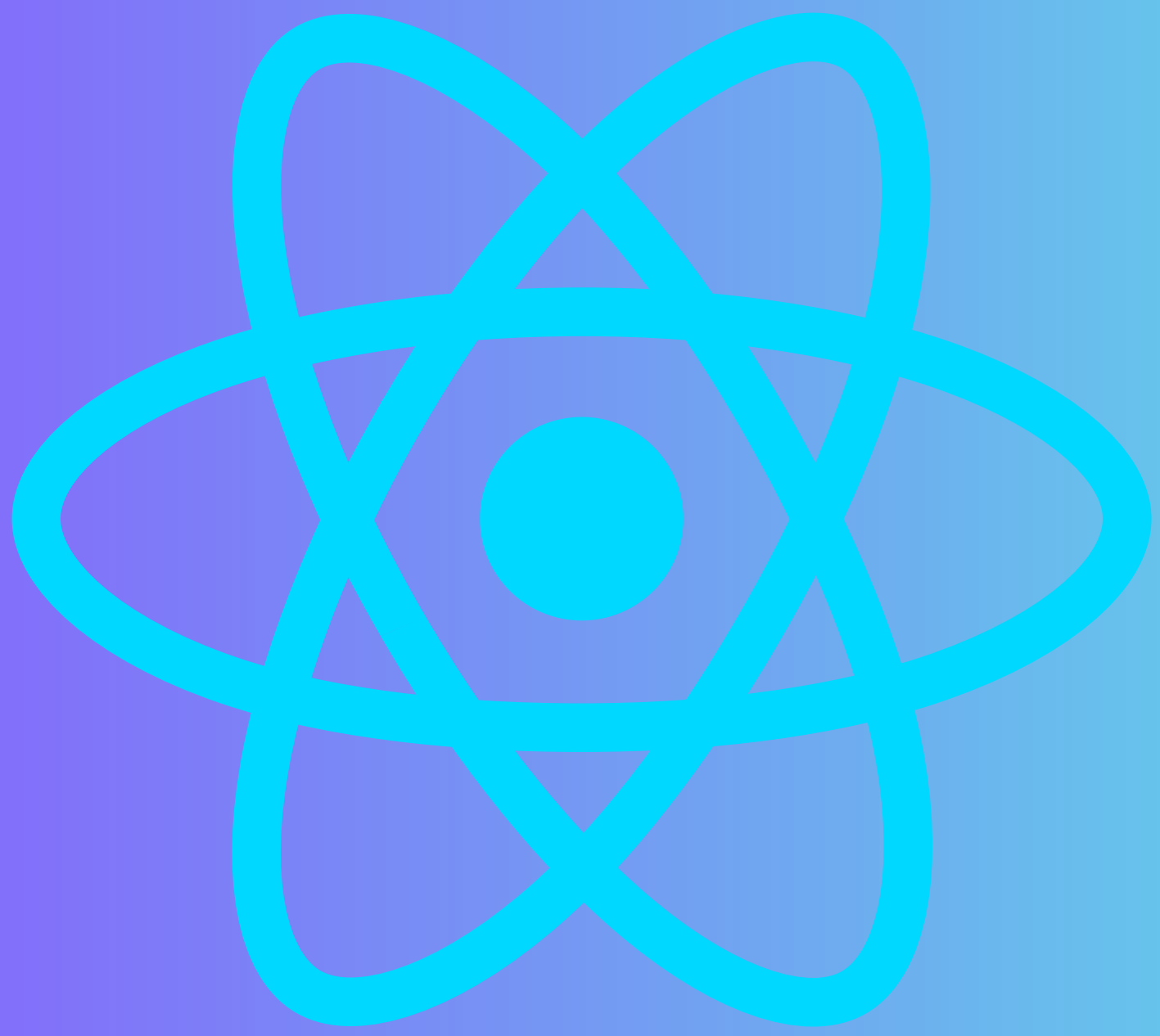


SOLID & MVVM

antes do refactoring

TodoList.ts

```
import React, { useState } from 'react';

function TodoList() {
  const [taskName, setTaskName] = useState<string>('');
  const [tasksList, setTasksList] = useState<Task[]>([]);

  const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setTaskName(event.target.value);
  };

   const handleAddTask = () => {
    if (taskName !== '') {
      const newTask: Task = { id: Math.floor(Math.random() * 10000), name: taskName };
      setTasksList([...tasksList, newTask]);
      setTaskName('');
    }
  };

  const handleDeleteTask = (taskId: number) => {
    setTasksList(tasksList.filter(task => task.id !== taskId));
  }

  return (
    <div>
      <h1>Todo List</h1>
      <input
        type="text"
        value={taskName}
        onChange={handleInputChange}
        placeholder="Add a new task"
      />
      <button onClick={handleAddTask}>Add Task</button>
      <ul>
        {tasksList.map(task => (
          <li key={task.id}>
            {task.name} <button onClick={() => handleDeleteTask(task.id)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

TodoList

index.tsx

1 - passo

trocar state por ref

TodoList.ts

```
const [taskName, setTaskName] = useState<string>("")

 const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
   setTaskName(event.target.value)
 }
```

TodoList.ts

```
const inputRef = useRef<ElementRef<"input">>(null)
```

trocando o state pelo ref evitamos atualizações desnecessárias e simplificamos o jeito de pegar o valor

2 - passo

usar early return

TodoList.ts

```
const handleAddTask = () => {
    if (taskName !== '') {
    }
};
```

TodoList.ts

```
const handleAddTask = () => {
    if (!inputRef.current?.value) return

}
```

a usarmos early return temos maior clareza do que esta acontecendo

3 - passo
trocar state por useReducer

TodoList.ts

```
const [tasksList, setTasksList] = useState<Task[]>([]);
```

TodoList.ts

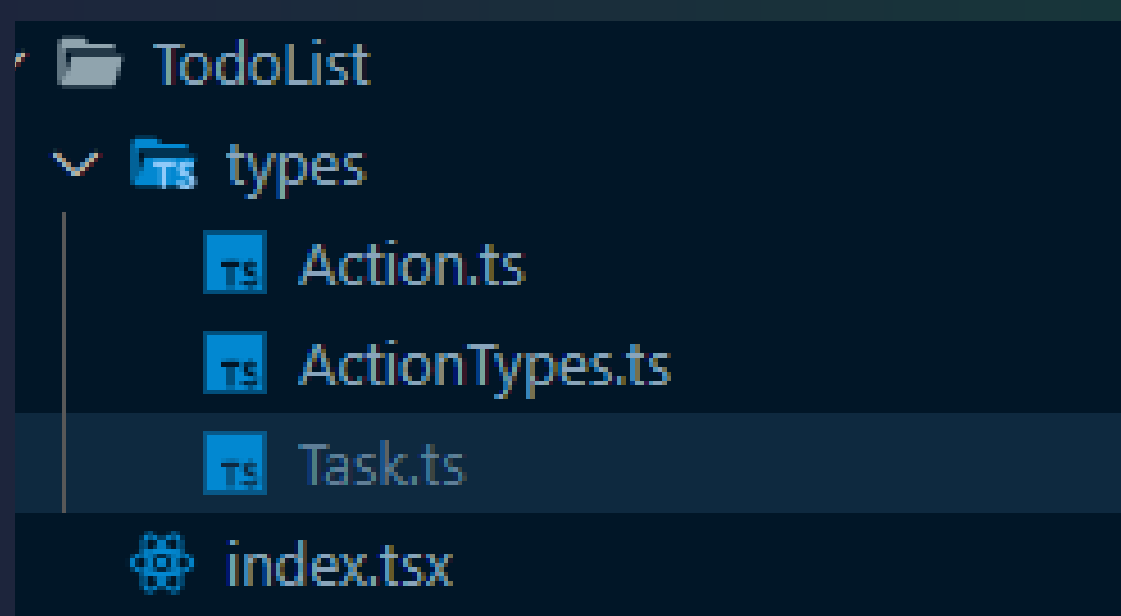
```
const tasksReducer = (state: Task[], action: Action): Task[] => {
  switch (action.type) {
    case ActionTypes.add:
      return [...state, action.task]
    case ActionTypes.delete:
      return state.filter((task) => task.id !== action.taskId)
    default:
      return state
  }
}

const [tasksList, dispatch] = useReducer(tasksReducer, [])
```

ao trocar state por reducer temos uma logica mais separada e clara, pois, passamos as ações para dentro do reducer

separação de types

TodoList.ts

```
export type Action =
  | { type: ActionTypes.add; task: Task }
  | { type: ActionTypes.delete; taskId: number }

export enum ActionTypes {
  add = "ADD_TASK",
  delete = "DELETE_TASK",
}

export type Task = {
  id: number
  name: string
}
```

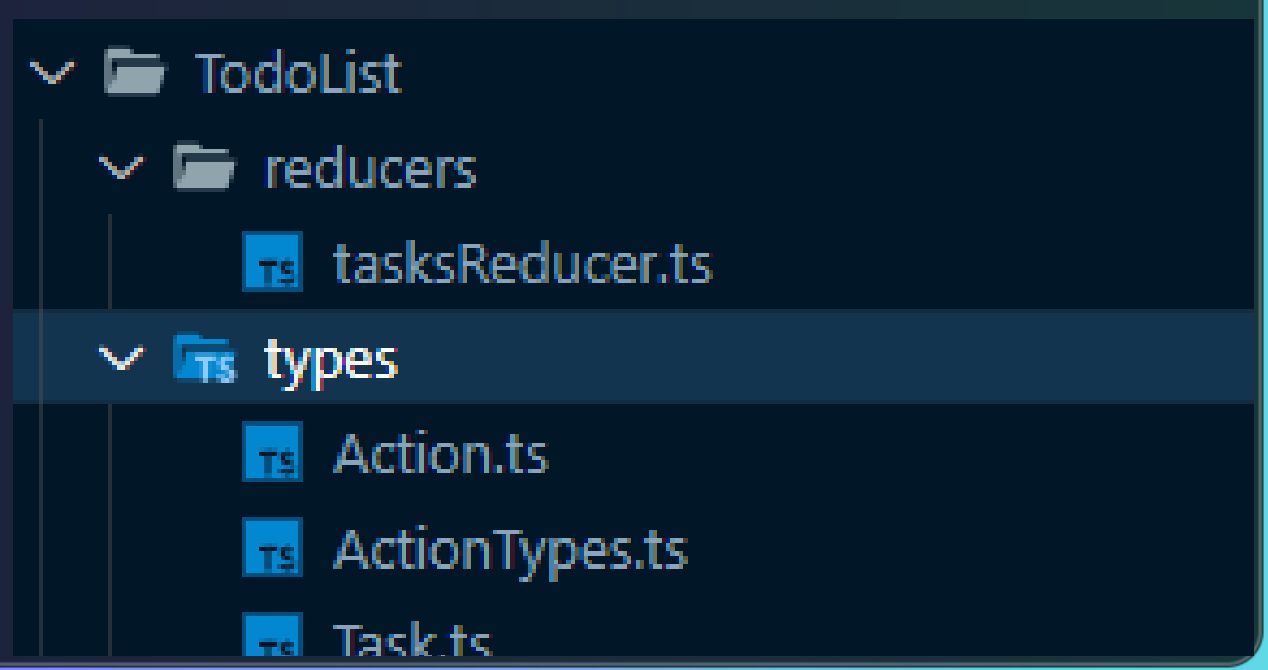


quando estamos dando manutenção queremos olhar pontos específicos com isso é interessante ter divisões claras

separando o reducer

TodoList.ts

```
export const tasksReducer = (state: Task[], action: Action): Task[] => {
  switch (action.type) {
    case ActionTypes.add:
      return [...state, action.task]
    case ActionTypes.delete:
      return state.filter((task) => task.id !== action.taskId)
    default:
      return state
  }
}
```

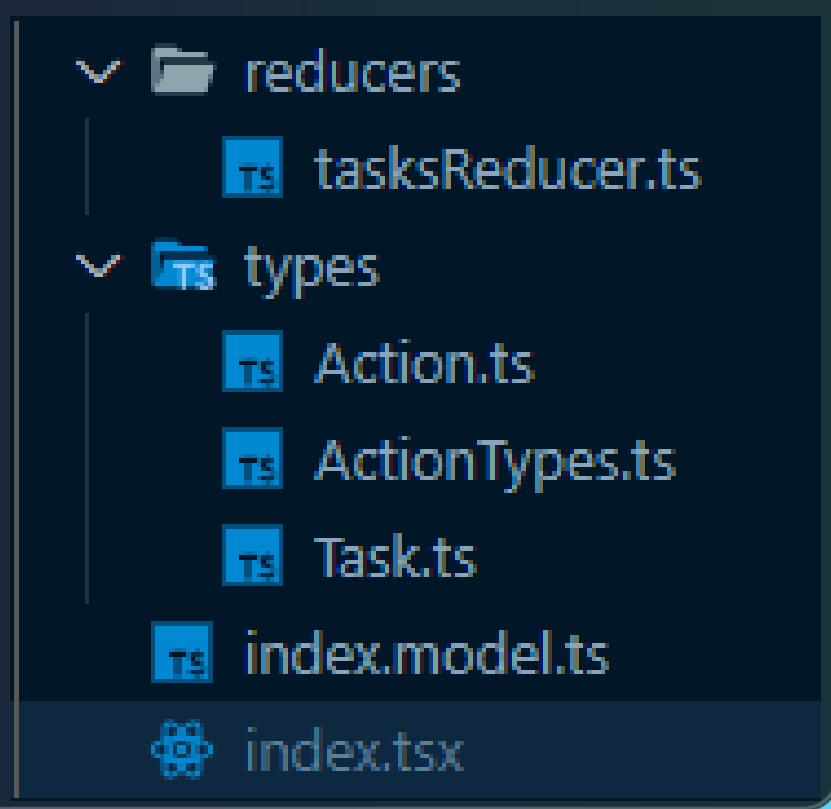


separamos o reducer para assim termos uma melhor organização no nosso index

separação do Model

TodoList.ts

```
const useTodoList = () => {
  const [tasksList, dispatch] = useReducer(tasksReducer, [])
  const inputRef = useRef<HTMLInputElement>(null)

  const handleAddTask = () => {
    if (!inputRef.current?.value) return

    const newTask: Task = {
      id: Math.floor(Math.random() * 10000),
      name: inputRef.current.value,
    }

    dispatch({ type: ActionTypes.add, task: newTask })
    inputRef.current.value = ""
  }

  const handleDeleteTask = (taskId: number) => {
    dispatch({ type: ActionTypes.delete, taskId })
  }

  return {
    inputRef,
    tasksList,
    handleAddTask,
    handleDeleteTask,
  }
}
```

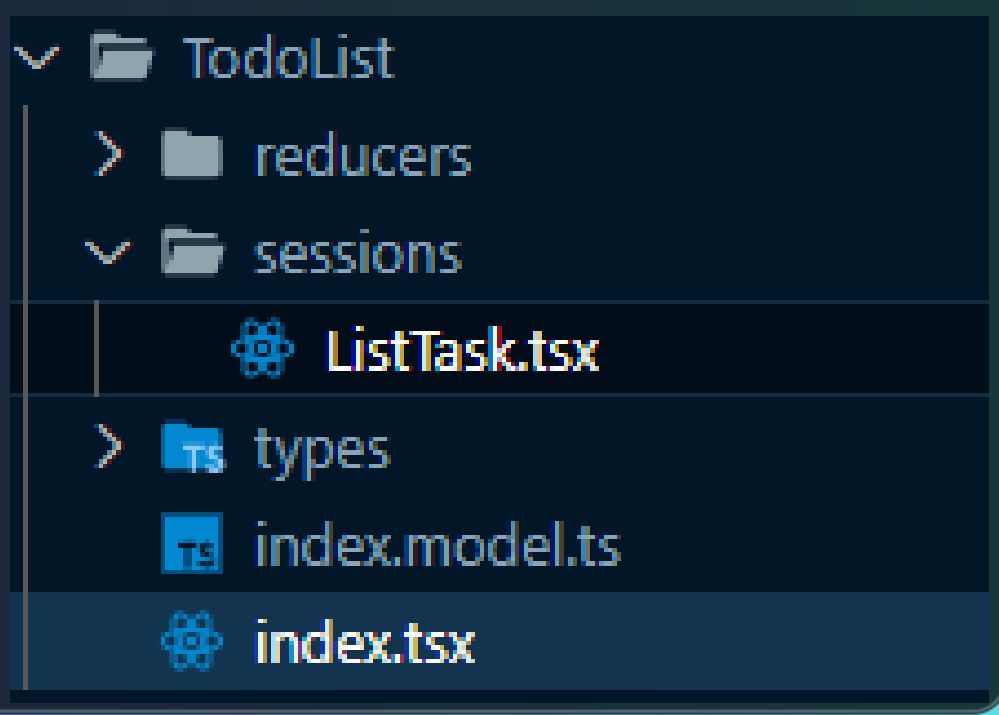


separamos a logica do index

sessions

TodoList.ts

```
export const TodoList = () => {
  const { handleAddTask, handleDeleteTask, tasksList, inputRef } = useTodoList()

  return (
    <div>
      <h1>Todo List</h1>
      <input type="text" ref={inputRef} placeholder="Add a new task" />
      <button onClick={handleAddTask}>Add Task</button>
      <ListTask handleDeleteTask={handleDeleteTask} tasksList={tasksList} />
    </div>
  )
}
```



sessions são trechos da tela que separamos para melhor fragmentação

separação da View

TodoList.ts

```
export const TodoListView = (props: TodoListViewProps) => {
  const { handleAddTask, handleDeleteTask, inputRef, tasksList } = props
  return (
    <div>
      <h1>Todo List</h1>
      <input type="text" ref={inputRef} placeholder="Add a new task" />
      <button onClick={handleAddTask}>Add Task</button>
      <ListTask handleDeleteTask={handleDeleteTask} tasksList={tasksList} />
    </div>
  )
}
```

- sessions
- types
- index.model.ts
- index.tsx
- index.view.tsx

separamos a UI do index

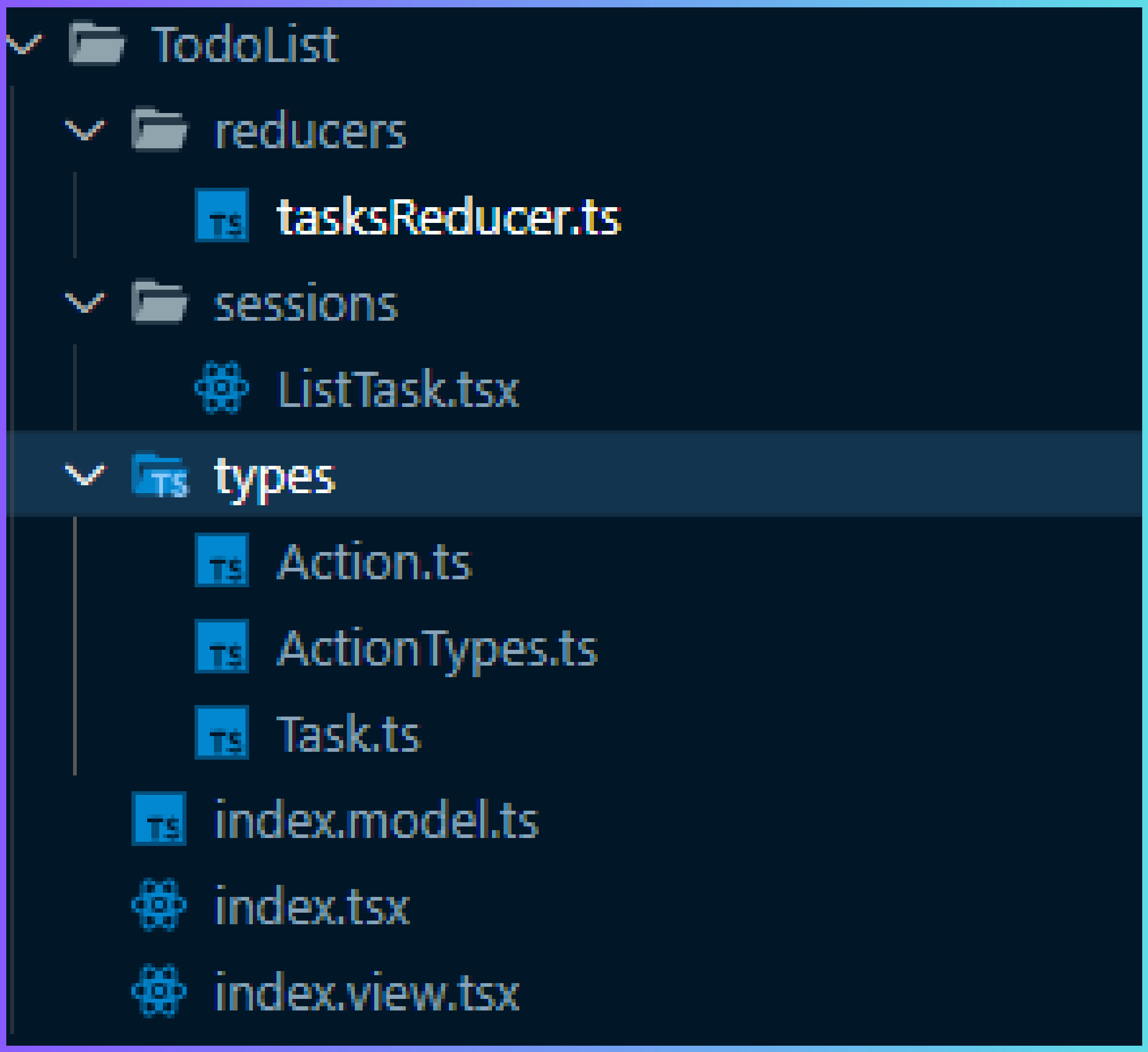
ViewModel

TodoList.ts

```
export const TodoList = () => {
  return <TodoListView {...useTodoList()} />
}
```

camada que une a logica com a UI

RESULTADO FINAL




Enter rows and columns for second mat
endl;
of second
elements of
" << i + 1 << j + 1