteBlogModal");
  const deleteBlogModal = useMemo(() => {
    return modalEl ? new Modal(modalEl) : null;
  }, [modalEl]);

  useEffect(() => {
    if (deleteBlog) {
      setBlog(deleteBlog);
      deleteBlogModal?.show();
    }
  }, [deleteBlog, deleteBlogModal]);

  return (...)
}
```

Frontend - Modal - Delete

- Once we have defined our *DeleteBlogModal* component we can create our action calls, *onClose* and *onDelete*.

```
const onClose = (e) => {
  e.preventDefault();
  resetBlog();
  deleteBlogModal?.hide();
};

const onDelete = (e) => {
  e.preventDefault();
  removeBlog(deleteBlog);
  resetBlog();
  deleteBlogModal?.hide();
};
```

```
const resetBlog = () => {
  setBlog({
    image: "",
    title: "",
    description: "",
    categories: [],
    content: [],
    authorId: ""
  });
};
```

Frontend - Modal - Delete

- We had already added the delete functionality to our *EditButtons* earlier.
- So lastly on our *BlogsPage*:
  - Declare *deleteBlog* state
  - Create *removeBlog* function
  - Embed *DeleteBlogModal*.
    - Passing in our required props.

```
import DeleteBlogModal from "../../components/DeleteBlogModal";

export default function BlogsPage() {
  const [deleteBlog, setDeleteBlog] = useState();

  const removeBlog = async (blog) => {
    try {
      setIsLoading(true);
      const blogRes = await blogService.deleteBlogsById(blog.id);
      setBlog(null);
      setIsLoading(false);
    } catch (error) {
      setIsError(true);
      setMessage(error.message || error);
      setIsLoading(false);
    }
  }

  return (
    ...
    <DeleteBlogModal
      deleteBlog={deleteBlog}
      removeBlog={removeBlog}
    />
  );
}
```

Homework

Apply what we have learned

Homework

- Utilizing what we have learnt in class and following the same update path for the services and updating pages to utilize the service
- Create categoryService.
  - Created in the new directory *frontend/src/services/*
- Requirements:
  - Functions inside *categoryService*:
    - *createCategory*
    - *fetchCategories*
    - *updateCategory*
    - *deleteCategoryById*
  - Pages:
    - *BlogsPage*
    - *CategoriesPage*
    - *HomePage*

Next Class

Course Recap & Succeeding In Your Internship