main.tsx

```tsx
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import "bootstrap/dist/css/bootstrap.css";
import "./index.css";

ReactDOM.createRoot(document.getElementById("root") as HTMLElement).render(
 <React.StrictMode>
   <App />
 </React.StrictMode>
);
```

index.css

```css
html,
body {
 margin: 0 auto;
 background: #333333;
 font-family: monospace;
 color: white;
 align-items: center;
 text-align: center;
 padding: 30px;
}
```

index.html

```html
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="UTF-8" />
   <link rel="icon" type="image/svg+xml" href="/vite.svg" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>Vite + React + TS</title>
 </head>
 <body>
   <div id="root"></div>
   <script type="module" src="/src/main.tsx"></script>
 </body>
</html>
```

ChatServer.tsx

```tsx
import React, { useEffect, useState } from "react";

const connect = () => console.log("connecting...");
const disconnect = () => console.log("disconnecting...");

function ChatServer() {
 useEffect(() => {
   connect();
   return () => disconnect();                                               /* - This is a function for cleaning up.
 });                                                                           - These should always clean up close
 return <div>ChatServer</div>;
}

export default ChatServer;
```

down what the effect was doing
      - see mounting and unmounting in the notes for this section
      */

ProductList.tsx

```
import React, { useEffect, useState } from "react";

const ProductList = ({ category }: { category: string }) => {
 const [products, setProducts] = useState<string[]>([]);

 useEffect(() => {
   console.log("Fetching products in ", category);

   setProducts(["Clothing", "Household", "Pets"]);
 }, [category]);

 return <div>ProductList</div>;
};

export default ProductList;
```

/* Without the second argument (array of dependencies), results are constant rendering loop. These dependencies can be one or more variables, props or state, upon which the effect will be dependent.
   - If any of the dependency values changes, React will re-run and re-render the effect.
   - Empty array means the effect is not dependent on any values.
   - This tells React to only run this effect the first time the component is rendered */

FetchingData.tsx

```
import React, { useEffect, useState } from "react";
import axios, { CanceledError } from "axios";

interface User {
 id: number;
 name: string;
}

function FetchingData() {
 const [users, setUsers] = useState<User[]>([]);
 const [error, setError] = useState("");
 const [isLoading, setIsLoading] = useState(false);

 useEffect(() => {
   // standard in browsers to allow for canceling requests
   const controller = new AbortController();

   // just before calling server, set isLoading to true
   setIsLoading(true);
   axios
     .get<User[]>("https://jsonplaceholder.typicode.com/users", {
       signal: controller.signal,
     })
     .then((response) => {
       setUsers(response.data);

       setIsLoading(false);
     })
```

```
/* Dummy JSON data:
https://jsonplaceholder.typicode.com/
- axios.get() returns a promise
- .then() takes the response from the promise
- the response object returns not only the
data, but other information
about the data */

/* only specifying the properties of users
returned that we will use
   - then we specify in axios.get the type of
data we want to get back,
     which will be formatted using this
interface: <User[]> (below)
   - empty array for the dependencies of the
effect prevents constant render */

       //hide loader when promise is settled,
either resolved or rejected

     /* catch used in this way will console
log errors that arise,
    .catch((error) => {
      // avoids unnecessary error cancel clutter on page
      if (error instanceof CanceledError) return;
      setError(error.message);
      // if something goes wrong and promise is rejected, hide loader
      setIsLoading(false);
    });

  return () => controller.abort();
}, []);

const deleteUser = (user: User) => {
  const originalUsers = [...users];
  setUsers(users.filter((u) => u.id !== user.id));

  // Updating server to persist changes, which returns a promise
  axios
    .delete("https://jsonplaceholder.typicode.com/users/" + user.id)
    .catch((error) => {
      // if there is an error, set the users back to the original users
      setError(error.message);
      setUsers(originalUsers);
    });
};

const addUser = () => {
  const originalUsers = [...users];
```

```
        ex: if the URL above is wrong or
something goes wrong in transmission
        - it will return an Axios error object
with information */

   // Optimistic update, calling the UI first
and then the server to persist changes
   // Updating by filtering users and getting
all users that are not the deleted id

   // IRL, these would be based on values from
a form.
   // In this case, values are hard-coded for
the purpose of the lesson focus

     /* if the call to the server is
```

```
  const newUser = { id: 0, name: "Evan Marie" };
  setUsers([newUser, ...users]);
  // call server to set changes
  axios
   .post("https://jsonplaceholder.typicode.com/users", newUser)
   .then(({ data: savedUser }) => [savedUser, ...users])
   // if an error occurs, return users to originalUsers
   .catch((error) => {
    setError(error.message);
    setUsers(originalUsers);
   });
 };

 const updateUser = (user: User) => {
  const originalUsers = [...users];
  // for now, update will just add an exclamation point to the end of the
username
  const updatedUser = { ...user, name: user.name + " - updated" };
  /* if the id of the current user (u) matches the id of the user passed,
return the updated user, otherwise return the same user object */
  setUsers(users.map((u) => (u.id === user.id ? updatedUser : u)));


  axios
   .patch(
    "https://jsonplaceholder.typicode.com/users/" + user.id,
    updatedUser
   )
```

```
successful, refresh list with saved newUser
     - the newUser will have an id generated
on the server
     - getting the data from the server and
setting users to include the newUser
     data, which is included in the body of
the response
     - could be written like this as well,
whichever is more clear
     .then((response) =>
setUsers([response.data, ...users])) */
```

```
  /* Updating on the server :
   .put is used for replacing an object.
.patch is used for updating one or more
   properties of an object. The choice depends
on how the backend is built. */
```

```
 /* the following renders the error message if
there is an error present
    spinner is rendered if isLoading is true
(use Throttling in network tools to sim)
    using Bootstrap ul and li elements
    - d-flex with justify-content-between
spreads the two buttons evenly in their div
```

```
    .catch((error) => {
      setError(error.message);
      setUsers(originalUsers);
    });
};


return (
  <>
    {error && <p className="text-danger">{error}</p>}
    {isLoading && <div className="spinner-border"></div>}
    <button className="btn btn-primary mb-3" onClick={addUser}>
      Add User
    </button>
    <ul className="list-group">
      {users.map((user) => (
        <li
          key={user.id}
          className="list-group-item d-flex justify-content-between"
        >
          {user.name}
          <div>
            <button
              className="btn btn-outline-secondary mx-3"
              onClick={() => updateUser(user)}
            >
              Update
            </button>
```

```
    - usernames are pushed left, and buttons
container is pushed right
    - mx-1 gives a small horizontal margin
between the two buttons
    - updateUser would generally be created
from an input form, but here we are
    simulating that with the button and
hard-coded data. */
```

```
              <button
                className="btn btn-outline-danger"
                onClick={() => deleteUser(user)}
              >
                Delete
              </button>
            </div>
          </li>
        ))}
      </ul>
    </>
  );
}

export default FetchingData;

DETAILED EXPLANATION:
- The component is mounted, and the useState hook is used to initialize the
users, error, and isLoading state variables.
- The useEffect hook is used to define an effect that runs when the component
is first mounted and every time users, error, or isLoading changes.
- The effect creates a new AbortController object, which will allow for
canceling requests if necessary.
- The effect sets isLoading to true.
- The effect makes an HTTP GET request using Axios to retrieve an array of
User objects from the JSONPlaceholder API.
- The get method of Axios is called with two arguments: the URL to the
JSONPlaceholder API and an options object that includes the signal property
```

set to the signal property of the AbortController object. This ensures that the request can be canceled if needed.

- The Axios get method returns a Promise that resolves with the response data, which is an array of User objects.

- If the request is successful, the effect sets users to the response data and sets isLoading to false.

- If there is an error, the effect sets the error state to the error message and sets isLoading to false.

- The effect returns a cleanup function that calls the abort method of the AbortController object to cancel any ongoing requests if the component is unmounted before the request completes.

- The component defines three functions: deleteUser, addUser, and updateUser. deleteUser takes a User object as an argument and creates a copy of the users array using the spread operator.

- It removes the passed user from the copied array using the filter method.

- It sets the users state to the new array.

- It makes an HTTP DELETE request to remove the user from the server using Axios.The URL for the request includes the id of the user to be deleted.

- If there is an error, the function sets the error state to the error message and sets the users state back to the original array.

addUser creates a copy of the users array using the spread operator.

- It creates a new User object with a hard-coded id and name.

- It adds the new user to the beginning of the copied array using the spread operator.

- It sets the users state to the new array.

- It makes an HTTP POST request to add the new user to the server using Axios. The body of the request includes the data for the new user.

- If there is an error, the function sets the error state to the error message

and sets the users state back to the original array.

updateUser takes a User object as an argument and creates a copy of the users array using the spread operator.

- It creates a new User object with an updated name property.

- It replaces the passed user with the updated user using the map method.

- It sets the users state to the new array.

- It makes an HTTP PATCH request to update the user on the server using Axios. The URL for the request includes the id of the user to be updated, and the body of the request includes the data for the updated user.

- If there is an error, the function sets the error state to the error message and sets the users state back to the original array.

- The component's return statement returns a fragment containing:

- A conditional rendering of the error message using the && operator.

- A conditional rendering of a Bootstrap spinner using the && operator.

- A button that calls addUser when clicked.

- A list of User objects, where each User object is displayed in a Bootstrap list-group-item element.

- For each User object, an "Update" button and a "Delete" button are displayed using Bootstrap btn elements.

- The "Update" button calls updateUser with the corresponding User object as an argument.

- The "Delete" button calls deleteUser with the corresponding User object as an argument.

FetchingData_02.tsx

Using await in an async function for error handling - somewhat more cumbersome approach

```
import React, { useEffect, useState } from "react";
import axios, { AxiosError } from "axios";
```

```
interface User {
 id: number;
 name: string;
}

function FetchingData() {
 const [users, setUsers] = useState<User[]>([]);

 const [error, setError] = useState("");

 useEffect(() => {
   const fetchUsers = async () => {
     try {
       const response = await axios.get<User[]>(
         "https://jsonplaceholder.typicode.com/users"
       );
       setUsers(response.data);
     } catch (error) {
       setError((error as AxiosError).message);
     }
   };
   fetchUsers();
 }, []);

 return (
   <>
     {error && <p className="text-danger">{error}</p>}
```

```
      <ul>
        {users.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </>
  );
}

export default FetchingData;
```

FetchingDataServices.tsx

```
import React, { useEffect, useState } from "react";
import { CanceledError } from "../services/api-client";
import userService, { User } from "../services/user-service";

function FetchingDataServices() {
 const [users, setUsers] = useState<User[]>([]);
 const [error, setError] = useState("");
 const [isLoading, setIsLoading] = useState(false);

 useEffect(() => {
   setIsLoading(true);
   const { request, cancel } = userService.getAll<User>();
   request
     .then((response) => {
```

More modular implementation of FetchData

- *all http request functionality is within the user-service.ts file*
- *the user-service file can now be used with other programs*
- *now, this component only handles requests through userService*
- *all interaction with API is also now happening through userService */*

```
      setUsers(response.data);
      setIsLoading(false);
    })

    .catch((error) => {
      if (error instanceof CanceledError) return;
      setError(error.message);
      setIsLoading(false);
    });

  return () => cancel();
}, []);

const deleteUser = (user: User) => {
  const originalUsers = [...users];
  setUsers(users.filter((u) => u.id !== user.id));

  userService.delete(user.id).catch((error) => {
    setError(error.message);
    setUsers(originalUsers);
  });
};

const addUser = () => {
  const originalUsers = [...users];
  const newUser = { id: 0, name: "Evan Marie" };
  setUsers([newUser, ...users]);
  userService
    .add(newUser)
    .then(({ data: savedUser }) => [savedUser, ...users])
    .catch((error) => {
      setError(error.message);
      setUsers(originalUsers);
    });
};

const updateUser = (user: User) => {
  const originalUsers = [...users];
  const updatedUser = { ...user, name: user.name + " - updated" };

  setUsers(users.map((u) => (u.id === user.id ? updatedUser : u)));
  userService.update(updatedUser).catch((error) => {
    setError(error.message);
    setUsers(originalUsers);
  });
};

return (
  <>
    {error && <p className="text-danger">{error}</p>}
    {isLoading && <div className="spinner-border"></div>}
    <button className="btn btn-primary mb-3" onClick={addUser}>
      Add User
    </button>
    <ul className="list-group">
```

```
       {users.map((user) => (
         <li
           key={user.id}
           className="list-group-item d-flex justify-content-between"
         >
           {user.name}
           <div>
             <button
               className="btn btn-outline-secondary mx-3"
               onClick={() => updateUser(user)}
             >
               Update
             </button>
             <button
               className="btn btn-outline-danger"
               onClick={() => deleteUser(user)}
             >
               Delete
             </button>
           </div>
         </li>
       ))}
     </ul>
   </>
 );
}
export default FetchingDataServices;
```

CH_FetchingDataServices.tsx

```tsx
import React, { useEffect, useState } from "react";
import { CanceledError } from "../services/api-client";
import userService, { User } from "../services/user-service";
import useUsers from "../hooks/useUsers";

function CH_FetchingDataServices() {
 const { users, error, isLoading, setUsers, setError } = useUsers();

 const deleteUser = (user: User) => {
   const originalUsers = [...users];
   setUsers(users.filter((u) => u.id !== user.id));

   userService.delete(user.id).catch((error) => {
     setError(error.message);
     setUsers(originalUsers);
   });
 };

 const addUser = () => {
   const originalUsers = [...users];
   const newUser = { id: 0, name: "Evan Marie" };
   setUsers([newUser, ...users]);

   userService
     .add(newUser)
```

```
    .then(({ data: savedUser }) => [savedUser, ...users])
    .catch((error) => {
      setError(error.message);
      setUsers(originalUsers);
    });
};

const updateUser = (user: User) => {
  const originalUsers = [...users];
  const updatedUser = { ...user, name: user.name + " - updated" };

  setUsers(users.map((u) => (u.id === user.id ? updatedUser : u)));
  userService.update(updatedUser).catch((error) => {
    setError(error.message);
    setUsers(originalUsers);
  });
};

return (
  <>
    {error && <p className="text-danger">{error}</p>}
    {isLoading && <div className="spinner-border"></div>}
    <button className="btn btn-primary mb-3" onClick={addUser}>
      Add User
    </button>
    <ul className="list-group">
      {users.map((user) => (
        <li
```

```
            key={user.id}
            className="list-group-item d-flex justify-content-between"
          >
            {user.name}
            <div>
              <button
                className="btn btn-outline-secondary mx-3"
                onClick={() => updateUser(user)}
              >
                Update
              </button>
              <button
                className="btn btn-outline-danger"
                onClick={() => deleteUser(user)}
              >
                Delete
              </button>
            </div>
          </li>
        ))}
      </ul>
    </>
  );
}

export default CH_FetchingDataServices;
```

api-client.ts

```ts
import axios from 'axios';
import { CanceledError } from "axios";

// headers is where things like api-key would go
export default axios.create({
   baseURL: "https://jsonplaceholder.typicode.com",
   })

export { CanceledError };
```

This code exports a default instance of the Axios library with a specified baseURL of "https://jsonplaceholder.typicode.com". Axios is a JavaScript library that allows us to make HTTP requests to web servers.

The axios.create() method returns an instance of the Axios library with a specified configuration. In this case, the configuration object only contains the baseURL property, which is the root URL for all HTTP requests made with this instance.

Additionally, the code exports the CanceledError class from Axios. This class is used when a request is canceled by an AbortController instance, and it can be used to handle this specific type of error.

Overall, this code exports a pre-configured Axios instance that can be used to make HTTP requests to the specified baseURL.

user-service.ts

```
import create from "./http-service";

export interface User {
 id: number;
 name: string;
}

export default create("/users");
```

http-service.ts

```
import apiClient from "./api-client";

interface Entity {
id: number;
}

class HTTPService {
   endpoint: string;
   constructor(endpoint: string) {
       this.endpoint = endpoint;
}

   // T = generic type parameter, replace later with <User> or <Post>, etc
   getAll<T>() {
       const controller = new AbortController();
```

This is a TypeScript code file that exports a factory function named "create" that returns an instance of the HTTPService class.

- The first line imports an "apiClient" module from a file named "api-client". We can assume that this module exports functions to handle HTTP requests.

- The next line defines an interface "Entity" that has a single property "id" of type number. This interface is used as a constraint on the "update" method of the "HTTPService" class.

- The "HTTPService" class is defined with a constructor that takes an "endpoint" string

```
        const request = apiClient.get<T[]>(this.endpoint, {
        signal: controller.signal,
      });
      return { request, cancel: () => controller.abort() }
  }

    delete(id: number) {
        return apiClient.delete(this.endpoint + "/" + id)
    };

    add<T>(entity: T) {
        return apiClient
        .post(this.endpoint, entity)
  };

    update<T extends Entity>(entity: T) {
    return apiClient.patch(this.endpoint + "/" + entity.id, entity)
  };
  }

const create = (endpoint: string) => new HTTPService(endpoint);

export default create;
```

parameter. The constructor sets the class property "endpoint" to the value of the "endpoint" parameter.
- The "getAll" method of the "HTTPService" class is defined with a generic type parameter "T". This method sends an HTTP GET request to the "endpoint" using the "apiClient" module's "get" function. The method returns an object with two properties - "request" and "cancel". The "request" property contains the actual request object that was sent, while the "cancel" property is a function that can be used to cancel the request using an AbortController.
- The "delete" method of the "HTTPService" class takes an "id" parameter of type number. This method sends an HTTP DELETE request to the "endpoint" with the specified ID using the "apiClient" module's "delete" function.
- The "add" method of the "HTTPService" class takes a generic type parameter "T" and an "entity" parameter of type "T". This method sends an HTTP POST request to the "endpoint" with the specified "entity" using the "apiClient" module's "post" function.
- The "update" method of the "HTTPService" class takes a generic type parameter "T" that extends the "Entity" interface, and an

"entity" parameter of type "T". This method sends an HTTP PATCH request to the "endpoint" with the ID of the specified "entity" using the "apiClient" module's "patch" function.

- The "create" function is defined as a function that takes an "endpoint" parameter of type string and returns a new instance of the "HTTPService" class with the specified "endpoint". This function is exported as the default export of the module.

To summarize, this code file exports a factory function that creates instances of an "HTTPService" class, which provides methods for making HTTP requests to a specified endpoint. The "getAll" method returns an object that can be used to cancel the request, while the "delete", "add", and "update" methods perform HTTP DELETE, POST, and PATCH requests respectively. The "update" method requires the "Entity" interface to be extended, which means that the "id" property must be present in any object passed to this method.

useUsers.ts

```
import apiClient from "./api-client";
```

This is a TypeScript code file that exports a

```
interface Entity {
id: number;
}

class HTTPService {
   endpoint: string;
   constructor(endpoint: string) {
       this.endpoint = endpoint;
}

   // T = generic type parameter, replace later with <User> or <Post>, etc
   getAll<T>() {
       const controller = new AbortController();
       const request = apiClient.get<T[]>(this.endpoint, {
       signal: controller.signal,
     });
     return { request, cancel: () => controller.abort() }
}

   delete(id: number) {
       return apiClient.delete(this.endpoint + "/" + id)
   };

   add<T>(entity: T) {
       return apiClient
       .post(this.endpoint, entity)
};
```

factory function named "create" that returns an instance of the HTTPService class.

- The first line imports an "apiClient" module from a file named "api-client". We can assume that this module exports functions to handle HTTP requests.

- The next line defines an interface "Entity" that has a single property "id" of type number. This interface is used as a constraint on the "update" method of the "HTTPService" class.

- The "HTTPService" class is defined with a constructor that takes an "endpoint" string parameter. The constructor sets the class property "endpoint" to the value of the "endpoint" parameter.

- The "getAll" method of the "HTTPService" class is defined with a generic type parameter "T". This method sends an HTTP GET request to the "endpoint" using the "apiClient" module's "get" function. The method returns an object with two properties - "request" and "cancel". The "request" property contains the actual request object that was sent, while the "cancel" property is a function that can be used to cancel the request using an AbortController.

```
   update<T extends Entity>(entity: T) {
   return apiClient.patch(this.endpoint + "/" + entity.id, entity)
};
}


const create = (endpoint: string) => new HTTPService(endpoint);

export default create;
```

- The "delete" method of the "HTTPService" class takes an "id" parameter of type number. This method sends an HTTP DELETE request to the "endpoint" with the specified ID using the "apiClient" module's "delete" function.

- The "add" method of the "HTTPService" class takes a generic type parameter "T" and an "entity" parameter of type "T". This method sends an HTTP POST request to the "endpoint" with the specified "entity" using the "apiClient" module's "post" function.

- The "update" method of the "HTTPService" class takes a generic type parameter "T" that extends the "Entity" interface, and an "entity" parameter of type "T". This method sends an HTTP PATCH request to the "endpoint" with the ID of the specified "entity" using the "apiClient" module's "patch" function.

- The "create" function is defined as a function that takes an "endpoint" parameter of type string and returns a new instance of the "HTTPService" class with the specified "endpoint". This function is exported as the default export of the module.

To summarize, this code file exports a factory function that creates instances of an "HTTPService" class, which provides methods for making HTTP requests to a specified endpoint. The "getAll" method returns an object that can be used to cancel the request, while the "delete", "add", and "update" methods perform HTTP DELETE, POST, and PATCH requests respectively. The "update" method requires the "Entity" interface to be extended, which means that the "id" property must be present in any object passed to this method.