Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2019, 6(6):52-59

Research Article ISSN: 2394 - 658X

Streamlining Multilingual UI Test Automation with Dynamic Localization

Amit Gupta

Staff Engineer / San Jose, CA
gupta25@gmail.com

ABSTRACT

UI test automation for applications supporting multiple locales poses significant challenges due to varying UI elements, labels, and languages. Ensuring that automated tests accurately reflect the user interface across different languages and regions requires addressing the complexities of internationalization (i18n). Traditional methods, which rely on static and hard coded identifiers and labels, are often inadequate as they can be error-prone and difficult to maintain. This paper proposes an innovative solution that focuses on delivering UI element IDs and labels dynamically at test automation runtime by making a call to a REST API.

The API call passes critical parameters such as application ID, language, and screen ID to fetch the appropriate locale-specific data in real-time. This dynamic approach allows for the decoupling of localization data from the test scripts, thus decentralizing the internationalization problem in UI-based test automation. By doing so, it provides a scalable and maintainable solution that significantly reduces the overhead associated with managing multiple locales.

In addition to detailing the implementation of this dynamic localization system, the paper includes a comprehensive comparison analysis with other available solutions, highlighting their respective advantages and drawbacks. This comparative study underscores the superior flexibility and reliability of the proposed REST API-based approach. Moreover, the paper outlines the various benefits of this solution, including improved synchronization, reduced maintenance efforts, and enhanced scalability.

Finally, the paper presents a future roadmap for further development and optimization of the solution. This includes potential enhancements such as advanced caching mechanisms, integration with continuous integration/continuous deployment (CI/CD) pipelines, and leveraging artificial intelligence to automatically adapt to UI changes. Through these future developments, the proposed solution aims to offer an even more robust framework for internationalized UI test automation, ultimately improving the quality and user experience of global applications.

Keywords: Testing, Software user interfaces, Automation, internationalization, localization, i18n, l10n, UI test automation

---

INTRODUCTION

As applications increasingly cater to global audiences, supporting multiple locales becomes crucial. The ability to offer a seamless user experience across different languages and regions is not just a luxury but a necessity in today's interconnected world. Ensuring consistent functionality across various locales necessitates robust UI test automation strategies that can adapt to the complexities of internationalization (i18n). However, traditional UI test automation faces significant challenges when handling i18n due to the reliance on static and hard coded UI element identifiers and labels.

These traditional methods involve manually updating resource files for each locale, which can be error-prone and labor-intensive. The maintenance overhead increases exponentially as the number of supported locales grows, leading to synchronization issues and inconsistencies in the user interface. Moreover, the static nature of

these methods means that any changes to the UI require corresponding updates in multiple resource files, which can result in delays and potential discrepancies.

The complexities are further compounded when dealing with right-to-left languages, regional date formats, and varying cultural contexts that affect UI design and functionality. In this context, ensuring that automated UI tests are robust and flexible enough to handle such variations becomes a critical challenge. The need for a dynamic and scalable solution is evident to mitigate these issues and enhance the reliability of UI test automation for multi-locale applications.

This paper introduces a dynamic solution to address these challenges by leveraging a REST API to provide locale-specific UI element IDs and labels at runtime. By moving away from static resource files and hardcoded strings, this approach ensures that the test automation framework can dynamically adapt to different locales. The proposed solution decentralizes the internationalization problem, allowing for real-time fetching of UI element identifiers and labels based on the application ID, language, and screen ID. This method not only reduces maintenance overhead but also ensures synchronization and consistency across different locales.

The dynamic retrieval of localization data at runtime through a REST API enables the test automation framework to remain up-to-date with the latest changes in the UI, regardless of the locale. This paper delves into the implementation details of this approach, compares it with traditional methods, and highlights its benefits and future potential. By adopting this solution, developers and testers can achieve a more scalable, maintainable, and reliable UI test automation process, ultimately enhancing the overall user experience for a global audience.

PROBLEM STATEMENT

Traditional UI test automation struggles with:

1. **Static UI Element Identifiers:** Hard Coded IDs and labels make tests brittle and hard to maintain.
2. **Complexity in Managing Multiple Locales:** Manually handling different locales increases the complexity and maintenance overhead.
3. **Synchronization Issues:** Keeping test scripts synchronized with UI changes across different locales is challenging.

PROPOSED SOLUTION

Our solution involves decentralizing the internationalization problem by dynamically fetching UI element IDs and labels at test runtime using a REST API. This method ensures that tests always have the correct locale-specific data without manual updates.

SOLUTION ARCHITECTURE

1. **REST API**: A REST API is developed to return UI element IDs and labels based on application ID, language, and screen ID.
2. **Test Automation Framework Integration**: The test automation framework makes an API call at the start of each test to retrieve the necessary locale-specific data.
3. **Dynamic Data Binding**: The retrieved data is dynamically bound to the UI elements during test execution.

IMPLEMENTATION DETAILS

1. **API Endpoint:**
   URL: /api/ui-elements
   Parameters: applicationId, language, screenId, version
   Response: JSON object containing UI element IDs and labels.
2. **Test Script Modification:** Test scripts are modified to include a setup step that fetches the locale-specific data from the API and binds it to the UI elements.
3. **Data Storage:** The API fetches data from a centralized repository where locale-specific UI elements are maintained.

PROPOSED ARCHITECTURE DIAGRAM

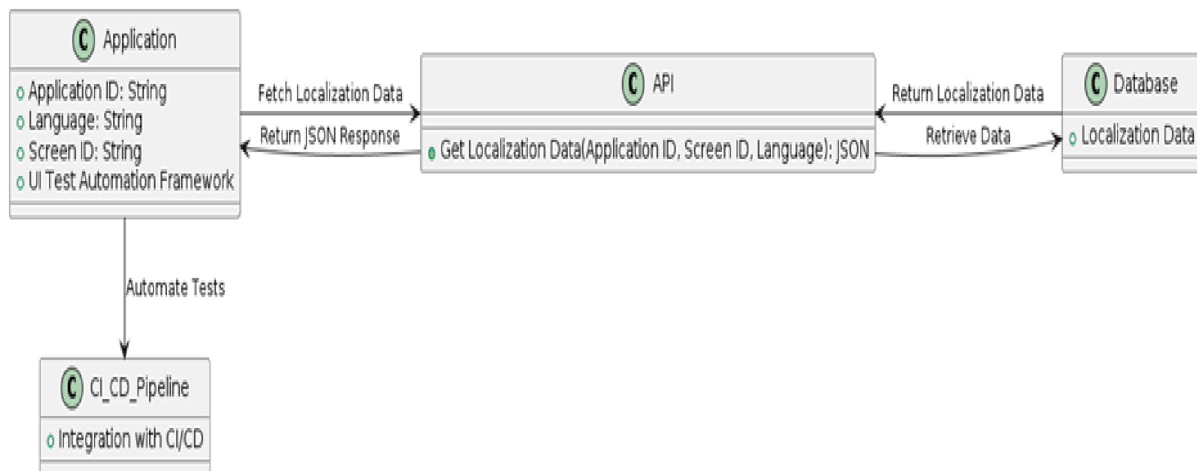

Figure 1: Data Flow of the proposed solution

1. **Application:**
   UI Test Automation Framework: This component initiates the process by making a call to the REST API with parameters such as Application ID, Screen ID, and Language.
2. **API (REST API)**
   Get Localization Data: The REST API endpoint that receives the request from the application. It processes the request and interacts with the database to fetch the required localization data.
3. **Backend Database**
   Localization Data: A centralized repository that stores all the locale-specific data. The API queries this database to get the relevant UI element IDs and labels based on the parameters provided.
4. **CI/CD Pipeline**
   Integration with CI/CD: The CI/CD pipeline integrates with the test automation framework to run tests automatically as part of the build process. It ensures that tests are always run with the latest localization data.

Flow of Data

1. The Application initiates the localization process by calling the API with the required parameters (Application ID, Screen ID, and Language).
2. The API processes the request and queries the backed database to retrieve the relevant localization data.
3. The backend database responds to the API with the required data in JSON format.
4. The API sends the JSON response back to the Application's test automation framework, which uses this data to dynamically bind UI elements during test execution.
5. The CI/CD Pipeline automates the fetching of localization data and the execution of tests, ensuring that the latest localization data is always used.

This architecture ensures a decentralized, scalable, and maintainable approach to handling localization in UI test automation.

Sample Server side implementation with Mock Data

```
import Vapor

func routes(_ app: Application) throws {
app.get("localization", ":bundleId", ":screenId", ":languageId") { req -> AppLocalization in
guard let bundleId = req.parameters.get("bundleId"),
let screenId = req.parameters.get("screenId"),
let languageId = req.parameters.get("languageId") else {
throw Abort(.badRequest)
}

// Mock data for demonstration purposes
let localization = AppLocalization(
bundleIdentifier: bundleId,
screens: [
Screen(
screenId: screenId,
name: "Home",
uiElements: [
UIElement(
identifier: "buttonId",
labels: [
"english": "Click Me",
"spanish": "Haz clic"
]
),
UIElement(
identifier: "labelId",
labels: [
"english": "Welcome",
"spanish": "Bienvenido"
]
)
]
),
Screen(
screenId: "XYZ",
name: "Settings",
uiElements: [
UIElement(
identifier: "switchId",
labels: [
"english": "Enable",
"spanish": "Habilitar"
]
)
]
)
]
```

Comparison with Other Solutions

STATIC RESOURCE FILES

Overview

Static resource files are a common approach to managing localization in applications. These files typically contain key-value pairs where keys are identifiers for UI elements, and values are the localized strings for different languages. Each language has its own resource file.

Pros

Simple to Implement: The approach is straightforward and easy to understand. Developers add new key-value pairs to the resource files as needed.

Immediate Access: Since the resource files are loaded with the application, access to localized strings is instantaneous, without any need for network calls.

Cons

High Maintenance: As the number of locales and UI elements grows, maintaining these files becomes cumbersome. Each change in the UI requires updates in multiple resource files.

Prone to Errors: Manual updates to multiple files can lead to inconsistencies and errors. It's easy to forget to update a string in one of the resource files.

Synchronization Issues: Keeping resource files synchronized across different environments (development, staging, production) and among different team members can be challenging.

Scalability Issues: Adding new languages or making bulk updates is time-consuming and error-prone.

DYNAMIC LOCALE SWITCHING IN CODE

Overview

Dynamic locale switching involves programmatically changing the language and corresponding UI elements at runtime based on user preferences or system settings. This often involves loading locale-specific data from a database or service.

Pros

Reduces Maintenance Overhead: Centralized management of localization data reduces the need for manual updates across multiple files. Changes can be made in one place (e.g., a database) and reflected across the application.

Flexibility: It's easier to support dynamic changes in the application's language settings without restarting or recompiling the application.

Consistent User Experience: Ensures that users see the correct localized content based on their preferences or device settings.

Cons

Significant Refactoring Required: Implementing dynamic locale switching may require significant changes to the existing codebase, especially if the application was originally designed with static resource files.

May Still Face Synchronization Issues: While the centralization of data helps, there can still be synchronization issues between the database and the application, especially if updates are frequent or the application is distributed.

Performance Considerations: Fetching localization data dynamically can introduce latency, especially if the data source is remote or large.

PROPOSED REST API-BASED SOLUTION

Overview

The proposed solution involves decentralizing the internationalization problem by using a REST API to dynamically fetch UI element IDs and labels at runtime. This approach leverages a central service that provides locale-specific data based on parameters such as application bundle ID, screen ID, and language ID.

Pros

Decentralizes the i18n Problem: By moving the responsibility of managing localization data to a central service, the application code remains clean and less complex.

Reduces Maintenance: Updates to localization data are made centrally, reducing the need for widespread manual updates and minimizing the risk of inconsistencies.

Ensures Synchronization: The API ensures that the application always retrieves the latest localization data, reducing the risk of outdated or inconsistent UI elements.
Scalable: Adding support for new locales or updating existing ones is straightforward. The API can be extended to handle new requirements without significant changes to the application code.
Improved Flexibility: Changes to localization data can be made and deployed independently of the application, allowing for quicker updates and fixes.

Cons

Initial Setup Complexity: Setting up the API and the associated infrastructure requires initial effort, including designing the API, setting up the database, and integrating the solution with the application.
Dependency on Network and API Availability: The application relies on the availability of the network and the API. Any downtime or latency in the API can affect the application's ability to fetch localization data.
Performance Considerations: While the approach is scalable, it introduces an additional layer of network calls, which could impact performance, especially in environments with limited connectivity.

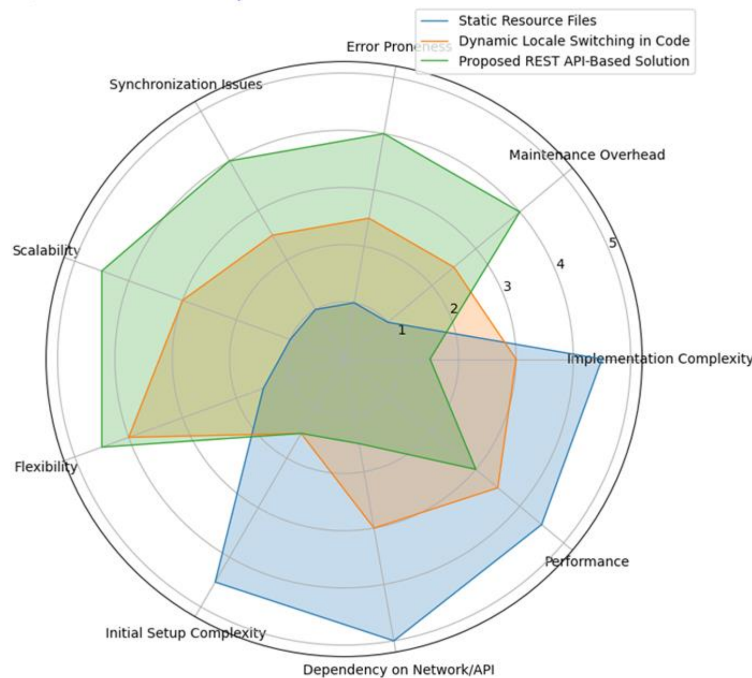

Figure 2: Comparison Analysis of Location Solutions

Table 1: Feature Comparison with other solutions

Feature/Criteria	Static Resource Files	Dynamic Locale Switching in Code	Proposed REST API-Based Solution
Implementation Complexity	Low	Medium	High
Maintenance Overhead	High	Medium	Low
Error Proneness	High	Medium	Low
Synchronization Issues	High	Medium	Low
Scalability	Low	Medium	High
Flexibility	Low	High	High
Initial Setup Complexity	Low	High	High
Dependency on Network/API	No	Optional	Yes
Performance	High (Local Access)	Medium (Depends on Implementation)	Medium (Depends on API/Network)

Table 2: table uses a 1-10 scale to quantify the qualitative assessments

Feature/Criteria	**Static Resource Files**	**Dynamic Locale Switching in Code**	**Proposed REST API-Based Solution**
Implementation Complexity (1: High, 10: Low)	9	6	3
Maintenance Overhead (1: High, 10: Low)	2	5	8
Error Proneness (1: High, 10: Low)	2	5	8
Synchronization Issues (1: High, 10: Low)	2	5	8
Scalability (1: Low, 10: High)	2	6	9
Flexibility (1: Low, 10: High)	3	8	9
Initial Setup Complexity (1: High, 10: Low)	9	3	3
Dependency on Network/API (1: High, 10: Low)	10	6	3
Performance (1: Low, 10: High)	9	7	6

The comparison highlights the trade-offs between different localization strategies. While static resource files offer simplicity, they come with high maintenance and synchronization challenges. Dynamic locale switching in code reduces maintenance but requires significant refactoring and may still face synchronization issues. The proposed REST API-based solution offers a more scalable and maintainable approach, ensuring synchronization and reducing maintenance efforts, albeit with initial setup complexity and dependency on network availability. This solution is well-suited for modern applications that require robust and flexible localization support.

FUTURE ROADMAP

1. Enhanced API Features: Implement caching and fallback mechanisms to improve performance and reliability.
2. Advanced Localization: Support for more complex localization scenarios, such as date and number formats.
3. Integration with CI/CD Pipelines: Automate the fetching of locale-specific data as part of continuous integration and delivery pipelines.
4. AI-Driven Improvements: Utilize AI to automatically detect and adapt to UI changes across different locales, further reducing maintenance efforts.

CONCLUSION

The proposed REST API-based solution for decentralizing the internationalization problem in UI test automation provides a robust, scalable, and maintainable approach. By dynamically fetching UI element IDs and labels at runtime, it ensures synchronization and reduces the complexity of managing multiple locales. This solution offers significant benefits over traditional methods and lays the groundwork for future enhancements in the field of UI test automation.

REFERENCES

[1]. R. Ramler and R. Hoschek, "How to Test in Sixteen Languages? Automation Support for Localization Testing," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, 2017, pp. 542-543, doi: 10.1109/ICST.2017.63.

[2]. R. Ramler and R. Hoschek, "How to Test in Sixteen Languages? Automation Support for Localization Testing," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, 2017, pp. 542-543, doi: 10.1109/ICST.2017.63.

[3]. Ramler, R. and Hoschek, R. (2017b). Process and tool support for internacionalization and localization testing. Product-Focused SW Process Improvement, pages 385–393.

[4]. Jingang Zhou and Kun Yin. 2014. Automated web testing based on textual-visual UI patterns: the UTF approach. SIGSOFT Softw. Eng. Notes 39, 5 (September 2014), 1–6.