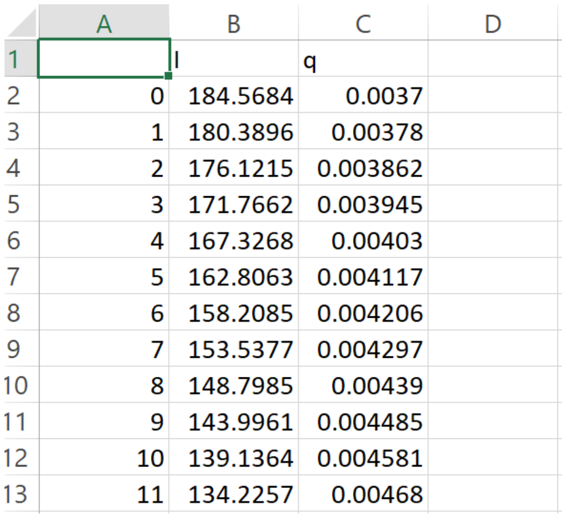
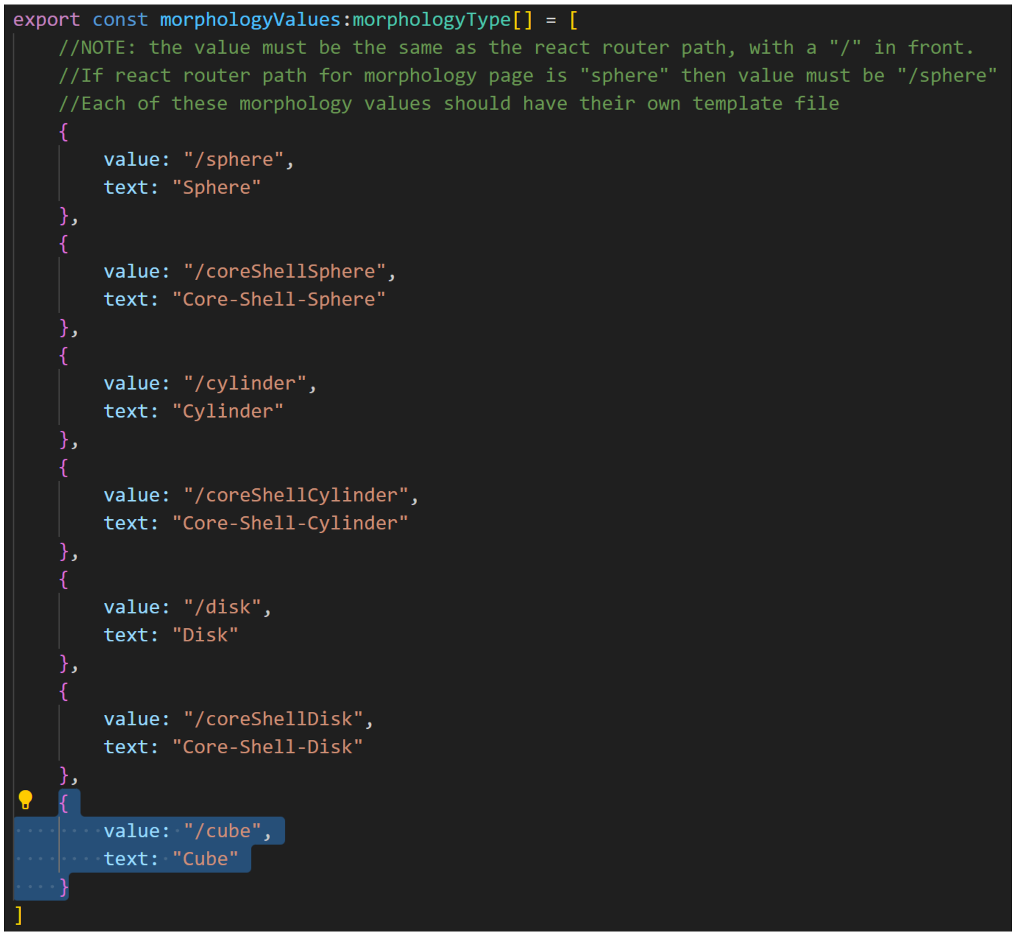
- Format for uploaded curve files is a CSV file with 3 columns in the below order and with the below labels on each column. The first column (no label) is numbered from 0 to 199. The second, labeled “I” is the intensity (Y-axis) points. The final column, labeled “q” is the q-value (X-axis) points, which are provided in q_values.txt in the “Extra Docs” folder. A template file with the middle column empty is provided in “Extra Docs,” empty_curve.csv. Examples are provided under “Extra Docs” on the github, with 2 examples (demo_curve.csv and example_curve.csv).



	A	B	C	D
1		I	q	
2	0	184.5684	0.0037	
3	1	180.3896	0.00378	
4	2	176.1215	0.003862	
5	3	171.7662	0.003945	
6	4	167.3268	0.00403	
7	5	162.8063	0.004117	
8	6	158.2085	0.004206	
9	7	153.5377	0.004297	
10	8	148.7985	0.00439	
11	9	143.9961	0.004485	
12	10	139.1364	0.004581	
13	11	134.2257	0.00468	

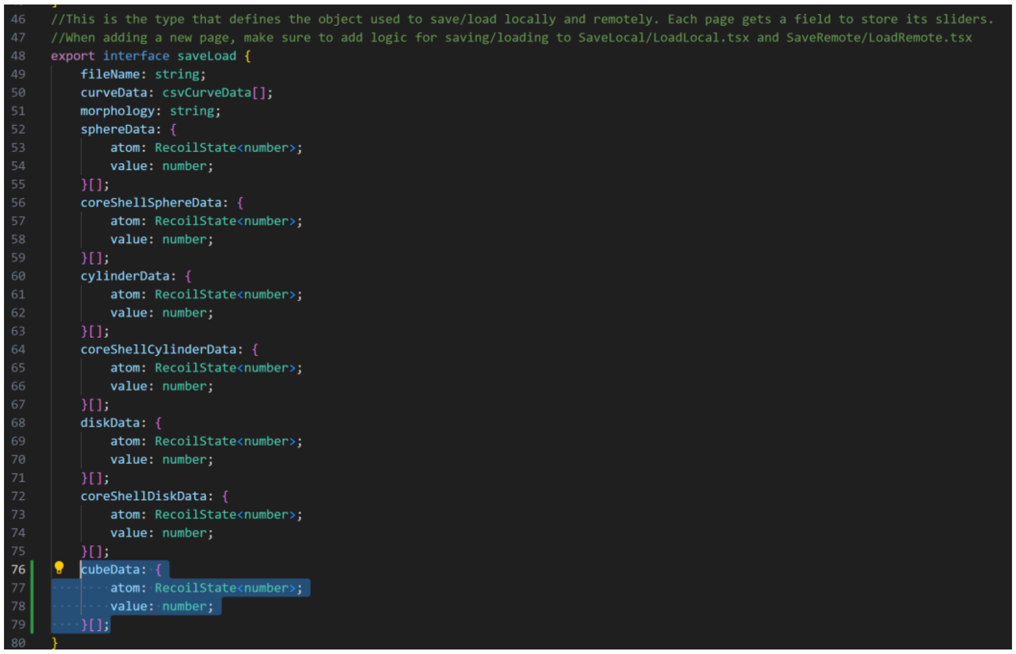
- Creating a new Morphology page (demonstrated by adding a “Cube” morphology):
  - Start in morphologyTemplate.tsx, in SASGUI-Frontend/src/atoms
    - Inside the morphologyValues array, add a new morphologyType object. The “value” will serve as both the path, and the string value to identify the morphology throughout the application. Because it serves as the path, the first character of the value string must be a forward slash (“/”). The “text” parameter is the name for the morphology that

will be visible to users.



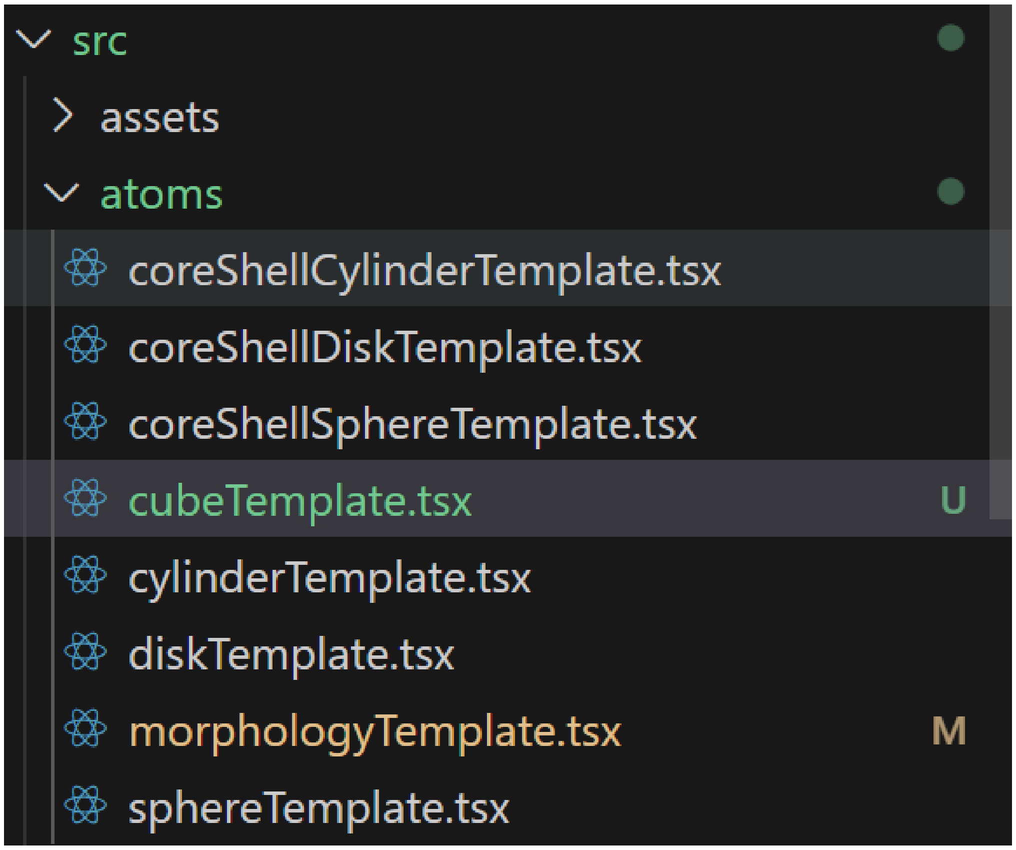
```
export const morphologyValues:morphologyType[] = [
    //NOTE: the value must be the same as the react router path, with a "/" in front.
    //If react router path for morphology page is "sphere" then value must be "/sphere"
    //Each of these morphology values should have their own template file
    {
        value: "/sphere",
        text: "Sphere"
    },
    {
        value: "/coreShellSphere",
        text: "Core-Shell-Sphere"
    },
    {
        value: "/cylinder",
        text: "Cylinder"
    },
    {
        value: "/coreShellCylinder",
        text: "Core-Shell-Cylinder"
    },
    {
        value: "/disk",
        text: "Disk"
    },
    {
        value: "/coreShellDisk",
        text: "Core-Shell-Disk"
    },
    {
        value: "/cube",
        text: "Cube"
    }
]
```

- Add a new object array for your morphology to the saveLoad interface. The format should be identical to the others present, just with a new name (NOTE: If you do not wish to add save/load functionality for the new morphology, do not do this step. If you do this step without implementing the save/load functionality it will break that functionality for all morphologies.)



```
//This is the type that defines the object used to save/load locally and remotely. Each page gets a field to store its sliders.
//When adding a new page, make sure to add logic for saving/loading to SaveLocal/LoadLocal.tsx and SaveRemote/LoadRemote.tsx
export interface saveLoad {
    fileName: string;
    curveData: csvCurveData[];
    morphology: string;
    sphereData: {
        atom: RecoilState<number>;
        value: number;
    }[];
    coreShellSphereData: {
        atom: RecoilState<number>;
        value: number;
    }[];
    cylinderData: {
        atom: RecoilState<number>;
        value: number;
    }[];
    coreShellCylinderData: {
        atom: RecoilState<number>;
        value: number;
    }[];
    diskData: {
        atom: RecoilState<number>;
        value: number;
    }[];
    coreShellDiskData: {
        atom: RecoilState<number>;
        value: number;
    }[];
    cubeData: {
        atom: RecoilState<number>;
        value: number;
    }[];
}
```

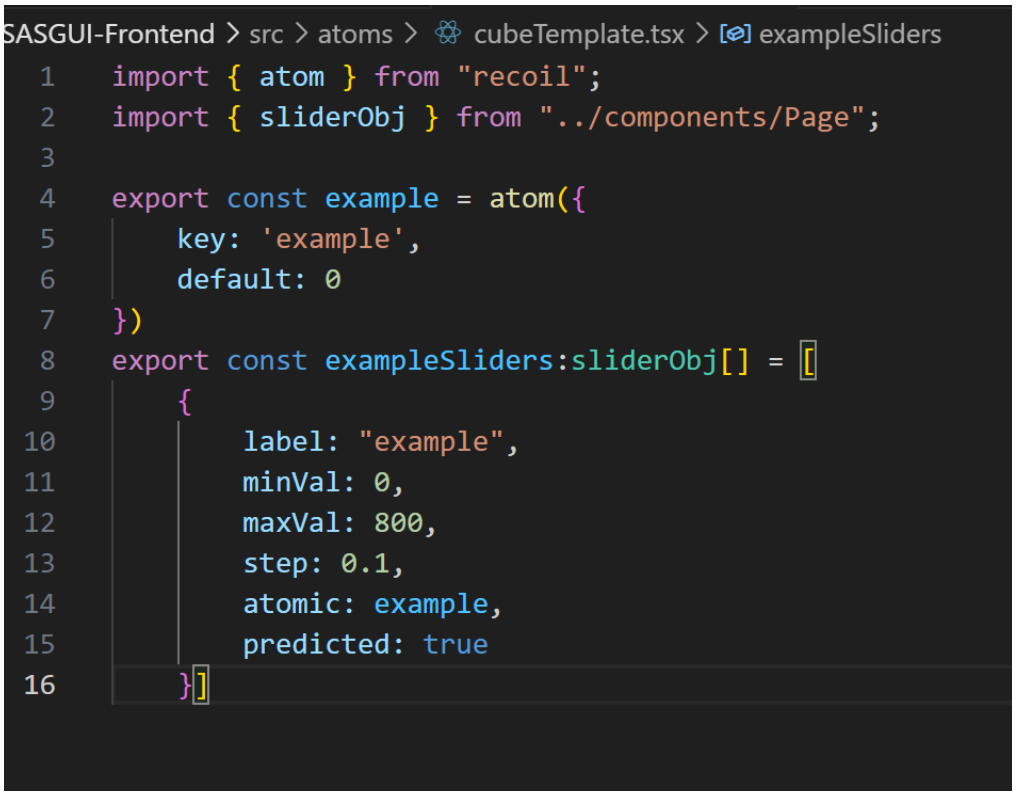
- Inside the atoms folder, create a new template file



```
src
  assets
  atoms
    coreShellCylinderTemplate.tsx
    coreShellDiskTemplate.tsx
    coreShellSphereTemplate.tsx
    cubeTemplate.tsx                U
    cylinderTemplate.tsx
    diskTemplate.tsx
    morphologyTemplate.tsx          M
    sphereTemplate.tsx
```

- This is the basic layout of a template file. Note that each slider follows the form of a sliderObj template, which is an object with a label, a minimum slider value, a maximum slider value, a “step” (the increment by which the slider will change the value), an atomic, which is what stores the slider’s value for access throughout the application, and a “predicted” boolean, which indicates (via a bold text label) whether or not the slider’s value is predicted by the

Machine Learning model.

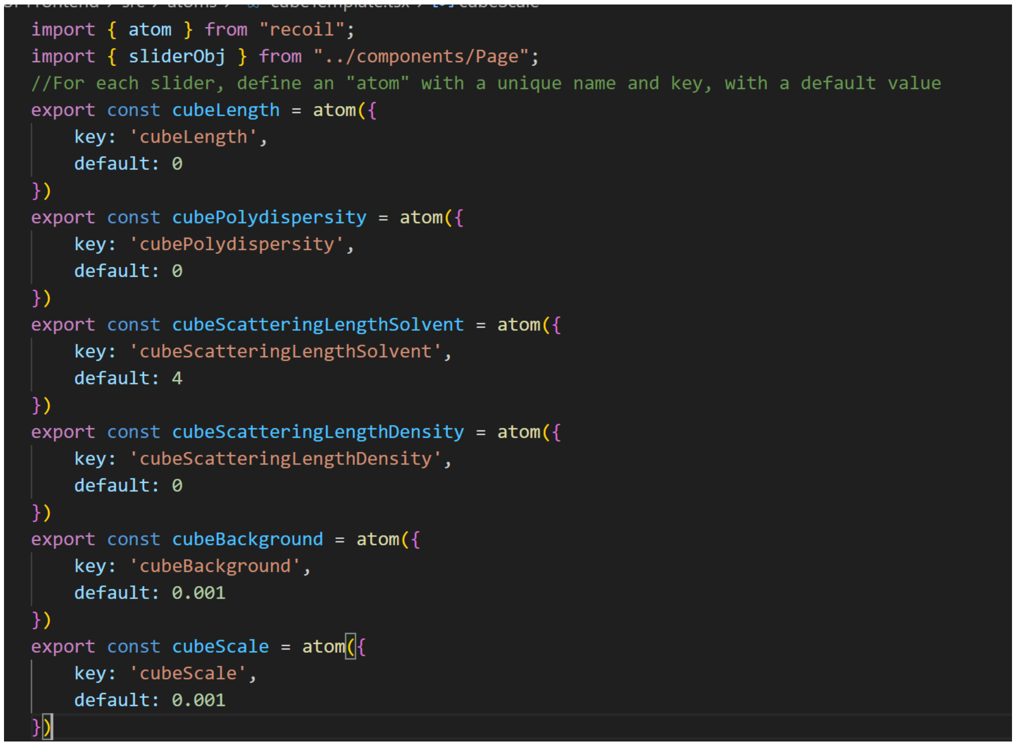


SASGUI-Frontend > src > atoms > cubeTemplate.tsx > exampleSliders

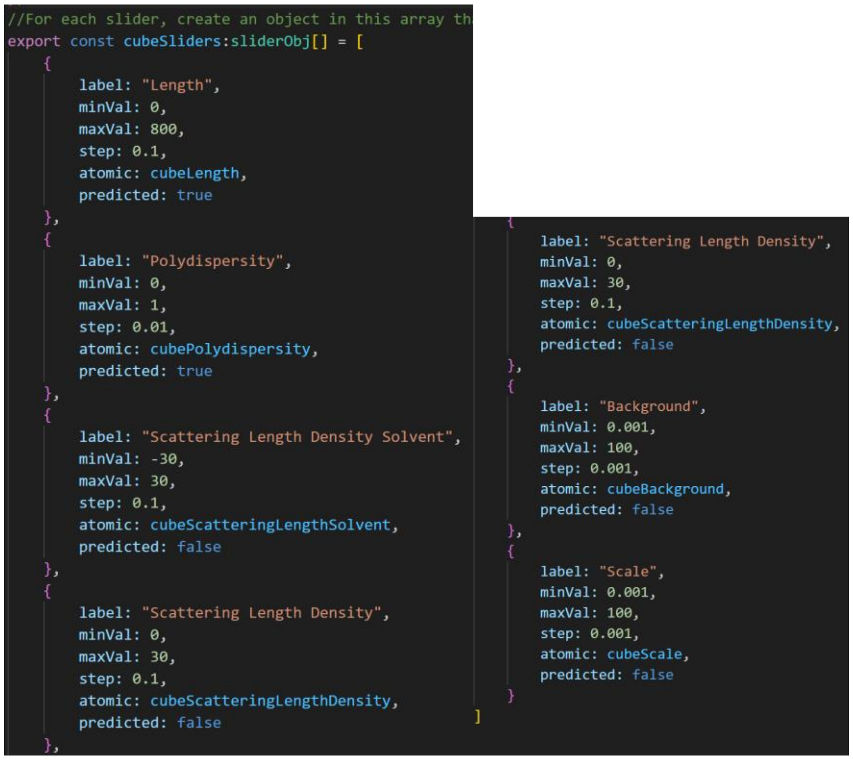
```tsx
import { atom } from "recoil";
import { sliderObj } from "../components/Page";

export const example = atom({
    key: 'example',
    default: 0
})
export const exampleSliders:sliderObj[] = [
    {
        label: "example",
        minVal: 0,
        maxVal: 800,
        step: 0.1,
        atomic: example,
        predicted: true
    }]
```

- Here is the example “cube” file:



```
import { atom } from "recoil";
import { sliderObj } from "../components/Page";
//For each slider, define an "atom" with a unique name and key, with a default value
export const cubeLength = atom({
    key: 'cubeLength',
    default: 0
})
export const cubePolydispersity = atom({
    key: 'cubePolydispersity',
    default: 0
})
export const cubeScatteringLengthSolvent = atom({
    key: 'cubeScatteringLengthSolvent',
    default: 4
})
export const cubeScatteringLengthDensity = atom({
    key: 'cubeScatteringLengthDensity',
    default: 0
})
export const cubeBackground = atom({
    key: 'cubeBackground',
    default: 0.001
})
export const cubeScale = atom({
    key: 'cubeScale',
    default: 0.001
})
```

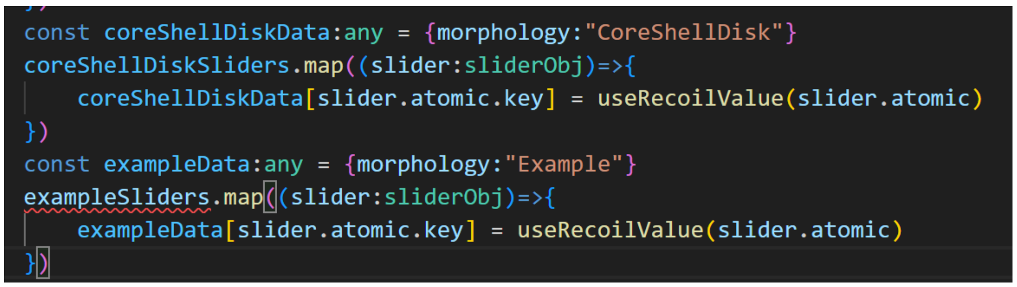


```
//For each slider, create an object in this array th
export const cubeSliders:sliderObj[] = [
    {
        label: "Length",
        minVal: 0,
        maxVal: 800,
        step: 0.1,
        atomic: cubeLength,
        predicted: true
    },
    {
        label: "Polydispersity",
        minVal: 0,
        maxVal: 1,
        step: 0.01,
        atomic: cubePolydispersity,
        predicted: true
    },
    {
        label: "Scattering Length Density Solvent",
        minVal: -30,
        maxVal: 30,
        step: 0.1,
        atomic: cubeScatteringLengthSolvent,
        predicted: false
    },
    {
        label: "Scattering Length Density",
        minVal: 0,
        maxVal: 30,
        step: 0.1,
        atomic: cubeScatteringLengthDensity,
        predicted: false
    },
```

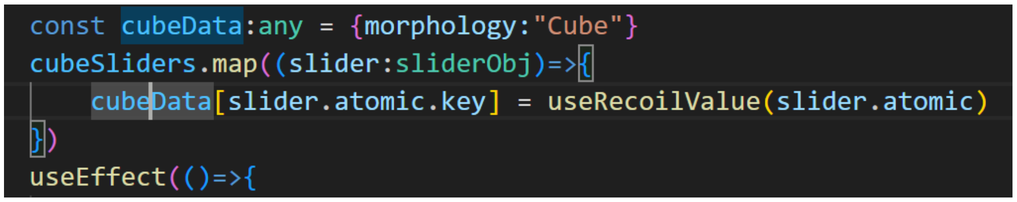
```
    {
        label: "Scattering Length Density",
        minVal: 0,
        maxVal: 30,
        step: 0.1,
        atomic: cubeScatteringLengthDensity,
        predicted: false
    },
    {
        label: "Background",
        minVal: 0.001,
        maxVal: 100,
        step: 0.001,
        atomic: cubeBackground,
        predicted: false
    },
    {
        label: "Scale",
        minVal: 0.001,
        maxVal: 100,
        step: 0.001,
        atomic: cubeScale,
        predicted: false
    }
]
```

- Next, move to wrapper.tsx in the SASGUI-Frontend/src/components folder
    - Near the top of the Wrapper function is a set of objects and mapping functions. These objects that are created by the mapping functions collect the data from the sliders when they are updated, and then

trigger a call to the backend to generate a graph. Below is a generic example, and a less generic one for the cube page we are creating:

- 

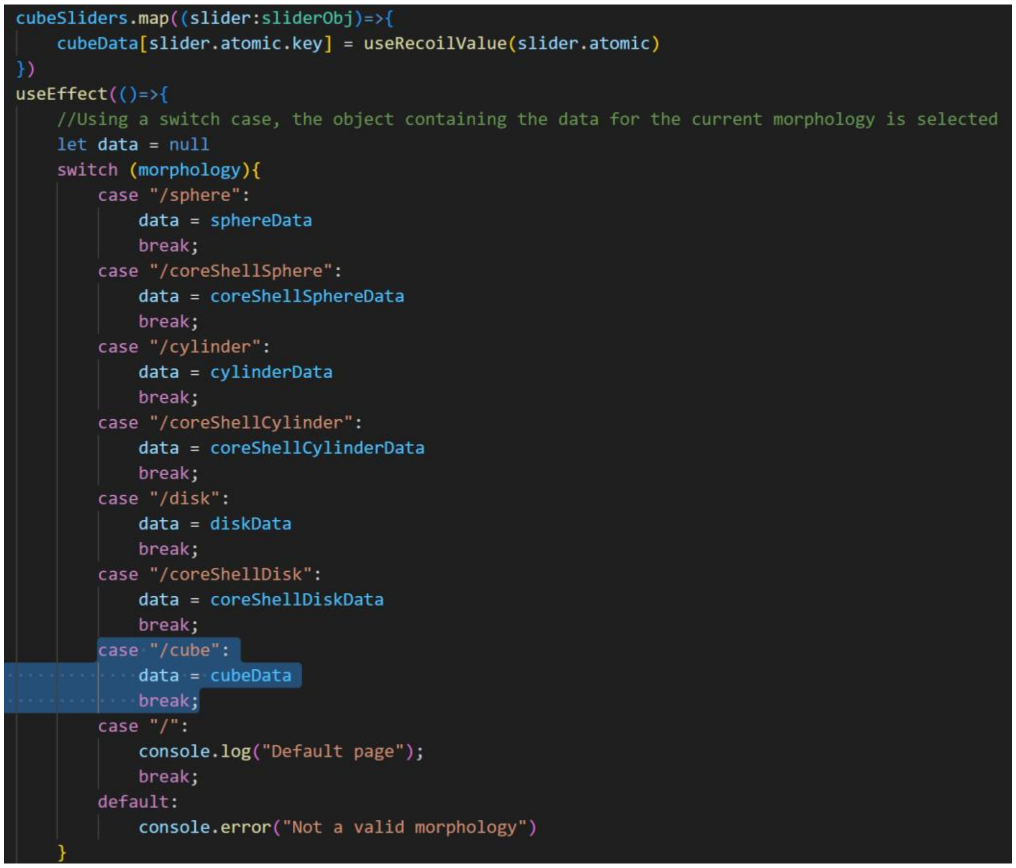
```
const coreShellDiskData:any = {morphology:"CoreShellDisk"}
coreShellDiskSliders.map((slider:sliderObj)=>{
    coreShellDiskData[slider.atomic.key] = useRecoilValue(slider.atomic)
})
const exampleData:any = {morphology:"Example"}
exampleSliders.map((slider:sliderObj)=>{
    exampleData[slider.atomic.key] = useRecoilValue(slider.atomic)
})
```



```
const cubeData:any = {morphology:"Cube"}
cubeSliders.map((slider:sliderObj)=>{
    cubeData[slider.atomic.key] = useRecoilValue(slider.atomic)
})
useEffect(()=>{
```

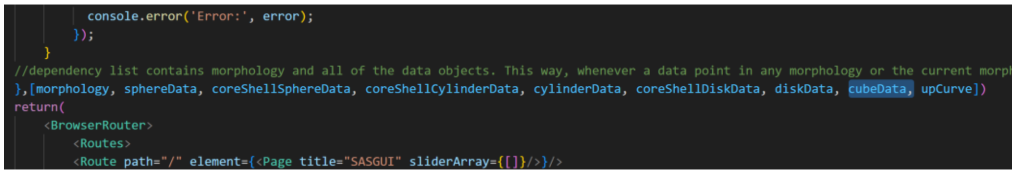
NOTE: the ”morphology” field in the cubeData object is the morphology identification string that gets sent to the backend. When expanding the backend to include the logic for this new page later, you must remember to use this string to identify the morphology for the incoming graphing request.

- In the switch case in the useEffect, add a new case for your new morphology. NOTE: make sure the value of the case is the same as the value you used in morphologyValues in morphologyTemplate.tsx



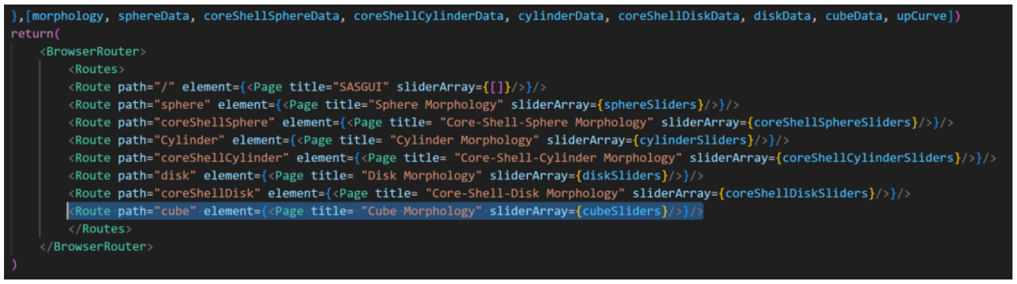
```
cubeSliders.map((slider:sliderObj)=>{
    cubeData[slider.atomic.key] = useRecoilValue(slider.atomic)
})
useEffect(()=>{
    //Using a switch case, the object containing the data for the current morphology is selected
    let data = null
    switch (morphology){
        case "/sphere":
            data = sphereData
            break;
        case "/coreShellSphere":
            data = coreShellSphereData
            break;
        case "/cylinder":
            data = cylinderData
            break;
        case "/coreShellCylinder":
            data = coreShellCylinderData
            break;
        case "/disk":
            data = diskData
            break;
        case "/coreShellDisk":
            data = coreShellDiskData
            break;
        case "/cube":
            data = cubeData
            break;
        case "/":
            console.log("Default page");
            break;
        default:
            console.error("Not a valid morphology")
    }
```

- Add the cubeData object to the dependency list for the useEffect, this will ensure the API for generating the graph gets called whenever the cube page’s data updates



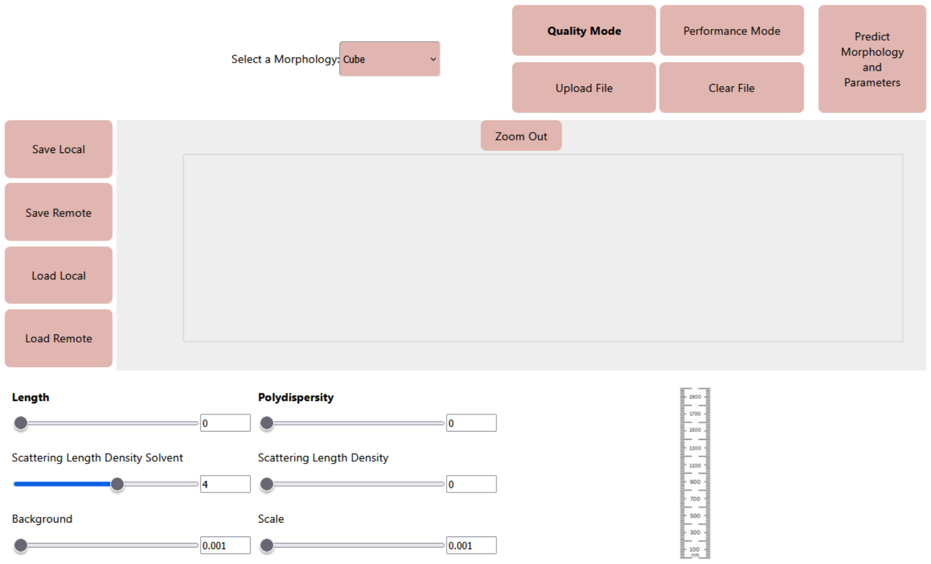
```
            console.error('Error:', error);
        });
    }
//dependency list contains morphology and all of the data objects. This way, whenever a data point in any morphology or the current morph
},[morphology, sphereData, coreShellSphereData, coreShellCylinderData, cylinderData, coreShellDiskData, diskData, cubeData, upCurve])
return(
    <BrowserRouter>
        <Routes>
        <Route path="/" element={<Page title="SASGUI" sliderArray={[]}/>}/>
```

        - Finally, add a new <Route> for your new page. The path is the value you set in morphologyValues in morphologyTemplate.tsx, but without that forward slash first character.



```
},[morphology, sphereData, coreShellSphereData, coreShellCylinderData, cylinderData, coreShellDiskData, diskData, cubeData, upCurve])
return(
    <BrowserRouter>
        <Routes>
        <Route path="/" element={<Page title="SASGUI" sliderArray={[]}/>}/>
        <Route path="sphere" element={<Page title="Sphere Morphology" sliderArray={sphereSliders}/>}/>
        <Route path="coreShellSphere" element={<Page title= "Core-Shell-Sphere Morphology" sliderArray={coreShellSphereSliders}/>}/>
        <Route path="Cylinder" element={<Page title= "Cylinder Morphology" sliderArray={cylinderSliders}/>}/>
        <Route path="coreShellCylinder" element={<Page title= "Core-Shell-Cylinder Morphology" sliderArray={coreShellCylinderSliders}/>}/>
        <Route path="disk" element={<Page title= "Disk Morphology" sliderArray={diskSliders}/>}/>
        <Route path="coreShellDisk" element={<Page title= "Core-Shell-Disk Morphology" sliderArray={coreShellDiskSliders}/>}/>
        <Route path="cube" element={<Page title= "Cube Morphology" sliderArray={cubeSliders}/>}/>
        </Routes>
    </BrowserRouter>
)
```

    - At this stage, the cube page is visible, but it lacks the ability to save/load state and neither the graph, nor the machine learning model function. To add that functionality, please see the section on expanding the backend functionality. For the saving and loading, continue below.

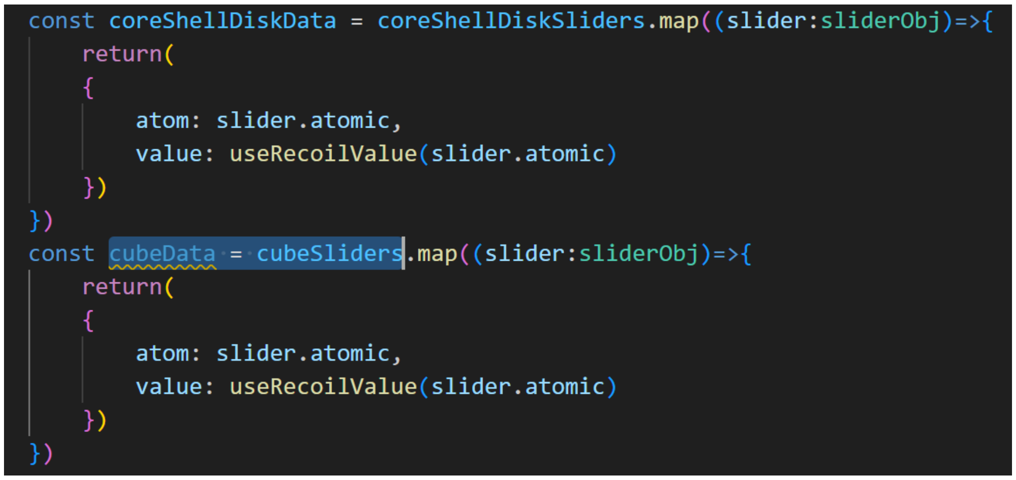


Select a Morphology: Cube

Length 0
Polydispersity 0
Scattering Length Density Solvent 4
Scattering Length Density 0
Background 0.001
Scale 0.001

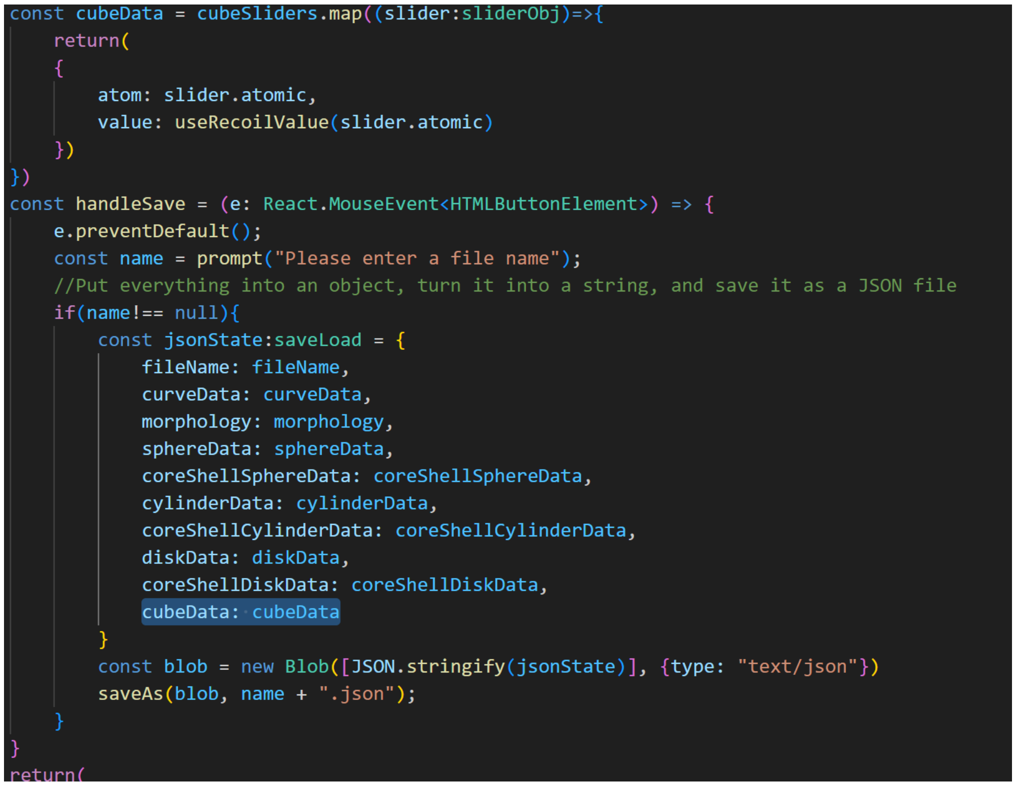
- Saving and loading for a new morphology page:
    - Start in SaveLocal.tsx in SASGUI-Frontend/src/components

- Near the top of the SaveLocal function, you will see a number of Data arrays, each created by performing a mapping function on the sliderObj object from the template file. You will need to make one of these for your new morphology. To make one, you just need to name the new data array, and to initialize it, run the mapping function on the sliderObj array from your template file, like below:



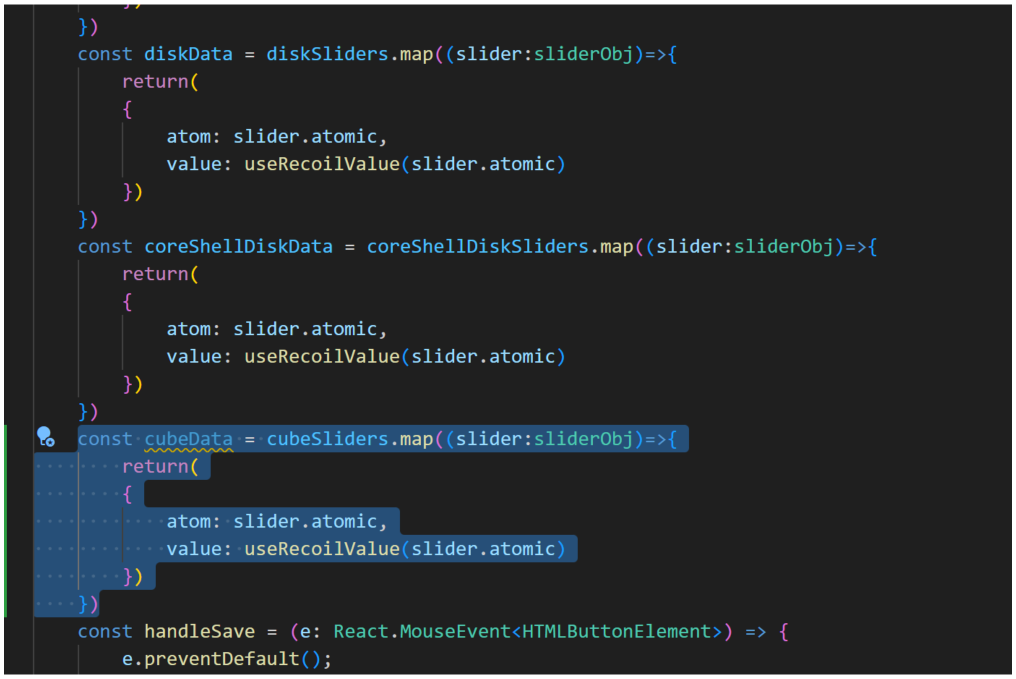
```
const coreShellDiskData = coreShellDiskSliders.map((slider:sliderObj)=>{
    return(
    {
        atom: slider.atomic,
        value: useRecoilValue(slider.atomic)
    })
})
const cubeData = cubeSliders.map((slider:sliderObj)=>{
    return(
    {
        atom: slider.atomic,
        value: useRecoilValue(slider.atomic)
    })
})
```

- Once that step is done, add your new data array to the jsonState object in the handleSave function, as below:

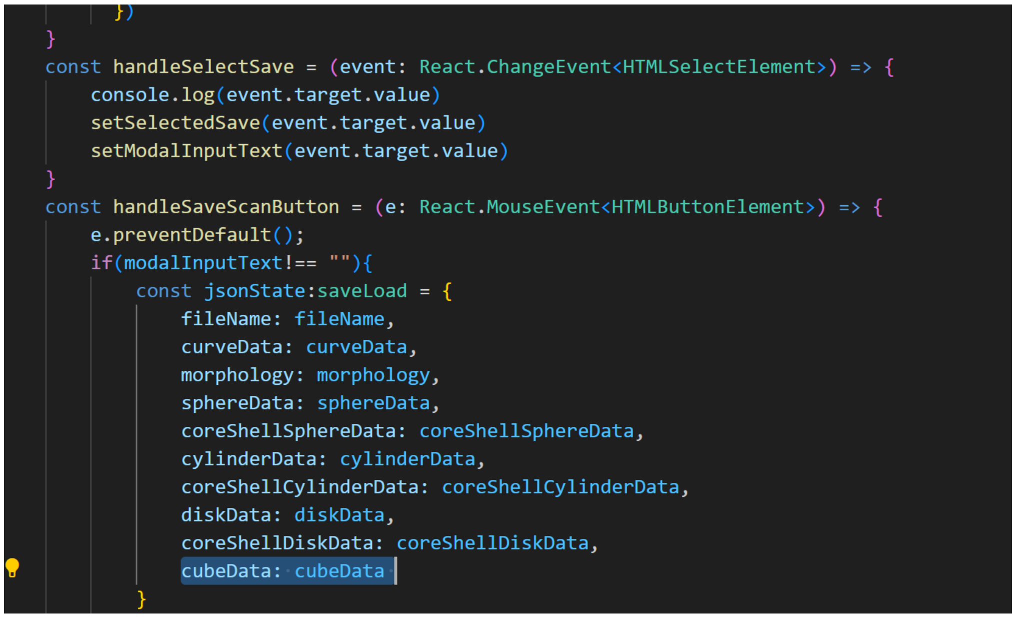


```
const cubeData = cubeSliders.map((slider:sliderObj)=>{
    return(
    {
        atom: slider.atomic,
        value: useRecoilValue(slider.atomic)
    })
})
const handleSave = (e: React.MouseEvent<HTMLButtonElement>) => {
    e.preventDefault();
    const name = prompt("Please enter a file name");
    //Put everything into an object, turn it into a string, and save it as a JSON file
    if(name!== null){
        const jsonState:saveLoad = {
            fileName: fileName,
            curveData: curveData,
            morphology: morphology,
            sphereData: sphereData,
            coreShellSphereData: coreShellSphereData,
            cylinderData: cylinderData,
            coreShellCylinderData: coreShellCylinderData,
            diskData: diskData,
            coreShellDiskData: coreShellDiskData,
            cubeData: cubeData
        }
        const blob = new Blob([JSON.stringify(jsonState)], {type: "text/json"})
        saveAs(blob, name + ".json");
    }
}
return(
```

- Local saving is now ready. For remote saving, the process is almost identical, but in SaveRemote.tsx.

- 

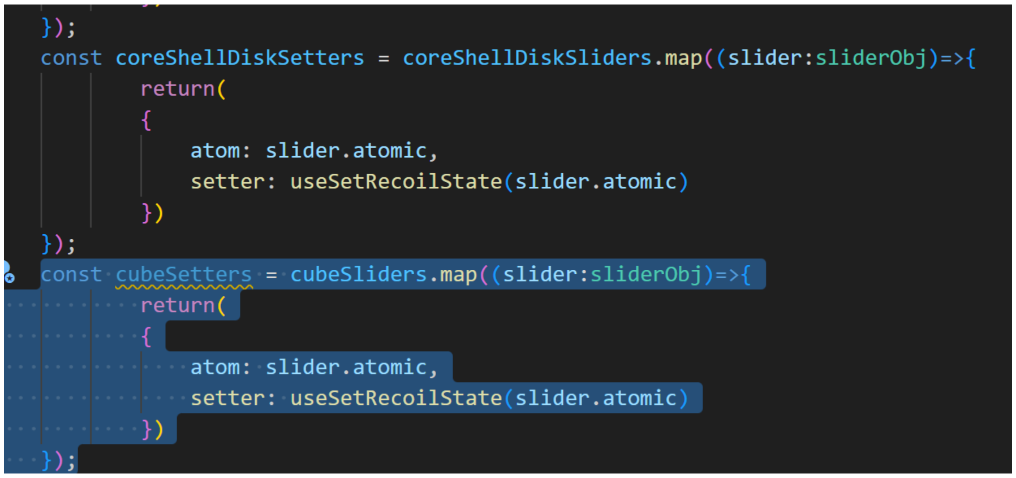
```
    })
    const diskData = diskSliders.map((slider:sliderObj)=>{
        return(
        {
            atom: slider.atomic,
            value: useRecoilValue(slider.atomic)
        })
    })
    const coreShellDiskData = coreShellDiskSliders.map((slider:sliderObj)=>{
        return(
        {
            atom: slider.atomic,
            value: useRecoilValue(slider.atomic)
        })
    })
    const cubeData = cubeSliders.map((slider:sliderObj)=>{
        return(
        {
            atom: slider.atomic,
            value: useRecoilValue(slider.atomic)
        })
    })
    const handleSave = (e: React.MouseEvent<HTMLButtonElement>) => {
        e.preventDefault();
```



```
        })
    }
    const handleSelectSave = (event: React.ChangeEvent<HTMLSelectElement>) => {
        console.log(event.target.value)
        setSelectedSave(event.target.value)
        setModalInputText(event.target.value)
    }
    const handleSaveScanButton = (e: React.MouseEvent<HTMLButtonElement>) => {
        e.preventDefault();
        if(modalInputText!== ""){
            const jsonState:saveLoad = {
                fileName: fileName,
                curveData: curveData,
                morphology: morphology,
                sphereData: sphereData,
                coreShellSphereData: coreShellSphereData,
                cylinderData: cylinderData,
                coreShellCylinderData: coreShellCylinderData,
                diskData: diskData,
                coreShellDiskData: coreShellDiskData,
                cubeData: cubeData
            }
```

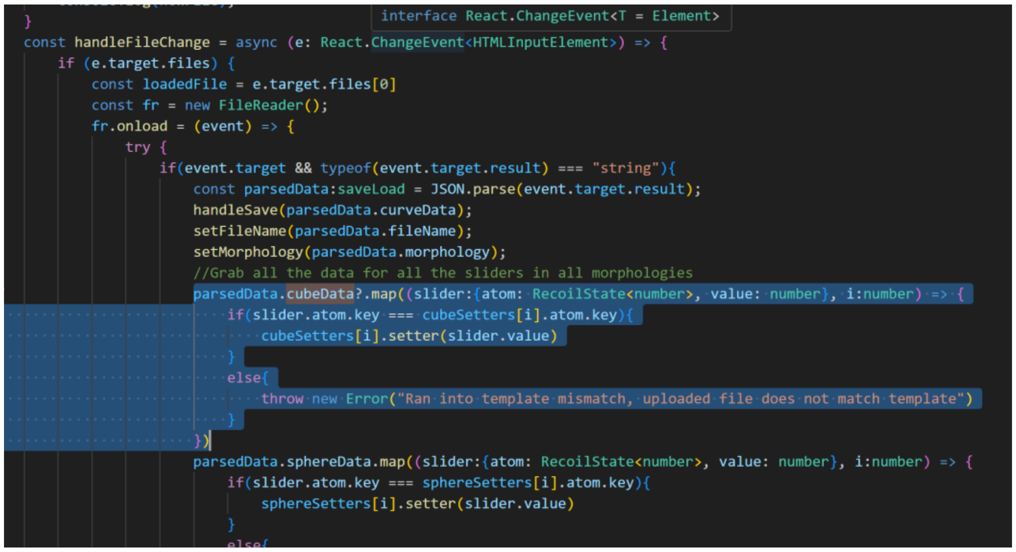
- Moving on to loading, we move to the LoadLocal.tsx file.

- You will need to create a setter object, which will contain the "atoms" of state that correspond to the data fields. By accessing these atoms, the values they hold can be changed to the values stored in a save. These setters are located near the top of the LoadLocal function.



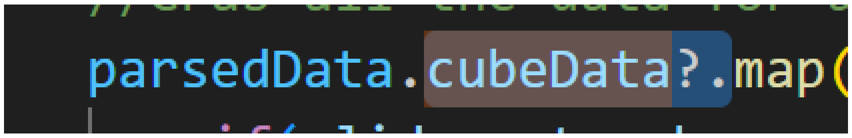
```
});
const coreShellDiskSetters = coreShellDiskSliders.map((slider:sliderObj)=>{
        return(
        {
            atom: slider.atomic,
            setter: useSetRecoilState(slider.atomic)
        })
});
const cubeSetters = cubeSliders.map((slider:sliderObj)=>{
        return(
        {
            atom: slider.atomic,
            setter: useSetRecoilState(slider.atomic)
        })
});
```

- Inside the handleFileChange function you will need to add logic to process the save data, like the one below:



```
}                                      interface React.ChangeEvent<T = Element>
const handleFileChange = async (e: React.ChangeEvent<HTMLInputElement>) => {
    if (e.target.files) {
        const loadedFile = e.target.files[0]
        const fr = new FileReader();
        fr.onload = (event) => {
            try {
                if(event.target && typeof(event.target.result) === "string"){
                    const parsedData:saveLoad = JSON.parse(event.target.result);
                    handleSave(parsedData.curveData);
                    setFileName(parsedData.fileName);
                    setMorphology(parsedData.morphology);
                    //Grab all the data for all the sliders in all morphologies
                    parsedData.cubeData?.map((slider:{atom: RecoilState<number>, value: number}, i:number) => {
                        if(slider.atom.key === cubeSetters[i].atom.key){
                            cubeSetters[i].setter(slider.value)
                        }
                        else{
                            throw new Error("Ran into template mismatch, uploaded file does not match template")
                        }
                    })
                    parsedData.sphereData.map((slider:{atom: RecoilState<number>, value: number}, i:number) => {
                        if(slider.atom.key === sphereSetters[i].atom.key){
                            sphereSetters[i].setter(slider.value)
                        }
```

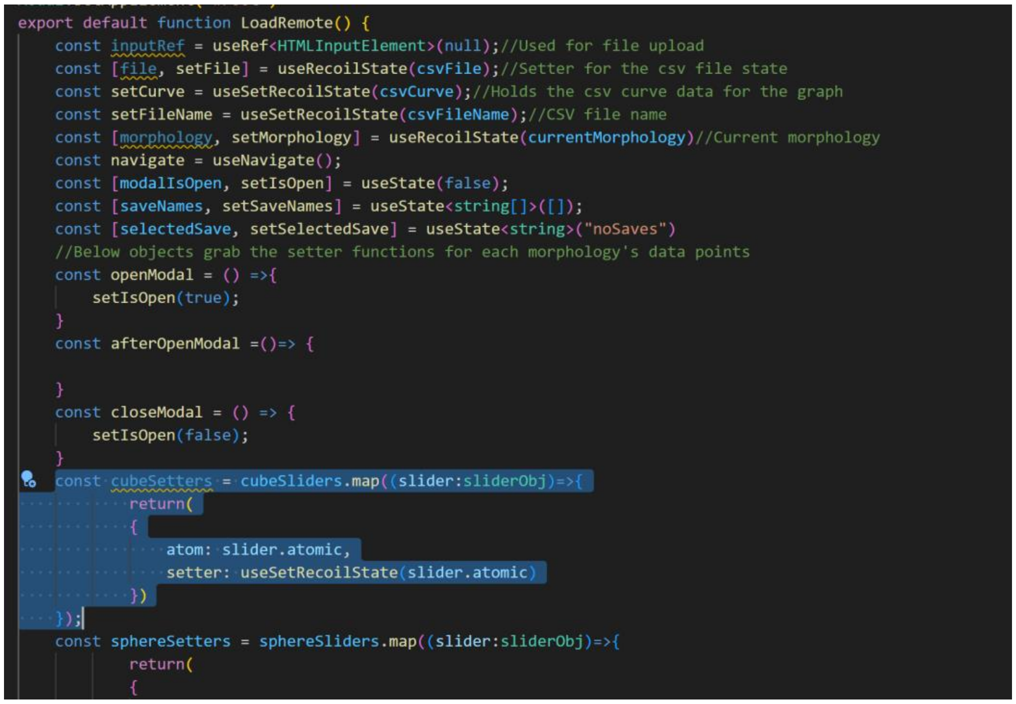
NOTE: for all added morphology, make sure to have a ?. operator between the data array name and "map", this ensures that old saves will remain compatible by only performing the logic if the new morphology's data is present!



```
parsedData.cubeData?.map(
```

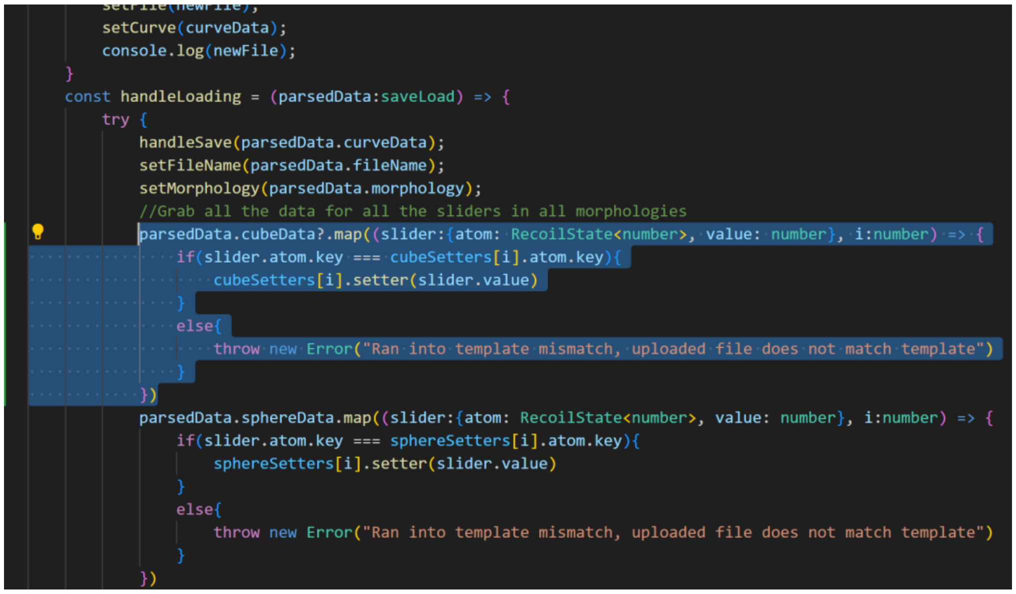
- The format of the processing logic is the same for all morphologies, with only the referenced objects changing.

- Make the same changes as LoadLocal in LoadRemote, adding the setter array and the processing logic. Note that the format is the same as in LoadLocal



```
export default function LoadRemote() {
    const inputRef = useRef<HTMLInputElement>(null);//Used for file upload
    const [file, setFile] = useRecoilState(csvFile);//Setter for the csv file state
    const setCurve = useSetRecoilState(csvCurve);//Holds the csv curve data for the graph
    const setFileName = useSetRecoilState(csvFileName);//CSV file name
    const [morphology, setMorphology] = useRecoilState(currentMorphology)//Current morphology
    const navigate = useNavigate();
    const [modalIsOpen, setIsOpen] = useState(false);
    const [saveNames, setSaveNames] = useState<string[]>([]);
    const [selectedSave, setSelectedSave] = useState<string>("noSaves")
    //Below objects grab the setter functions for each morphology's data points
    const openModal = () =>{
        setIsOpen(true);
    }
    const afterOpenModal =()=> {

    }
    const closeModal = () => {
        setIsOpen(false);
    }
    const cubeSetters = cubeSliders.map((slider:sliderObj)=>{
            return(
            {
                atom: slider.atomic,
                setter: useSetRecoilState(slider.atomic)
            })
    });
    const sphereSetters = sphereSliders.map((slider:sliderObj)=>{
            return(
            {
```



```
        setFile(newFile);
        setCurve(curveData);
        console.log(newFile);
    }
    const handleLoading = (parsedData:saveLoad) => {
        try {
            handleSave(parsedData.curveData);
            setFileName(parsedData.fileName);
            setMorphology(parsedData.morphology);
            //Grab all the data for all the sliders in all morphologies
            parsedData.cubeData?.map((slider:{atom: RecoilState<number>, value: number}, i:number) => {
                if(slider.atom.key === cubeSetters[i].atom.key){
                    cubeSetters[i].setter(slider.value)
                }
                else{
                    throw new Error("Ran into template mismatch, uploaded file does not match template")
                }
            })
            parsedData.sphereData.map((slider:{atom: RecoilState<number>, value: number}, i:number) => {
                if(slider.atom.key === sphereSetters[i].atom.key){
                    sphereSetters[i].setter(slider.value)
                }
                else{
                    throw new Error("Ran into template mismatch, uploaded file does not match template")
                }
            })
```

NOTE: Once again make sure to have a ?. between the data array and map

- Expanding the 3D model:

Database

Primary use: stores scan data and user credentials

Tools: SQLite

Tables:

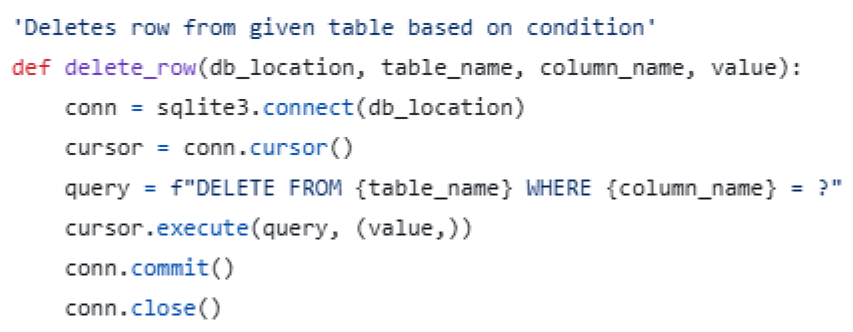
- Users
    - Stores information about each registered user

userID	INT	Primary Key, auto-incrementing
username	Text	Unique username, Not Null
password	Text	Password, Not Null
email	Text	email address
securityQuestion	Text	Security question for password recovery
securityAnswer	Text	Answer to the security question

- Scans
    - Stores uploaded scans with associated metadata

userId	INT	Used to reference users.userId
fileName	Text	Name of the scan file, Primary Key
fileDate	Text	JSON string with scan parameters and results

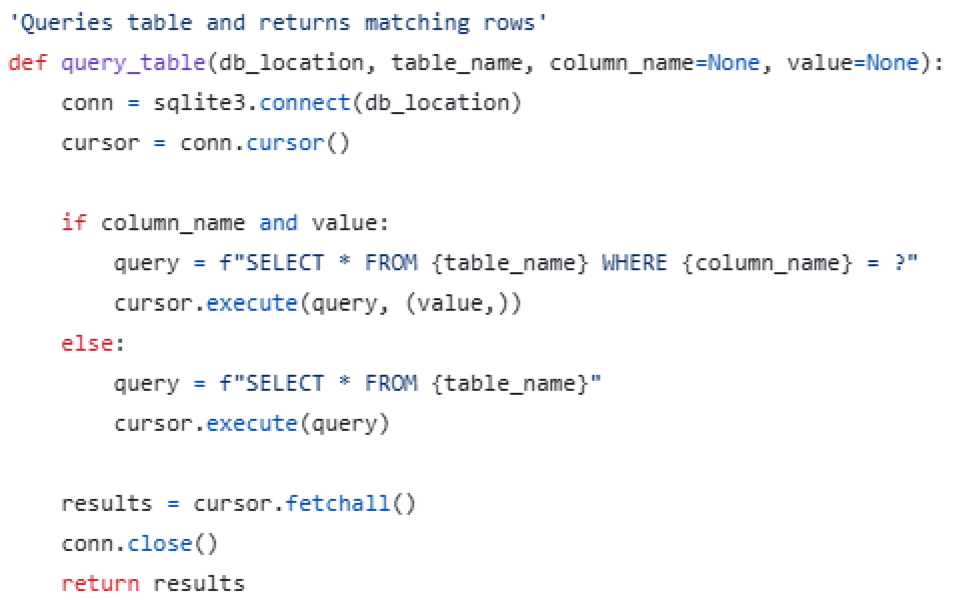
General functions:



```
'Deletes row from given table based on condition'
def delete_row(db_location, table_name, column_name, value):
    conn = sqlite3.connect(db_location)
    cursor = conn.cursor()
    query = f"DELETE FROM {table_name} WHERE {column_name} = ?"
    cursor.execute(query, (value,))
    conn.commit()
    conn.close()
```

Deletes a row based on a column-value condition.

- Constructs a DELETE FROM ... WHERE ... SQL query.
- Uses parameter substitution to avoid injection.
- Commits the deletion and closes the DB connection.



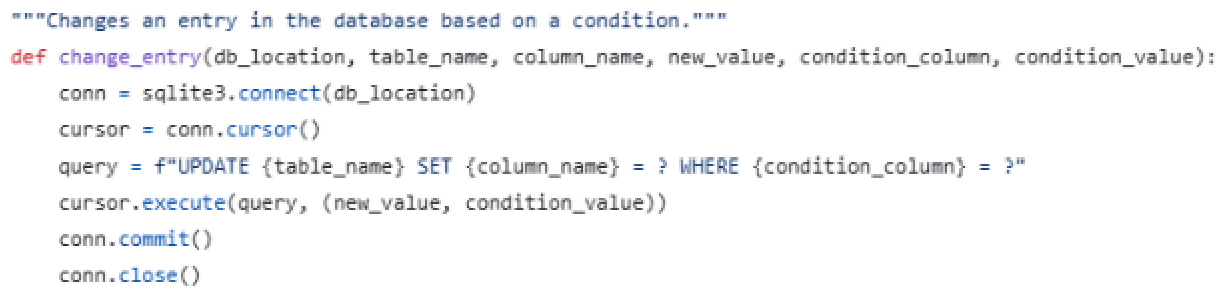
```python
'Queries table and returns matching rows'
def query_table(db_location, table_name, column_name=None, value=None):
    conn = sqlite3.connect(db_location)
    cursor = conn.cursor()

    if column_name and value:
        query = f"SELECT * FROM {table_name} WHERE {column_name} = ?"
        cursor.execute(query, (value,))
    else:
        query = f"SELECT * FROM {table_name}"
        cursor.execute(query)

    results = cursor.fetchall()
    conn.close()
    return results
```

Fetches records based on a condition or all rows if no condition is passed.

- If a column and value are provided, uses a WHERE clause.
- Otherwise, selects all rows.
- Returns a list of tuples (rows).

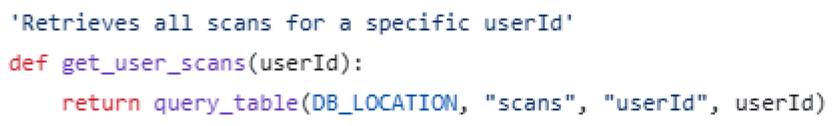


```python
"""Changes an entry in the database based on a condition."""
def change_entry(db_location, table_name, column_name, new_value, condition_column, condition_value):
    conn = sqlite3.connect(db_location)
    cursor = conn.cursor()
    query = f"UPDATE {table_name} SET {column_name} = ? WHERE {condition_column} = ?"
    cursor.execute(query, (new_value, condition_value))
    conn.commit()
    conn.close()
```

Updates a specific value in a given row.

- Constructs a SQL UPDATE ... SET ... WHERE ... query using parameters.
- Commits the change.

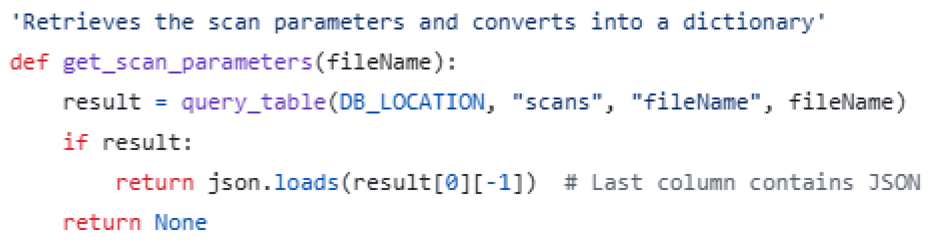
Scan specific functions:



```python
'Retrieves all scans for a specific userId'
def get_user_scans(userId):
    return query_table(DB_LOCATION, "scans", "userId", userId)
```

Returns all scans associated with a user ID.

- Calls query_table() on scans with a userId condition.



```python
'Retrieves the scan parameters and converts into a dictionary'
def get_scan_parameters(fileName):
    result = query_table(DB_LOCATION, "scans", "fileName", fileName)
    if result:
        return json.loads(result[0][-1])  # Last column contains JSON
    return None
```

Parses fileData (JSON) from a scan entry.

- Queries scans table by fileName.
- Parses the last column (fileData) from JSON to dict.

2) add_to_table(table_name, column_names, new_values, db_location = DB_LOCATION)
    a) Description: base function to add an entry to a given table.
    b) Parameters:
        i) table_name
            (1) String. Name of the table within the SQL database.
        ii) column_names
            (1) List of strings. List must contain all mandatory columns. Order must match of values within new_values
        iii) new_values
            (1) Tuple of values. Tuple must match order of strings in column names. Values must match data types in schema.
        iv) db_location
            (1) String, default is DB_LOCATION
    c) Return: dictionary of json format
        i) success
            (1) True or false depending on success of function to change dictionary
        ii) message
            (1) Custom message
        iii) error
            (1) Exception data

3) add_or_replace_to_table(table_name, column_names, new_values, db_location = DB_LOCATION)
    a) Description: A variation of add_to_table, if an entry with the same primary key exists, it will replace the old one.
    b) Parameters:
        i) table_name
            (1) String. Name of the table within the SQL database.
        ii) column_names
            (1) List of strings. List must contain all mandatory columns. Order must match of values within new_values
        iii) new_values
            (1) Tuple of values. Tuple must match order of strings in column names. Values must match data types in schema.

        iv) db_location
            (1) String, default is DB_LOCATION
    c) Return: dictionary of json format
        i) success
            (1) True or false depending on success of function to change dictionary
        ii) message
            (1) Custom message
        iii) error
            (1) Exception data

4) change_entry(table_name, column_name, new_value, condition_column, condition_value, db_location = DB_LOCATION)
    a) Description: Changes an entry in the database based on a condition.
    b) Parameters:
        i) table_name
            (1) String. Name of the table within the SQL database.
        ii) column_name
            (1) String. The column of the entry you want to change.
        iii) new_value
            (1) Must match data type of column.
        iv) condition_column
            (1) String. The column you are checking for the condition.
        v) condition_value
            (1) Must match data type of condition_column. Will target all entries that have a value that is equal to this condition_value within the condition_column.
        vi) db_location
            (1) String, default is DB_LOCATION
    c) Return:
        i) None

5) add_to_users(username, password, email, securityQuestion = "", securityAnswer = "")
    a) Description: Adds a new user to the database,which will automatically give them a userID.

b) Parameters:
   i) username
      (1) String.
   ii) password
      (1) String.
   iii) email
      (1) String.
   iv) securityQuestion
      (1) String, default is “”, an empty string.
   v) securityAnswer
      (1) String, default is “”, an empty string.
c) Return:
   i) The result dictionary from add_to_table

6) def add_to_scans(file_name, file_data, userId = 1)
   a) Description:
      i) Adds file data and some metadata to the scans table
   b) Parameters:
      i) file_name
         (1) String.
      ii) file_data
         (1) String. JSON string with scan parameters and datapoints
      iii) userID
         (1) Integer, default = 1. The userID of 1 can be seen as a global user, mainly for use in testing.
   c) Return: None

7) get_user_info(userId)
   a) Description:
      i) Retrieves all available data of a user given the user’s id.
   b) Parameters:
      i) userId
         (1) Integer.
   c) Return:
      i) {"userId": String, "username": String, "password": String, "email": String, "security_question": String, "security_answer": String}

8) get_id_by_username(username)
    a) Description:
        i) Will find any userID associated with a given username within the database
    b) Parameters:
        i) username
            (1) String.
    c) Return: {"userId": String}

9) get_id_by_email(email)
    a) Description:
        i) Will find any the userID associated with a given email within the database
    b) Parameters:
        i) email
            (1) String.
    c) Return: {"userId": String}

10) change_password_by_userId(userId, new_password)
    a) Description:
        i) Will change the password within the database of the user with the matching userID
    b) Parameters:
        i) UserId
            (1) Integer
    c) Return:
        i) None