Lampiran

Lampiran 1 Kode Program Autocorrect

Berikut adalah kode Pemilihan bahasa

```
// autocorrector.js
import fs from 'fs';
import path from 'path';
import { fileURLToPath } from 'url';
import { loadDictionaries } from './dictUtils.js';
import { levenshtein } from './correctionUtils.js';
import { autocorrectWithBigram } from './autocorrect.js';
const labelToCode = {
  English: 'en',
  Indonesia: 'id',
  Spanish: 'es',
};
const dictionaries = {
  id: {
    kamus: '../Autocorrect/Dictionary/KBBI.txt',
    dataset: '../Autocorrect/Dictionary/CorpusIndo.json',
    bigram: '../Autocorrect/Dictionary/BigramKataUmum.json'
  },
  en: {
    kamus: '../Autocorrect/Dictionary/OID.txt',
    dataset: '../Autocorrect/Dictionary/CorpusEngUnigram.json',
    bigram: '../Autocorrect/Dictionary/BigramKataUmumEng.json'
  },
  es: {
    kamus: '../Autocorrect/Dictionary/Spanish.txt',
    dataset: '../Autocorrect/Dictionary/CorpusSpanUnigram.json',
    bigram: '../Autocorrect/Dictionary/BigramKataUmumSpan.json'
  }
};

export async function autocorrectMessage(sentence, language) {
  if (!dictionaries[labelToCode[language]]) {
    throw new Error(`Bahasa "${language}" tidak didukung.`);
  }

  try {
    const { kamus, kamusList, dataset, bigramDataset, bigramWords } =
loadDictionaries(labelToCode[language], dictionaries);
```

```
    const corrected = autocorrectWithBigram(sentence, kamus, kamusList,
dataset, bigramDataset, bigramWords);
    return corrected;
  } catch (error) {
    throw new Error(`Gagal melakukan autocorrect: ${error.message}`);
  }
}
```

Berikut adalah kode untuk membaca dataset

```
import fs from 'fs';
import path from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

/**
 * Loads dictionaries and datasets based on selected language
 * @param {string} language - Selected language code
 * @param {object} dictionaries - Configuration object for dictionaries
 * @returns {object} Object containing loaded dictionaries and datasets
 */
function loadDictionaries(language, dictionaries) {
  const languageConfig = dictionaries[language];
  const kamusPath = path.join(__dirname, languageConfig.kamus);

  const kamusText = fs.readFileSync(kamusPath, 'utf-8');
  const kamusList = kamusText.split('\n').map(k =>
k.trim().toLowerCase()).filter(Boolean);
  const kamus = new Set(kamusList);

  const datasetPath = path.join(__dirname, languageConfig.dataset);
  const dataset = JSON.parse(fs.readFileSync(datasetPath, 'utf-8'));

  const bigramPath = path.join(__dirname, languageConfig.bigram);
  let bigramDataset = {};

  try {
    bigramDataset = JSON.parse(fs.readFileSync(bigramPath, 'utf-8'));
  } catch (err) {
    console.warn("⚠️ File BigramProbs.json tidak ditemukan atau tidak valid.
Bigram tidak akan digunakan.");
  }
```

```
  const bigramWords = new Set();
  Object.keys(bigramDataset).forEach(bigram => {
    const words = bigram.split(' ');
    if (words.length === 2) {
      bigramWords.add(words[0]);
      bigramWords.add(words[1]);
    }
  });

  console.log(`🔍 Jumlah kata dalam dataset bigram: ${bigramWords.size}`);

  return {
    kamus,
    kamusList,
    dataset,
    bigramDataset,
    bigramWords
  };
}

export { loadDictionaries };
```

Berikut adalah kode untuk menghasilkan kandidat *menggunakkan Levenshtein Distance* dan *N-Gram*

```
/**
 * Calculate Levenshtein distance between two strings
 * @param {string} a - First string
 * @param {string} b - Second string
 * @returns {number} Distance
 */
function levenshtein(a, b) {
  const alen = a.length;
  const blen = b.length;
  const arr = [];

  for (let i = 0; i <= alen; i++) arr[i] = [i];
  for (let j = 0; j <= blen; j++) arr[0][j] = j;

  for (let i = 1; i <= alen; i++) {
    for (let j = 1; j <= blen; j++) {
      const cost = a[i - 1] === b[j - 1] ? 0 : 1;
      arr[i][j] = Math.min(
        arr[i - 1][j] + 1,
        arr[i][j - 1] + 1,
```

```
        arr[i - 1][j - 1] + cost
      );
    }
  }

  return arr[alen][blen];
}

/**
* Check if a string is a number or contains only digits
* @param {string} str - String to check
* @returns {boolean} True if the string is a number
*/
function isNumber(str) {
return !isNaN(str) && /^\d+$/.test(str);
}

/**
* @param {string} word - Word to correct
* @param {string} prevWord - Previous word in sentence
* @param {Set} kamus - Dictionary set
* @param {Array} kamusList - Dictionary list
* @param {Object} dataset - Unigram dataset
* @param {Object} bigramDataset - Bigram dataset
* @param {Set} bigramWords - Set of words from bigram dataset
* @returns {Array} List of candidates
*/
function getCandidates(word, prevWord, kamus, kamusList, dataset,
bigramDataset, bigramWords) {
const punctuation = getPunctuation(word);
const wordWithoutPunctuation = removePunctuation(word);
const lowerWord = wordWithoutPunctuation.toLowerCase();

if (lowerWord === '') {
  return [{
    word: word,
    originalWord: word,
    distance: 0,
    probability: 0,
    bigramProb: 0
  }];
}

if (isNumber(lowerWord)) {
  return [{
    word: wordWithoutPunctuation,
```

```
    originalWord: word,
    distance: 0,
    probability: 0,
    bigramProb: 0,
    punctuation
  }];
}

const candidates = new Map();

if (kamus.has(lowerWord)) {
  return [{
    word: maintainCase(wordWithoutPunctuation, lowerWord),
    originalWord: word,
    distance: 0,
    probability: dataset[lowerWord] || 0,
    bigramProb: 0,
    punctuation
  }];
}

for (const kata of kamusList) {
  const distance = levenshtein(lowerWord, kata);
  if (distance <= 3 && dataset[kata]) {
    candidates.set(kata, {
      word: kata,
      originalWord: word,
      distance,
      probability: dataset[kata],
      punctuation
    });
  }
}

for (const bigramWord of bigramWords) {
  const distance = levenshtein(lowerWord, bigramWord);
  if (distance <= 3) {
    const probability = dataset[bigramWord] || 0;

    let bigramProb = 0;
    if (prevWord) {
      const prevWordClean = removePunctuation(prevWord).toLowerCase();
      const bigram = `${prevWordClean} ${bigramWord}`;
      bigramProb = bigramDataset[bigram] || 0;
    }
```

```
    candidates.set(bigramWord, {
      word: bigramWord,
      originalWord: word,
      distance,
      probability,
      bigramProb,
      punctuation
    });
  }
}

if (candidates.size === 0) {
  return [{
    word: wordWithoutPunctuation,
    originalWord: word,
    distance: 0,
    probability: 0,
    bigramProb: 0,
    punctuation
  }];
}

return Array.from(candidates.values())
  .sort((a, b) => {
    if (a.distance !== b.distance) return a.distance - b.distance;

    const aUnigramScore = a.probability || 0;
    const bUnigramScore = b.probability || 0;
    const aBigramScore = a.bigramProb || 0;
    const bBigramScore = b.bigramProb || 0;

    const aTotalScore = (aBigramScore * 1000) + aUnigramScore;
    const bTotalScore = (bBigramScore * 1000) + bUnigramScore;

    return bTotalScore - aTotalScore;
  })
  .slice(0, 5);
}

/**
* @param {string} word
* @returns {object}
*/
function getPunctuation(word) {
const leadingPunctRegex = /^[^\w\s]+/;
const trailingPunctRegex = /[^\w\s]+$/;
```

```
const leadingMatch = word.match(leadingPunctRegex);
const trailingMatch = word.match(trailingPunctRegex);

const result = {
  leading: leadingMatch ? leadingMatch[0] : '',
  trailing: trailingMatch ? trailingMatch[0] : ''
};

console.log(`Tanda baca untuk "${word}":`, result);
return result;
}

/**
* Remove punctuation from a word
* @param {string} word - Input word
* @returns {string} Word without punctuation
*/
function removePunctuation(word) {
return word.replace(/^[^\w\s]+|[^\w\s]+$/g, '');
}

/**
* Maintain case pattern of original word in corrected word
* @param {string} originalWord - Original word with case
* @param {string} correctedWord - Corrected word
* @returns {string} Corrected word with original case pattern
*/
function maintainCase(originalWord, correctedWord) {
if (originalWord === originalWord.toUpperCase()) {
  return correctedWord.toUpperCase();
}

if (originalWord[0] === originalWord[0].toUpperCase()) {
  return correctedWord.charAt(0).toUpperCase() + correctedWord.slice(1);
}

return correctedWord.toLowerCase();
}

export {
levenshtein,
getCandidates,
getPunctuation,
removePunctuation,
maintainCase,
```

```
isNumber
};
```

Berikut adalah kode untuk perbaikan kata

```
import { getCandidates, maintainCase }  from './correctionUtils.js';

/**
 * @param {string} text - Input text
 * @returns {Array} Array with words and separators alternating
 */
function splitKeepingSeparators(text) {
   const bracketedContents = [];
   const bracketPattern = /(\{[^{}]*\}|\([^()]*\)|\[[^\[\]]*\])/g;

   const textWithPlaceholders = text.replace(bracketPattern, (match) => {
     const placeholder = `__BRACKET_${bracketedContents.length}__`;
     bracketedContents.push(match);
     return placeholder;
   });

   const regex = /(\s+)/;
   const parts = textWithPlaceholders.split(regex);

   return parts.map(part => {
     const bracketMatch = part.match(/__BRACKET_(\d+)__/);
     if (bracketMatch) {
       return bracketedContents[parseInt(bracketMatch[1])];
     }
     return part;
   });
 }

/**
 * Check if a string is a number or contains only digits
 * @param {string} str - String to check
 * @returns {boolean} True if the string is a number
 */
function isNumber(str) {
  // Remove leading/trailing punctuation if any
  const cleanStr = str.replace(/^[^\w\s]+|[^\w\s]+$/g, '');
  // Check if it's a number or contains only digits
  return !isNaN(cleanStr) && /^\d+$/.test(cleanStr);
}

/**
 * @param {string} sentence - Input sentence to correct
```

```
 * @param {Set} kamus - Dictionary set
 * @param {Array} kamusList - Dictionary list
 * @param {Object} dataset - Unigram dataset
 * @param {Object} bigramDataset - Bigram dataset
 * @param {Set} bigramWords - Set of words from bigram dataset
 * @returns {string} Corrected sentence
 */
function autocorrectWithBigram(sentence, kamus, kamusList, dataset,
bigramDataset, bigramWords) {
  const leadingSpace = sentence.match(/^\s*/)[0];
  const trailingSpace = sentence.match(/\s*$/)[0];

  const parts = sentence.trim().split(/\s+/);
  const corrected = [];

  for (let i = 0; i < parts.length; i++) {
    const currentWord = parts[i];
    const prevWord = i > 0 ? corrected[i-1] : null;

    if (!currentWord.trim()) {
      corrected.push(currentWord);
      continue;
    }

    if (/^\{.*\}$|^\(.*\)$|^\[.*\]$/.test(currentWord)) {
      corrected.push(currentWord);
      continue;
    }

    // Skip correction for numbers
    if (isNumber(currentWord)) {
      corrected.push(currentWord);
      continue;
    }

    let candidates = getCandidates(currentWord, prevWord, kamus, kamusList,
dataset, bigramDataset, bigramWords);

    if (candidates.length === 0) {
      corrected.push(currentWord);
      continue;
    }

    let bestCandidate = candidates[0];

    const finalCandidate = applyOriginalCaseAndPunctuation(bestCandidate);
```

```
    corrected.push(finalCandidate);
  }

  return leadingSpace + corrected.join(' ') + trailingSpace;
}
  /**
   * Apply original case pattern and punctuation to corrected word
   * @param {object} candidate - Candidate object
   * @returns {string} Final word with original case and punctuation
   */
  function applyOriginalCaseAndPunctuation(candidate) {
    let finalWord = candidate.word;

    if (candidate.originalWord) {
      finalWord = maintainCase(candidate.originalWord, finalWord);
    }

    if (candidate.punctuation) {
      finalWord = candidate.punctuation.leading + finalWord +
candidate.punctuation.trailing;
    }

    return finalWord;
  }

export {
    autocorrectWithBigram,
    splitKeepingSeparators,
    isNumber
  };
```