Guide to use markdown parser for formatting Server-Side Error

Source Code

```
import { createElement, useEffect, useState } from 'react';

export default function markdownParser() {
  const markers: Record<string, string> = {
    '*': 'b',
    '%': 'em',
    '--': 'del',
    '==': 'ins'
  };

  function recurseOnMarkerFound(
    value: string,
    v1: string[],
    v2: RegExp[]
  ): any {
    function getSplitString(v: string): Array<string> {
      const s = v
        .split(new RegExp(`(${v1.map(e => `${e}.*?${e}`).join('|')})`))
        .filter(Boolean); // seperate matching marker
      return s;
    }
    let s = getSplitString(value);
    let b = s.map(str => {
      const reMarker = v2.find(re => re.test(str));
      if (reMarker === undefined) return str;
      const reMatchArray = str.match(reMarker) ?? [];
      let [, marker, insideMarker] = reMatchArray;
      // console.log(insideMarker, marker, reMatchArray)
      insideMarker = recurseOnMarkerFound(insideMarker, v1, v2);
      // marker is not useful to render the element.
      // return { marker, insideMarker }
      // instead of marker return tag or type
      // type is what React.createElement uses.
      return { type: markers[marker], children: insideMarker };
    });
    // console.log(b)
    return b;
  }
  const lexer = (value: string) => {
    const escMrkr = (function (value) {
      let markers = Object.keys(value);
      markers = markers.map(marker => marker.replace(/./g, '\\$&'));
      return markers;
    })(markers);
    const escMrkrRegExp = (function (value) {
```

```
      return value.map(m => new RegExp(`(${m})(.*?)${m}`));
    })(escMrkr);
    // const mrkrMeta = markerFoundRecurse(value, escMrkr, escMrkrRegExp);
    const mrkrMeta = recurseOnMarkerFound(value, escMrkr, escMrkrRegExp);
    // return { markers, mrkrMeta}
    // markers are not useful to be returning value
    return mrkrMeta;
  };
  const parser = (tokens: any): JSX.Element | null => {
    if (!tokens) {
      return null;
    }
    const innerFn = (input: any) => {
      // case 1 = length is undefined
      // case 2 = length is 0
      if (!input.length) return null;
      const mapFn = (item: any, index: number) => {
        // item should be string or object.
        // case 1 = item is text content
        // case 2 = item is object.
        // all the item is on leaf element are string.
        // while recursion it will be breaker.
        if (typeof item === 'string') return item;
        // object can have two prop type and children
        // value is array or string(text content) and object.
        return createElement(
          item.type ?? 'span',
          { key: index },
          item.children.map(mapFn)
        );
      };
      return input.map(mapFn);
    };
    const elements = innerFn(tokens);
    //  if tag not found return the content
    //  else call createElement with tag and
    //  pass recursion for the inner content.
    return <>{elements}</>;
  };
  return { lexer, parser };
}

export const isMarkedString = (value: string): [boolean, string] => {
  // f works as triggering switch.
  // Possibility that line contain markdown syntax.
  // but running check for every character in string is heavylifting.
  // instead using trigger switch, we an call the parser conditionally
  // to avod unnecessary parsing.
  const tf = '/f';
  if (value.slice(-tf.length) === tf) {
    return [true, value.replace(tf, '')];
  }
```

```
  return [false, value];
};

export const MarkedString = ({
  value,
  log = false
}: {
  value: string;
  log?: boolean;
}) => {
  // TODO: multiple lines delimeter using \n
  const [render, setRender] = useState<any>(value);
  useEffect(() => {
    const [isMarkdown, newValue] = isMarkedString(value);
    if (!isMarkdown) {
      if (log) console.info('[Info]', newValue, 'is not markdown.');
      return;
    }

    function parsingValue(value: string) {
      const { lexer, parser } = markdownParser();
      setRender(parser(lexer(value)));
    }

    parsingValue(newValue);
  }, [value, log]);
  if (isMarkedString(value)[0] === false) {
    // return <span dangerouslySetInnerHTML={{__html:value}} />;
    return value;
  }
  return render;
};
```

It looks like you've shared a TypeScript file that defines a markdown parser and a React component for rendering marked strings. Let me provide a brief overview of the code:

Markdown Parser

The `markdownParser` function exports two main functions: `lexer` and `parser`.

1. **Lexer (`lexer`):**
    - It takes a string as input and tokenizes it based on certain markers defined in the `markers` object.
    - The `escMrkr` array contains escaped versions of the markers for later use in regular expressions.
    - The `escMrkrRegExp` array contains regular expressions for matching the escaped markers in the input string.
    - The `recurseOnMarkerFound` function is a recursive helper function that processes the input string and returns a structured representation of the tokens.
2. **Parser (`parser`):**

    - It takes the output of the lexer (tokens) and recursively converts it into React elements using the `createElement` function.
    - The resulting JSX elements are returned.

Marked String Checker (`isMarkedString`):

The `isMarkedString` function checks if a string ends with a specific trigger (`/f`). If it does, it considers the string as a marked string and returns a tuple with a boolean indicating whether it's a marked string and the string without the trigger.

React Component (`MarkedString`):

This component takes a `value` prop, checks if it's a marked string using `isMarkedString`, and then uses the `markdownParser` and `parser` functions to render the marked string as React elements. It utilizes the `useEffect` hook to handle the parsing when the `value` prop changes.

If the input is not a marked string, it either returns the original string or renders it using `dangerouslySetInnerHTML` (commented out).

Notes:

- The code seems to be well-organized and follows TypeScript conventions.
- The `markers` object defines the mapping of markdown markers to HTML tags.
- The parser is recursive and can handle nested structures.

Todos:

- Implement newline delimeter or line array for multiple lines

If you have any specific questions or if there's something you'd like assistance with, feel free to ask!