Universal Acceptance

Universal Acceptance Readiness Framework

6 April 2020

TABLE OF CONTENTS

Introduction

Universal Acceptance (UA) is about enabling the use of identifier technologies that were not present at the beginning of the Internet deployment, especially when it comes to internationalization. More specifically, and for the purpose of this document, we summarize UA as the conformance status of an application or service that supports the following features:

- Internationalized Domain Name (IDN)
  - Because some software expects a domain name to be in ASCII.
- Email Address Internationalization (EAI)
  - Because some software expects the local part (e.g. the string at the left of the "@" sign for left-to-right scripts) of an email address to be in ASCII.
  - Because some mail servers or services do not expect to receive emails with the local part of the email address in non-ASCII.
- Long top-level domain (TLD) strings
  - Because some software expects a TLD string to be no longer than three characters.
- Added/removed TLD strings
  - Because some software checks TLDs based on an out of date and static list of possible TLDs.

Regarding TLD strings being added or removed, it is worth noting that contrary to some user beliefs, TLDs are often added or removed from the root zone, even on a daily basis. For example, during a 10 day period in November 2019, 7 TLDs were removed from the root.

Identifying whether or not an application of any kind, on any platform, correctly supports UA can be complicated. The purpose of this paper is to define a scope and framework that will identify the issues facing the developer community so that they can work to fix them and increase UA readiness.

Related work on UA is available at https://uasg.tech/information/developers/.

Categories of Applications

For the purpose of this study, applications are either web-based or native as described below.

Web-based application

- The User Interface (UI) is in a browser.
- The browser is either standalone or embedded. Note that some applications seem native but in fact are displaying HTML pages using an embedded browser. An embedded browser may be in desktop applications, cars, wearables, consumer products, and more.

Native application

- Either computer-based or on mobile platforms (mobile phone, tablets, etc.)
- Runs directly on an operating system (OS).
- Does not primarily use a browser or an embedded browser view.

Components

An application is typically made up of various components. Each application category defined above has its own set of components. This work takes a deliberately simplified approach and will only concentrate on the major components of an application. It may mean that sometimes the model is not as accurate for some applications. Splitting the application into components enables the UA compliance work to be applied to various gates between components, as discussed later. Gates can be viewed as functional separations in the application.

Web-based Application

A web-based application is made up of the following components:

- Web browser: The browser is another application; however, it is an integral part of a web application delivery. Note that sometimes web browsers appear as embedded in a native application.
- Front-end: The front-end is the component that creates and enables code to be executed into the browser. It is responsible for creating HTML and allowing HTML, CSS, and Javascript code to be executed in the browser.
- Back-end: The back-end is the component running on a remote server which typically executes more complex tasks, as well as maintaining state and accessing a database.
- Database: The database is the component storing information on a remote server. A database may be managed by a typical SQL database engine or even just a file.
- Filesystem: The filesystem is the place where files are stored on the operating system (OS).
- External Service: Sometimes an application uses an external service to complement its core functionalities. A typical example of an external service is an authentication service provided internally to an enterprise or by a cloud service provider.

Considerations

- Given the nature of current web-based applications, this model is simplified but it does present typical components used in software development and is sufficient for the purpose of this work.
- Modern web browsers offer local storage which is usually limited in size and capabilities, but could be used by application developers to store UA identifiers.
- In modern cloud based applications, the front-end and the back-end are typically stateless, so they can be deployed horizontally dynamically in multiple instances as load requires. Therefore, the front-end and the back-end typically do not involve application data storage.
- Most web sites are typically engineered as applications; therefore, we assume that for the purpose of this work, a web site is a web-based application. Thus, the proposed model may be applied to a web site.

Native Application

A native application is made of the following components:

- User Interface (UI): The visual interface that interacts with the user. This interface is responsible for collecting user input such as domain names or email addresses.
- Internal: The code processing the user's input and providing the functions of the application.
- Database: The database is the component storing application information. A

database may be managed by a typical SQL database engine, or even just a file. The database may be local or remote.
- Filesystem: The filesystem is the place where files are stored on the operating system (OS).
- External Service: Sometimes an application uses an external service to complement its core functionalities. A typical example of an external service is an authentication service provided internally to an enterprise or by a cloud service provider.

Gating Steps

The UA community has defined the following steps within an application to process UA related identifiers:

1. Accept: how an application accepts the user input which consists of UA identifiers.
2. Validate: how an application validates the UA identifier.
3. Process: after validation, how an application processes the UA identifier.
4. Store: after processing, how an application stores the UA identifier.
5. Display: how an application displays the UA identifier.

This work proposes the addition of a step between store and display in order to provide a full sequence of steps that can be used as gates to verify conformance. The new list is as follows:

1. Accept: how an application accepts the user input which consists of UA identifiers.
2. Validate: how an application validates the UA identifier.
3. Process: after validation, how an application processes the UA identifier.
4. Store: after processing, how an application stores the UA identifier.
5. Process: after storing, how an application processes the UA identifier, typically for displaying.
6. Display: how an application displays the UA identifier.

It should be noted that for some applications, some steps or gates may not be relevant. For example, some applications may not store identifiers, therefore steps 4 and 5 would be irrelevant.

Modeling Applications

The two categories of applications are modeled based on their respective components and the gating steps.

Tables below show Y when the component plays a role in the step, and M when the component may play a role.

Web-Based Application

	Accept	Validate	Process	Store	Process	Display
Browser	Y	Y (only IDN in URL)		M (local storage)		Y
Frontend		M	M		M	

Backend		M	M		M	
Database				Y		
Filesystem				Y		
External service		M	M	M	M	

Native Application

	Accept	Validate	Process	Store	Process	Display
UI	Y	M				Y
Internal		M	M		M	
Database				Y		
Filesystem				Y		
External service		M	M	M	M	

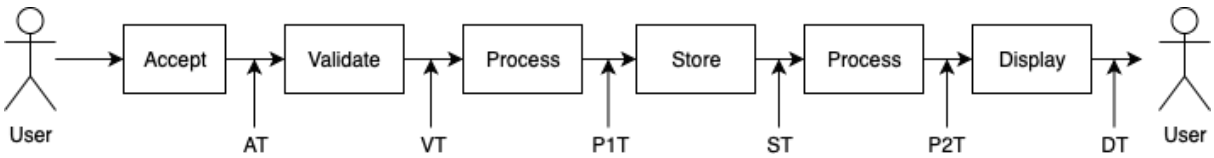
Gating Approach

This work proposes a gating approach to verify UA conformance of an application. This gating approach is based on applying tests at the various steps, now named gates, on the various components.

The following test categories are defined.

Keyword	Expansion
AT	Accept test
VT	Validate test
P1T	Process test on the input
ST	Store test
P2T	Process test on the output
DT	Display test

The following figure shows the proposed gating approach. Each test category is applied at a different step of the application.



Gating Considerations

Modern software architectures are complex with containers, micro services, multi-tier, caches at all levels, cloud services, etc. In fact, an application is a web of components that have complex interactions. Even developers involved in the project have difficulties pinpointing the issues because of the complexity.

While steps (accept, validate, process, store, display) are defined, some may be aggregated together so that there is no separation between some of the steps. For example, validating and processing might take place together within the same method on the identifier object. This work proposes a model, but frontiers between components are often blurred. Modern software architectures often use a variety of cloud services from different cloud service providers. The code of the services themselves, and that of their components, is normally not exposed. Therefore, these services should be considered black boxes with no way to look into them; they can only be tested from the interface that they provide. Given their dominance in the software development market, they must be taken into account in verifying UA conformance.

Examples of such services are third party identity/authentication/authorization services. For a software developer, it is easier to use Google, Facebook, Apple or other pre-existing user authentication services, as it enables the developer to focus on aspects which will differentiate the relevant service without having to incur time and cost in 'reinventing the wheel' for commonly occurring functions. It also provides a more seamless experience for the end-users. As domains or email addresses are often used as identifiers, or as backup identifiers, they become a significant part of UA conformance of an application when such applications uses these services. Email address are not only used as unique identifiers for users, but there are often backup emails used for restoring the account or resetting the password which also must support UA conformance.

Web-Based Application Components

This section lists some web-based application components with examples. This list is not exhaustive.

Category	Examples	Type of Application
Browser	Chrome, Firefox, Edge, Safari, Opera, Brave, UC, 360, Sogou, Baidu, Tencent, QQ	Component of web-based application
Web server	nginx, Apache, lighttpd, tomcat, IIS	Backend component of web-based application
TLS Certificates	openssl, letsencrypt	Backend component of web-based application
Backend web framework/libraries	Django/Python, Flask/Python, Spring/Java, Express/NodeJS, Ruby on Rails,	Component of web-based application

	Lumen/PHP, .Net, Laravel	
Frontend web framework/libraries	Angular, React, Vue.js, Backbone, Ember Bootstrap	Component of web-based application

Applications

This section list some applications, either web-based or native, that would be candidates for UA conformance testing.

Email (webmail)	Gmail, Xgenplus, Coremail, Hotmail/Outlook/Live, Yahoo, Datamail, Mailrelay	Web-based application
Email (native client (MUA, MSA))	Gmail, Outlook, Xgenplus, Foxmail	Native
Email (server (M*A)	MSexchange,postfix, Dovecot, Xgenplus, courier	Native
Email (service provider)	Gmail, Hotmail/Outlook/Live, Yahoo, datamail CoreMail, QQ mail; 163/126 mail; China Mobile 139.com	Native
Antispam (AS)	Spamjadoo, Comodo, SpamAssassin, SpamTitan	Native
OS tools	Curl, Wget, nslookup, Dig, Telnet, SSH, Host, Hostname, SCP, Ping, Tracert, PathPing	Native
CMS	WordPress, Joomla, Drupal, Typo3, Discuz, Pageadmin	Web-based application
Social Media Web Apps	Facebook, Twitter, Pinterest, Instagram, LinkedIn, WhatsApp, WeChat, Sina Weibo, QQ	Web-based application
Social Media Apps	Facebook, Twitter, Pinterest, Instagram, LinkedIn, WhatsApp, WeChat, Sina Weibo, QQ	Native
E-Commerce Web Apps	Amazon, Alibaba, Ebay	Web-based application

It is worth noting that the same application is often delivered in two flavors: a web-based and a native version. For example, Facebook is available as a web-based application and as a native mobile application. However, for the purpose of UA conformance, those should be considered separate and different since one may behave differently to the other regarding UA.

Expected Behaviors

This section describes the expected behavior of the applications at each gate.

Accept

- All forms possible: i.e. IDN – either U-Label or A-Label.

Validate

- Normalize if non-ASCII
- Optionally:
  - Verify if valid IDN
  - Verify if label is too long (> 63 octets, A-label if IDN)
- Maybe (has pros and cons):
  - Verify if TLD is active

Process on Input

- Does not truncate
- Does not inadvertently convert to ASCII. For example, converting ‘é’ to ‘e’.

Store

- EAI: local part UTF8
- IDN: either U-Label or A-Label
- Long/new TLD: as is; no truncation

Process on Output

Display

- Should display in UTF8
- May also display IDN as A-label

Tests

This section lists the various high-level tests that can be applied at each gate. The various possible datasets are described in the UASG-004 document.

AT: Accept Test

- EAI: verify if it accepts any UTF8
- IDN: verify if it accepts any UTF8
- Long/new TLD: verify if it accepts any TLD as long label

VT: Validate Test

- Verify if it normalizes a non-normalized UTF8 label
- Verify if it validates a valid/non-valid domain name
- Verify if it validates a valid/non-valid IDN
- Verify if it validates a valid/non-valid long/new TLD

P1T: Process on Input Test

- Verify that the input of the processing does not change the output, unless it normalizes the string.

ST: Store test

- EAI:

    - Given good EAI input, verify that the database content for that input is correct: i.e. either identical or normalized version of the local part. The domain part, if IDN, could be either A-Label or U-Label form.
    - Given bad EAI input, verify that the database content for that input is not saved.
- IDN:
    - Given good IDN input, verify that the database content is either A-Label or U-Label form.
    - Given bad IDN input, verify that the database content for that input is not saved.
- Long/new TLD strings:
    - Given long TLD string as input, verify that the database content is not truncated.
    - Given too long TLD string as input, verify that the database content for that input is not saved.
    - Given new TLD string as input, verify the database content contains the new TLD.

P2T: Process on Output Test

- Given a good value in the database, verify that the value going to be displayed by the UI is correct: local part is UTF8, IDN is in U-Label format, long string TLD is not truncated, new TLD string is processed.

DT: Display Test

- Given a good value in the database, verify that the displayed value by the UI is correct: local part is UTF8, IDN is in U-Label format, long string TLD is not truncated, new TLD (from the root) is shown, removed TLD (from the root) is not shown.

Normalization

Normalization is required since the same string can be represented by different codepoints. Unicode defines various normalization forms[UTR15]. In absence of specifics, developers may use Normalization Form C (NFC).

Proposed Approach

The proposed approach to verify UA compliance of an application is as follows:

- Identify the components used in the application.
- Verify the UA compliance of each component.
- Test the application for UA compliance, from an input/output perspective without looking “under the hood.”
- If tests results are negative, then investigate by applying the gating approach to the various interfaces of the components, as described before.

It should be noted that a component may not be UA compliant, but that does not mean the whole application is not UA compliant. This is because the developers of the application may have added another layer above the base component to support UA. For example, as shown by testing Java libraries for UA compliance, the JAVA base JRE is not UA compliant as it supports IDNA2003. However, a developer may use and add a new library that supports UA, so the fact that an application is implemented in Java does not mean it is not UA compliant.

This framework is intended for testing software applications. It may not be applicable as is for some other use cases such as libraries or infrastructure components. SMS messaging is an example of an infrastructure that may carry internationalized identifiers, but does not necessarily apply directly with this framework.

Web-Based Application: CMS Sub-Category

The following is an application of the approach to the web-based application category of Content Management Systems (CMS). This is a non-comprehensive list of CMS, but provides the most popular ranked by various sites. The market leader is WordPress.

These are all open-source. The code can be downloaded and installed on a server, or many can also be used on hosting services.

	Browser	Frontend	Backend	Database	Gates/test control points
WordPress	Any	PHP	PHP/Apache	MySQL/MariaDB	AT, VT, P1T, ST, P2T, DT
Joomla	Any	PHP	PHP/Apache	MySQL/MariaDB	AT, VT, P1T, ST, P2T, DT
Drupal	Any	PHP	PHP/Apache /nginx	MySQL/MariaDB/ PostgreSQL/SQLite	AT, VT, P1T, ST, P2T, DT
Typo3	Any	PHP	PHP/Apache /nginx	MySQL/MariaDB/ PostgreSQL/SQLite	AT, VT, P1T, ST, P2T, DT

One can see that these share the same architecture and same components. Same tests can be applied to all.

Summary

This work provides a framework for scoping UA conformance work on software applications. It defines two categories of application and a series of gates for testing the application against various tests.

References

[UASG] https://uasg.tech
[UASG-004] Use Cases for UA Readiness Evaluation, April 2017, https://uasg.tech/wp-content/uploads/documents/UASG004-en-digital.pdf
[UTR15] Unicode Normalization Forms, Unicode® Standard Annex #15, https://unicode.org/reports/tr15/