Modernizing JS Internationalization with AI Translation Support for Music Blocks

Google Summer of Code 2025

Personal Information

Name: Will Cheng

University: National Tsing Hua University Interdisciplinary Program of Tsing Hua (Major in CS)

Email: will060710@gmail.com

GitHub: https://github.com/weifish0

Country: Taiwan

Time Zone: Taipei (GMT + 8)

Size of the Project: Medium (175 Hours)

Mentor

Walter Bender (@walterbender)

Background

From my early experiences working with students in rural Taiwan to developing educational software, I have always been passionate about leveraging technology to bridge educational gaps. My journey in software development and community engagement has led me to apply for Google Summer of Code (GSoC) with Sugar Labs, a project that deeply aligns with my mission to make learning more accessible and engaging for children worldwide.

Bridging the Gap in Education

In 2024, I joined **Teach For Taiwan** as a summer volunteer, where I taught AI concepts to students in remote areas. Through this experience, I observed a stark contrast: while government policies ensured that each child had access to a tablet, the lack of stable teaching resources and digital learning materials left many students struggling to make effective use of technology. Teachers frequently rotated in and out of these communities, and students lacked exposure to interactive and engaging educational tools.

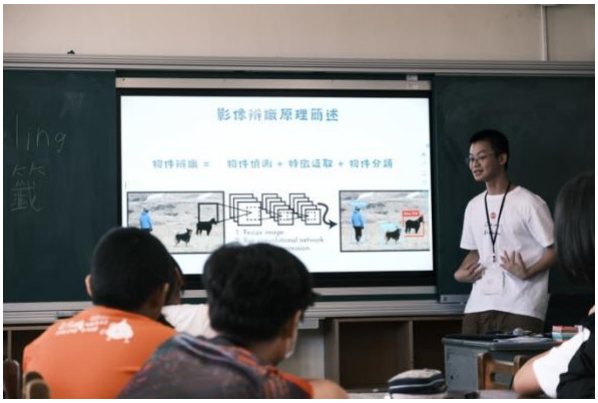

Fig. 1: Teaching students image recognition

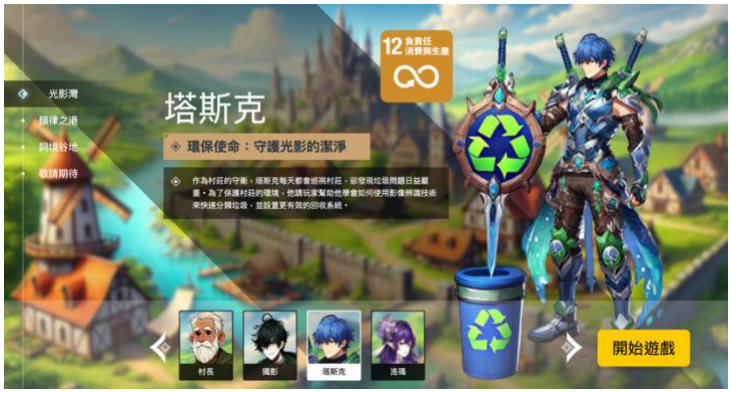

Fig. 2: AI Odyssey NPC page

Determined to address this issue, I created **AI Odyssey**, an interactive gamified platform that teaches AI concepts to elementary school students. Built using **HTML, CSS, JavaScript, Tailwind CSS, and TensorFlow.js**, the project aimed to make AI education fun and accessible. AI Odyssey won first place in the **Samsung Solve for Tomorrow** competition, reinforcing my belief that technology can be a powerful tool to transform learning experiences.

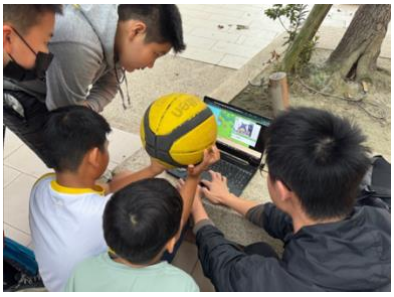

Fig. 3: Students were training image recognition model on the platform

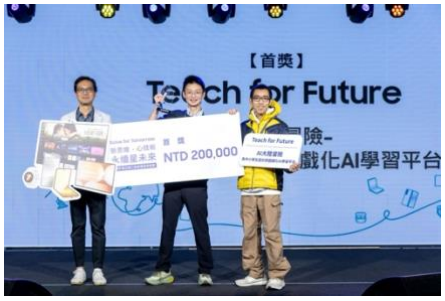

Fig. 4: Samsung Solve for Tomorrow 1st prize

While I have previously worked on independent educational technology projects, I have not yet had the opportunity to contribute to large-scale open-source projects. Through GSoC, I discovered **Sugar Labs**, an organization that aligns perfectly with my mission to use software to reduce educational inequality. Sugar Labs' commitment to fostering learning through open-source software resonates deeply with my belief that technology should be an enabler for all students, regardless of their background.

Why GSoC and Sugar Labs?

My previous work in AI education and community engagement has shown me the transformative power of software in making learning more engaging. However, I recognize that to create a larger and more sustainable impact, I need to collaborate with an open-source community that shares my vision. Sugar Labs has already made a significant impact, with **over 3 million children** using its learning platform and educational software translated into **170 languages**.

By contributing to **Music Blocks' internationalization (i18n) system**, I aim to enhance its multilingual capabilities, making it more accessible to students from diverse linguistic backgrounds. By implementing **AI-assisted translation**, I hope to reduce the manual workload for translators and expand Sugar Labs' reach to more communities worldwide.

Through this project, I seek to combine my expertise in **JavaScript, AI, and web development** with my passion for educational equity. I am excited about the opportunity to contribute to Sugar Labs and learn from experienced mentors while building software that empowers children across the globe.

Synopsis

Music Blocks is an innovative project that enables children to explore music and coding interactively. However, it currently relies on an outdated internationalization system, webL10n.js, which lacks modern features such as pluralization and language-specific formatting.

This project aims to modernize the i18n framework by implementing a contemporary JavaScript internationalization library while integrating AI-powered translation services to enhance multilingual support and reduce manual translation effort.

This proposal outlines a structured approach to researching, evaluating, and implementing a robust i18n solution that aligns with the existing Music Blocks ecosystem. By incorporating AI-assisted translation, the project will improve accessibility and usability for non-English speakers, making Music Blocks more inclusive and engaging for learners worldwide.

Benefits to the Community

Sugar Labs has had a profound impact since its founding in 2008, empowering millions of children and educators worldwide. With over **3 million children** using the Sugar Learning Platform and **11.5 million activities downloaded**, the community has demonstrated a strong commitment to open-source education. Additionally, **170 languages** have been integrated into Sugar's educational software, emphasizing the importance of multilingual support.

This project aligns with Sugar Labs' mission by modernizing its internationalization system, which will:

1. **Enhance Multilingual Support:** With a more advanced i18n framework, Sugar Labs can expand beyond the current 170 languages, improving accuracy, pluralization, and right-to-left (RTL) language support.
2. **Reduce Translation Workload:** AI-assisted translation will generate initial translations, significantly reducing the effort required from the 55+ mentors and contributors working on internationalization.
3. **Improve Scalability and Maintainability:** Replacing webL10n.js with a well-supported framework ensures that future updates remain efficient and sustainable.
4. **Increase Accessibility for Children Worldwide:** With over **3 million learners** already benefiting from Sugar, better translation quality will make learning tools more effective for non-English speakers.
5. **Support Educators and Developers:** More than **344 projects** have been created for teaching and learning through Sugar Labs. A robust translation system will further support teachers in adapting the platform to diverse educational needs.

Goals

Throughout the course of GSoC, my primary focus would be:

1. Researching and selecting the most suitable JavaScript i18n framework for Music Blocks.
2. Evaluating AI translation services and determining the best approach for integration.
3. Developing a migration plan to transition from webL10n.js to the new i18n framework.
4. Implementing core internationalization features, including pluralization and locale-aware formatting.
5. Establishing an AI-assisted translation pipeline to generate initial translations.
6. Ensuring compatibility with PO file workflows to maintain existing translation contributions.
7. Documenting the entire implementation process and providing a developer guide for future contributors.
8. Engaging with the Sugar Labs community to ensure the solution meets the needs of educators and learners worldwide.

Deliverables

By the end of GSoC, I will have successfully implemented the following functionalities:

1. **Migration to a Modern i18n Framework:** Replace the outdated webL10n.js with a well-supported i18n framework such as i18next or FormatJS.
2. **AI-Assisted Translation Pipeline:** Integrate AI translation services (e.g., Google Translate or DeepL) to assist human translators and automate initial translations.
3. **Full UI Translation Support:** Ensure that all user interface components in Music Blocks are fully translatable with the new framework.
4. **Pluralization and Context-Based Translations:** Implement proper pluralization rules and contextual translations for better language accuracy.
5. **Right-to-Left (RTL) Language Support:** Improve support for RTL languages like Arabic and Hebrew.
6. **Locale-Specific Formatting:** Ensure numbers, dates, and other locale-sensitive content are correctly formatted based on user preferences.
7. **Performance Optimization:** Ensure that the new i18n implementation does not introduce significant performance overhead.
8. **Comprehensive Documentation:** Create developer-friendly documentation explaining how to contribute translations and maintain the i18n system.
9. **Community Training and Knowledge Sharing:** Organize a session to educate contributors on the new i18n system and best practices.

Implementation details

Researching Modern JavaScript i18n Frameworks & AI Translation Services

1. Identify Constraints and Requirements

1. **Existing PO File Workflow**
    - Music Blocks uses .po files for managing translations. We need to ensure we can **import, export, or directly use .po files** without losing data.
    - The existing toolchain might rely on gettext utilities, so any new framework should play nicely with or replace those steps in the build process.
2. **Pluralization and Locale Formatting**
    - The new framework must handle different locale rules, e.g., plural forms, date formatting, and right-to-left (RTL) languages. This is critical for broad language support.
3. **AI Translation Integration**
    - The goal is to **automatically generate initial translations** for new or missing strings, then allow human translators to review and refine them.
    - Typical API options:
        - **Google Cloud Translate**
        - **DeepL**
        - **Microsoft Translator**
    - We want an approach that can be automated (scripted) or integrated into a Continuous Integration (CI) workflow.
4. **Browser Compatibility**
    - Music Blocks must run on various browsers and devices (some older). We must check that the i18n solution doesn't break older environments or rely on Node.js-only features.
5. **Community & Maintenance**
    - Ideally, we want a **well-documented**, **actively maintained** library so future contributors can easily jump in.

2. Evaluate Candidate i18n Frameworks

After listing out multiple frameworks, below are **three** that stand out due to popularity and rich ecosystems:

1. **i18next**
    - **Pros**:
        - Very feature-rich (handles plurals, nesting, interpolation).

        - Large community support and ecosystem of plugins (React, Vue, Angular).
        - Can load and parse translations from custom formats using loaders or converters.
    - **PO File Integration**:
        - There is an official plugin/tool called i18next-gettext-converter that can convert .po files to the JSON format i18next typically uses.
    - **AI Integration**:
        - i18next itself does not provide built-in AI translation, but we can automate translation steps with scripts (Node.js or GitHub Actions) that invoke Google, DeepL, etc., then store results in .po or .json.
2. **Polyglot.js**
    - **Pros**:
        - Lightweight, straightforward API.
        - Good for small applications that don't require the full overhead of i18next.
    - **Cons**:
        - May require additional tooling for handling .po files or advanced pluralization rules.
    - **AI Integration**:
        - Similar approach via external scripts, but fewer existing examples or built-in tools compared to i18next.
3. **FormatJS (react-intl / Intl-based libraries)**
    - **Pros**:
        - Built on the native Intl APIs in JavaScript, great for date/number formatting.
        - Good for React projects (react-intl).
    - **Cons**:
        - Less direct support for .po files—likely need a converter or a custom extraction process.
        - Might be overkill if the app is not React-based.

Conclusion on Framework Research

- **i18next** emerges as the most flexible solution given its ecosystem, built-in loaders, and existing community converters for .po files.
- **Next step**: Perform a proof-of-concept (PoC) using i18next + i18next-gettext-converter.

3. PoC: Using i18next + .po Files

The following outlines a **small-scale proof of concept** to ensure .po integration and AI translation can work:

1. **Convert .po to i18next JSON**
    - Use i18next-gettext-converter or gettext-parser to parse .po files.
    - The result is a JSON file that i18next can load at runtime.
    -
```
# Example: i18next-gettext-converter usage
# npm install -g i18next-gettext-converter
i18next-conv \
  -l en \
  -s locales/en.po \
  -t locales/en.json
```
    - This command converts the English en.po into an en.json structured for i18next.
2. **Set Up a Minimal i18next Configuration**

```
// i18n-setup.js (sample)
import i18next from 'i18next';

i18next.init({
  lng: 'en', // default language
  resources: {
    en: {
      translation: require('./locales/en.json'),
    },
    // Additional languages can be added similarly
  },
  interpolation: {
    escapeValue: false, // React/Vue handle XSS if applicable
  },
});

export default i18next;
```

3. **Integrate with Music Blocks Code**
    - Music Blocks use plain vanilla JS so we can import i18next and set up the translations.
    - For each text string, replace the existing webL10n.js calls with i18next.t('translation_key').
4. **Check Pluralization**
    - i18next uses CLDR rules for plural forms. Confirm that the converted .po correctly preserves plural forms, especially in languages like Russian or Arabic.

4. Integrating AI Translation

1. **Script to Auto-Translate Untranslated Strings**
    - Write a **Node.js script** that scans .po or .json files for missing translations, calls an AI translation service, and populates a new .po or .json file with the results.

- Example using Google Cloud Translate (pseudo-code):

```
// translate-missing-strings.js
const fs = require('fs');
const { Translate } = require('@google-cloud/translate').v2;

const translateClient = new Translate({ projectId: 'YOUR_PROJECT_ID' });

async function autoTranslatePO(poFilePath, targetLang) {
  // 1. Parse existing .po
  // 2. Identify msgid that has no msgstr
  // 3. Call AI service for each missing translation
  // 4. Insert AI-generated translations into the .po structure
  // 5. Write updated .po to disk

  // Pseudo steps (not full code):
  const poData = parsePO(fs.readFileSync(poFilePath, 'utf8')); // using e.g. gettext-parser

  for (let message of poData.messages) {
    if (!message.msgstr[0]) {
      const [translation] = await translateClient.translate(message.msgid, targetLang);
      message.msgstr[0] = translation;
    }
  }

  // Save back to .po
  const updatedPO = compilePO(poData);
  fs.writeFileSync(poFilePath, updatedPO);
}

autoTranslatePO('./locales/fr.po', 'fr')
  .then(() => console.log('Translation complete'))
  .catch(console.error);
```

2. **Human Review Process**
   - After auto-generating translations, **human translators** can review in a translation platform (Weblate) or locally using a PO editor.
   - Edits are committed back to the repository.
3. **Continuous Integration Hook**
   - Optionally, integrate into CI to **auto-check for new keys** in .po files. If found, AI-translate them in a staging branch or push to a "translate" branch.

Evaluating and Recommending the Most Suitable Framework and AI Approach

1. Create Evaluation Criteria

1. **Technical Compatibility**
   - Ability to handle .po files (or a straightforward conversion path).

    - Pluralization support (CLDR-compliant).
    - Locale-specific formatting (numbers, dates, RTL support).
    - Browser compatibility (ES5+ or ES6, older mobile devices, etc.).
2. **Ease of Integration**
    - Minimal disruption to existing Music Blocks code base.
    - Clear documentation and straightforward API.
3. **AI Translation Workflow**
    - Straightforward approach to batch auto-translate missing strings.
    - Scriptable in CI/CD or local environment.
4. **Community & Maintenance**
    - Active repository, quick issue resolution, broad user base.
5. **Performance**
    - Lightweight enough for the Music Blocks environment.
    - Reasonable memory usage when storing multiple languages.

2. Conducting Practical Tests / Proof-of-Concept

To ensure the recommendation is grounded in **real-world usage**:

1. **Minimal "Hello World" Project**
    - Set up a small test repo or a separate branch in Music Blocks to try out each candidate framework (e.g., i18next, Polyglot.js) with a handful of translations in .po format.
    - Validate how easily I can **parse .po files** into the format the framework requires, then retrieve translations in the UI.
2. **Performance Baseline**
    - Load 200–300 test strings (common in Music Blocks for commands, UI labels).
    - Measure initialization time, memory footprint, and rendering performance across different browsers.
3. **AI Translation Prototype**
    - Integrate with at least **one** AI service (e.g., Google Translate) for each tested framework.
    - Create or adapt a script to:
        1. Identify missing translations in .po or JSON.
        2. Send them to the AI service.
        3. Populate translations in .po or JSON.
        4. Log or report the results for translator review.
4. **User/Contributor Feedback**
    - Share the test branch or repo with other contributors to get initial feedback on syntax, usability, and maintainability.

3. Comparison Results

Use the tests to **rate** each candidate framework against the criteria. For example:

3.1. i18next

- **PO File Compatibility**
  - Using i18next-gettext-converter or gettext-parser was straightforward.
  - Minimal friction in generating .json from .po.
- **Pluralization & Formatting**
  - Robust CLDR-based rules, tested with languages like **French** (2 forms) and **Russian** (3 forms).
  - Built-in support or easy integration for date/number formatting via the i18next ICU plugin.
- **AI Integration**
  - No direct plugin, but easy to add a Node.js script for Google Translate or DeepL.
  - Verified in a PoC that newly created translations can be merged back into .po files.
- **Community & Docs**
  - Large ecosystem, regularly maintained, many tutorials and examples.
- **Performance**
  - Initial load times for 300 test strings was acceptable (measured via Chrome DevTools).
  - Memory usage stable in older Chrome and Firefox versions.

3.2. Polyglot.js

- **PO File Compatibility**
  - Lacks direct .po integration; would require a custom parser to map .po data to Polyglot's phrase object.
- **Pluralization & Formatting**
  - Basic pluralization for English, but more advanced locale rules would need custom logic or additional libraries.
- **AI Integration**
  - Similar scripting approach to i18next, but no existing tutorials for .po integration.
- **Community & Docs**
  - Smaller user base, less frequent updates.
- **Performance**
  - Lightweight library, minimal overhead.
  - Good choice for smaller or simpler projects.

3.3. FormatJS (e.g., React Intl)

- **PO File Compatibility**
  - Typically uses message descriptors with React components, so would require some bridging to .po files.
  - More friction if the codebase isn't already React-based.
- **Pluralization & Formatting**
  - Uses the native Intl API with full support for locales.
  - Great for date, time, currency formatting.
- **AI Integration**
  - Similar external script approach, but fewer examples for .po ingestion.
- **Community & Docs**
  - Large if using React, but less so if purely a vanilla JS setup.
- **Performance**
  - Generally good, though if not in a React environment, might require extra setup.

4. Recommended Framework: i18next

Based on the PoC and tests:

1. **Direct .po Conversion** – i18next has existing tools that simplify the process.
2. **Robust Plural & Locale Handling** – It aligns well with Music Blocks' global audience.
3. **Scalable Ecosystem** – i18next's extensive plugin system (e.g., ICU formatting, backend loaders) helps to future-proof the project.
4. **Proven AI Scripting** – The community has used external scripts to integrate with major translation APIs, so it's easier to replicate.

Conclusion: i18next is the best fit for Music Blocks' needs, balancing ease of integration, maintainability, and advanced locale support.

5. Recommended AI Translation Approach

After reviewing **Google Cloud Translate**, **DeepL**, and **Microsoft Translator**, choose an API based on:

1. **Language Coverage** – DeepL is strong in European languages, Google covers more worldwide languages, Microsoft has broad coverage but sometimes less accurate in certain languages.
2. **Cost & API Limits** – Check the free tiers or cost per million characters.
3. **Integration** – Libraries or official SDKs for Node.js.

Initial Recommendation:

- **Google Cloud Translate**: It has wide language support, straightforward Node.js library, and is widely used—ideal to start with.
- Keep the approach **modular** so the project can easily switch to a different provider if needed.

6. Ongoing Validation of Assumptions

1. **Deeper PoC / Pilot Integration**
    - Migrate a small set of real UI strings from webL10n.js to i18next in the **development branch**.
    - Confirm that everything functions as expected in the Music Blocks UI (plural forms, placeholders, etc.).
2. **Translator Feedback Loop**
    - Demonstrate the AI-assisted workflow to actual translators.
    - Gather feedback on the .po merges and whether the auto-generated translations are easy to review/fix.
3. **Performance & Browser Testing**
    - Test on older browsers or devices if Music Blocks needs wide coverage (e.g., Chrome 49, older iPads).
    - Ensure minimal load overhead.
4. **Documentation & Collaboration**
    - Document the migration process, including the PoC results, in a **wiki** or README for future contributors.
    - Encourage other contributors to test i18next-based branches to discover any hidden issues early.

Migration Strategy for i18next Integration

1.1. Establish a Migration Branch & Codebase Backup

- **Create a dedicated branch** (e.g., i18next-migration) to isolate the changes from the main codebase.
- **Backup existing .po files and code** that use webL10n.js so I can revert if needed.

1.2. Conversion of .po Files

- **Tool Selection**: Use i18next-gettext-converter or gettext-parser.

- **Automated Conversion Script**: Write a Node.js script to batch-convert existing .po files to JSON format.

```
# Example command-line usage for a single language:
i18next-conv -l en -s locales/en.po -t locales/en.json
```

1.3. Incremental Replacement of Localization Calls

- **Identify Key Areas**: Locate calls to webL10n.js in the code (document.webL10n.get (inputText) in /js/utils.js).
- **Wrap New Calls**: Replace these with i18next.t(inputText). To ease transition, create a shim:

```
// l10nShim.js
import i18next from './i18n-setup'; // i18next initialization file

export function get(key) {
  return i18next.t(key);
}
```

Then, update UI code to use the shim rather than directly calling webL10n.js.

- **Progressive Migration**: Start with non-critical UI elements to test integration, then gradually replace all legacy calls.

1.4. Testing & Validation

- **Automated Tests**: Develop unit and integration tests to ensure that translated strings appear correctly.
- **Browser Compatibility Testing**: Validate that the i18next-based solution works across different devices and browsers (including older ones).
- **Fallback Mechanism**: Retain a fallback to the original system in case a translation is missing or the conversion fails.

2. Designing an AI-Assisted Translation Workflow

2.1. Define the AI Translation Pipeline

- **Workflow Overview**:
    1. **Detection**: Automatically detect untranslated or outdated strings in the JSON translation files.

2. **Translation Request**: Send these strings to the AI translation API (e.g., Google Cloud Translate).
3. **Integration**: Merge the returned translations into the JSON (or revert back to .po for translator review).
4. **Review Process**: Provide a web interface or CLI tool for human translators to review and edit the AI-generated translations.

2.2. AI Translation Script

```
// ai-translate.js
const fs = require('fs');
const path = require('path');
const {Translate} = require('@google-cloud/translate').v2;

// Instantiate the AI translation client
const translate = new Translate({ projectId: 'YOUR_PROJECT_ID' });

// Load translations JSON
const filePath = path.join(__dirname, 'locales', 'fr.json');
let translations = JSON.parse(fs.readFileSync(filePath, 'utf8'));

async function translateMissingStrings(translations, targetLang) {
  const keys = Object.keys(translations);
  for (const key of keys) {
    if (!translations[key] || translations[key].trim() === "") {
      try {
        const [translatedText] = await translate.translate(key, targetLang);
        translations[key] = translatedText;
        console.log(`Translated [${key}] to [${translatedText}]`);
      } catch (err) {
        console.error(`Error translating key "${key}":`, err);
      }
    }
  }
  return translations;
}

translateMissingStrings(translations, 'fr')
  .then(updatedTranslations => {
    fs.writeFileSync(filePath, JSON.stringify(updatedTranslations, null, 2));
    console.log('AI translation complete and file updated.');
  })
  .catch(err => console.error('Translation pipeline error:', err));
```

- **Enhancements**:

- **Batching**: For large numbers of strings, batch API calls to reduce overhead.
- **Caching**: Store translations in a temporary cache to avoid redundant API calls.
- **Logging**: Detailed logging of API responses for review and troubleshooting.

2.3. Human-in-the-Loop Review

- **Translation Interface**:
  - Use the existing translation management tools(Weblate).
  - **Display**: List all AI-generated translations side-by-side with original strings.
  - **Editing**: Allow translators to accept, edit, or reject AI suggestions.
- **Version Control**:
  - Automatically create a commit or pull request for updated translation files.
  - Maintain a changelog for human-edited translations versus AI-generated ones.

2.4. Continuous Integration (CI) Integration

- **CI Pipeline**:
  - Add a step in the CI process to check for new keys in .po or JSON files.
  - Trigger the AI translation script automatically, then open a pull request for review.
- **Automated Alerts**:
  - Notify translators via email when new translations require review.

Pre-GSoC Preparation (Before June 2)

1. **Community Bonding & Familiarization**
   - **Engage with Mentors and Contributors**
     - Join Sugar Labs' communication channels (mailing lists, bi-weekly meeting)
     - Discuss project scope, set expectations, and clarify any uncertainties about existing Music Blocks architecture.
   - **Codebase Exploration**
     - Fork and clone the Music Blocks repository.
     - Identify how webL10n.js is currently integrated (e.g., look for references in index.html, utils.js, etc.).
     - Review the structure of .po files used for translations (e.g., file organization, naming conventions).
   - **Local Development Setup**
     - Ensure a working development environment.

        - Confirm the ability to build and run Music Blocks locally in different browsers.
    - **Preliminary Research**
        - Review documentation for i18next, its plugins (i18next-gettext-converter, i18next-ICU), and possible AI translation APIs (Google Cloud, DeepL, Microsoft).
        - Draft a high-level approach for .po → i18next JSON conversion and how AI translations will be fed back into .po or JSON.
2. **Initial Proof-of-Concept & Technical Validation**
    - **Minimal i18next Project**
        - Set up a tiny test repo to experiment with i18next + i18next-gettext-converter.
        - Convert a small .po file to JSON and confirm that I can retrieve translated strings in a JavaScript demo.
    - **AI Translation Quick Test**
        - Write a simple Node.js script to call Google Cloud Translate or DeepL for a few strings.
        - Verify I can parse those translations back into .po format (optional) or a local JSON.

By the time **June 2** arrives, I'll have a solid **understanding of the codebase**, a **tested local environment**, and a **mini PoC** demonstrating .po conversion and an AI translation call.

12-Week Detailed Timeline (Starting June 2)

Note: Weeks are labeled **W1–W12**, with **W1** beginning on **June 2**. Milestones are designed to be **iterative**, allowing for feedback from mentors and early users.

Week 1 (June 2 – June 8)

1. **Finalize i18next Architecture & Project Setup**
    - Integrate the PoC code into a new branch (e.g., i18next-migration).
    - Align on file structure (where .json translation files will be stored, naming conventions, etc.).
2. **Refine Migration Plan**
    - Create a detailed list of components, modules, or UI screens that require i18n migration.
    - Determine which parts of Music Blocks will be migrated first (e.g., simpler UI elements).
3. **Mentor Review**

- Present the final architecture proposal to the mentor(s) for feedback and sign-off.

Week 2 (June 9 – June 15)

1. **PO to JSON Conversion Tools**
   - Write or refine an **automated script** using i18next-gettext-converter or gettext-parser to batch-convert existing .po files into i18next-compatible JSON.
   - Validate correctness of plural rules in the generated JSON.
2. **Initial Integration**
   - Replace a **small subset** of webL10n.js calls with i18next.t(...) for a few UI elements.
   - Ensure fallback mechanisms exist (e.g., revert to webL10n.js if a key is missing).

Week 3 (June 16 – June 22)

1. **Incremental Migration & Testing**
   - Migrate additional UI sections to i18next, focusing on text that's easy to test.
   - Write **unit tests** or simple functional tests to verify new translations appear correctly.
2. **Basic AI Translation Implementation**
   - Build a **Node.js script** to scan JSON or .po files for missing translations.
   - Integrate with **Google Cloud Translate** (or the selected provider) to fill these gaps.
   - Provide console output or logs to show which keys were auto-translated.

Week 4 (June 23 – June 29)

1. **Human-in-the-Loop Process**
   - Introduce a **review mechanism** for translations. For example, set up **Weblate** or prepare .po files for manual checking.
   - Document how translators can override AI suggestions (e.g., editing .po or using a web interface).
2. **Performance & Browser Checks**
   - Evaluate the **loading time** of i18next with the updated translations.
   - Test in older browsers or devices, ensuring no major regressions compared to webL10n.js.

Week 5 (June 30 – July 6)

1. **Expand Migration to Core Functionalities**

    - Move from simpler UI elements to **core Music Blocks features** (toolbars, menus, main interface).
    - Ensure that **placeholder or dynamic strings** (e.g., text that changes based on user actions) are migrated properly.
2. **Continuous Integration (CI) Integration**
    - Automate detection of new or updated strings in .po/JSON.
    - Configure a CI workflow (e.g., GitHub Actions) to **auto-run the AI translation script** and open a pull request for review.

Week 6 (July 7 – July 13)

1. **Advanced Locale Features**
    - Integrate **pluralization** thoroughly (test languages with complex rules, such as Russian or Polish).
    - Add or verify **right-to-left (RTL) support** for Arabic/Hebrew and ensure the UI flips correctly.
    - Explore **locale-based formatting** for numbers, dates, etc. (possibly using i18next-ICU or a similar plugin).
2. **Midterm Check-In**
    - **Mid-project review** with mentors: demonstrate progress, gather feedback, and confirm if the timeline needs adjustments.

Week 7 (July 14 – July 20)

1. **Refactoring & Code Cleanup**
    - Consolidate newly introduced scripts and confirm consistent code style.
    - Remove any deprecated references to webL10n.js in files fully migrated.
2. **Community Test & Feedback**
    - Encourage early adopters or community members to **test the new translation system**.
    - Collect bug reports or usability issues, especially from translators.

Week 8 (July 21 – July 27)

1. **Enhance AI Translation Scripts**
    - Implement **batch processing** for large sets of strings (reduce the chance of hitting API limits).
    - Consider adding **caching** so repeated or identical strings don’t cost additional API calls.
2. **Bug Fixes & Stabilization**
    - Resolve any known or newly found issues from the Week 7 tests.

- Improve error handling in scripts and UI.

Week 9 (July 28 – August 3)

1. **Finalizing Full Migration**
   - Ensure **all** major UI components are using i18next.
   - Confirm there are no references left to the old webL10n.js system.
2. **Documentation Draft**
   - Start writing a **comprehensive developer guide** on how to add new translations, run AI scripts, and handle localizing new features.
   - Include step-by-step instructions for future contributors.

Week 10 (August 4 – August 10)

1. **Testing & Polishing**
   - Conduct **in-depth QA testing** focusing on edge cases: unusual plural forms, custom date/time formats, extremely long strings, etc.
   - Evaluate **performance** once more, especially with final translations loaded.
2. **Translator Training / Demo**
   - Organize a short **demo session** (virtual meeting or documentation) for translators and the community.
   - Show the new AI-assisted workflow in action (e.g., how to run the script, how to review translations).

Week 11 (August 11 – August 17)

1. **Final Integration & Cleanup**
   - Address any last-minute issues from translator feedback or QA.
   - Refine documentation based on user questions or confusion points.
2. **Backup & Safety Net**
   - Ensure backups of final .json files, .po files, and scripts are stored in the repository.
   - Prepare a rollback plan (if needed).

Week 12 (August 18 – August 24)

1. **Project Wrap-Up**
   - Conduct a **final project review** with mentors, verifying all deliverables (AI pipeline, i18n framework migration, documentation) meet the original goals.
   - Merge or finalize the i18next-migration branch into the main Music Blocks repository (if not already done).

2. **Post-GSoC Handoff**
    - Publish final documentation and tutorials in a prominent location (Wiki, README, or official Sugar Labs docs).
    - Optional: record a **short video walkthrough** showing how to add and translate new strings.

Thank you for considering my proposal. I look forward to contributing to Sugarlabs and making Music Blocks more inclusive!