Universal Acceptance (UA) for Java Developers

July 2020

Target Audience, Objectives and Goal

- Target audience:
    - Java developers
    - Software project managers
    - CTO
- Objectives:
    - Understand base key concepts related to internationalized domain names and email
    - Understand issues with using plain Java for validating and using internationalized domain names and email addresses
    - Identify which libraries are appropriate for the use case
    - Know how to use the libraries
    - Develop best current practices for UA compliant applications
- Goal:
    - develop UA-compliant Java applications

Plan

- Problem statement
- Key fundamental concepts related to UA
  - Unicode
  - IDN
  - EAI
- Validating UA identifiers Input
- Using UA identifiers:
  - Resolving domain names
  - Sending Email
- Best practices
- Conclusion
- References

Problem Statement

Validating Email: A Real Example

- A company built a website where consumers can subscribe to its offering via their email.
- Since subscription form is user input, developers validated the email address before trying to send the email:

```
if (isEmailValid(email)) {
  subscribe();
} else {
  throw new Exception("Invalid email address, please review it and submit again");
}
```

ICANN

Validating Email

- Developers went to Stackoverflow and found a regex to perform the validation:

20 Answers

Active | Oldest | Votes

242

FWIW, here is the Java code we use to validate email addresses. The Regexp's are very similar:

```
public static final Pattern VALID_EMAIL_ADDRESS_REGEX =
    Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}$", Pattern.CASE_INSENS

public static boolean validate(String emailStr) {
        Matcher matcher = VALID_EMAIL_ADDRESS_REGEX.matcher(emailStr);
        return matcher.find();
}
```

Validating Email

- The company went international and expanded its offerings into non-English regions
- The sales team then started to log bugs about customers unable to subscribe, the website, as they say, always returns: "Invalid email address, ..." to valid customer emails.
- Developers digged into the website logs and found an email reproducing the problem:

  普遍接受-测试@普遍接受-测试.世界

- Maybe a simple regex is not a good idea and the dev team started searching a proper email validation library

Validating Email

- Dev team realized that the package com.sun.mail:javax.mail:1.5.6 used to send email subscription confirmation via SMTP had already a "validate" function. They rewrote the isEmailValid method:

```
public static boolean isEmailValid(String email) {
  try {
    var iEmail = new javax.mail.internet.InternetAddress(email);
    iEmail.validate();
    return true;
  } catch (AddressException e) {
    return false;
  }
}
```

Validating Email

- However, they realized the method is still failing. They saw that this internationalization feature was corrected in a newer version so they upgraded to -> com.sun.mail:jakarta.mail:1.6.5

```
public static boolean isEmailValid(String email) {
  try {
    var iEmail = new javax.mail.internet.InternetAddress(email);
    iEmail.validate();
    return true;
  } catch (AddressException e) {
    return false;
  }
}
```

- Finally, by inspecting these fixes in javamail and its renamed version jakartamail, they realized they needed to also modify the subscribe function and their SMTP server to support a new "SMTPUTF8" flag. Is the bug fixed now?

Validating Email

- Later on, a security audit was performed on the web application. The external security auditors gave bad scores to email validation and provided a standard recommended fix, provied by a internationally recognized security authority: Open Web Application Security Project (OWASP). The OWASP recommended regex for email is:
  - ^[a-zA-Z0-9_+&*-]+(?:\.[a-zA-Z0-9_+&*-]+)*@(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,7}$]]
- Should the company implements the recommended security fix?

Key Fundamentals

Unicode

- encoding glyphs into codepoints
- in specifications, codepoints are shown in hex using the U+XXXX notation
- codepoints are typically carried using the UTF-8 format
  - variable number of bytes for a single codepoint.
  - ascii is used as is
  - gold standard for carrying Unicode codepoints, in web, protocols, etc...
- multiple ways to use a glyph:
  - “è” = U+00E8
  - “e`” = “è” = U+0065 U+0300
  - Normalization is a process to insure that whatever the user types, the end representation will be the same.
    - for the two entries above, Normalization Form C(NFC) will generate U+00E8 for both

Universal Acceptance (UA)

- How to appropriately support internationalized identifiers and long TLDs
  - internationalized identifiers:
    - idn
    - eai
  - longer string TLDs:
    - some time ago, TLDs were two or three characters long (i.e. .ca, .com). Then TLDs were longer strings (i.e. .info, .google).
    - some applications are still verifying that the TLD entered by a user has a maximum of 3 characters…
  - added/removed TLDs:
    - TLDs come and go on a daily basis. Some applications are verifying the correctness of a TLD based on a static list which is not the latest one.

Domain Names

- domain name is an ordered set of labels: a.b.c.d
- a top-level domain is the rightmost label
- Domain Name System(DNS) is the distributed database and service for querying domain name records
- a domain name may have multiple DNS records such as:
  - IPv4 address for that domain name
  - IPv6 address for that domain name
  - hostname of the email server responsible for that domain name
  - ...
- a zone is the list of domain name records (called Resource Records(RR)) for the labels under another label (a bit simplified…)

ICANN

Internationalized Domain Names (IDN)

- enables using non-ASCII characters for any label of a domain name
  - not all labels of a domain name may be internationalized

- ex: exâmple.ca

- user uses the idn version, but the idn is converted into ascii
  - exâmple => exmple-xta => xn--exmple-xta
  - the xn-- prefix is added to identify an IDN

Internationalized Domain Names (IDN) (cont.)

- example process of using idn:
  - user enters in a browser: http://exâmple.ca
  - browser do normalization on the user entry
  - browser convert exâmple.ca in an ASCII compatible representation, called Punycode[RFC3492] and adds ‘xn--’ in front of it.
    - xn--exmple-xta.ca
  - browser calls the DNS for getting the IP address of xn--exmple-xta.ca

Internationalized Domain Names (IDN) (cont.)

- The protocol is named IDN for Applications (IDNA)
  - two versions: IDNA2003 and IDNA2008. Latter is the currently used one.
- U-Label is the Unicode native representation of an IDN label: viagénie
- A-Label is the Punycode representation of an IDN label: xn--viagnie-eya

ICANN

Two IDN Standards

- First version: named IDNA2003 (RFC3490)
  - Algorithms named StringPrep(RFC3454) and NamePrep(RFC3491).
  - Encoding in ascii uses Punycode (RFC3492).
  - identifies an idn by adding xn-- to the punycode encoding of the domain
  - Was defined against Unicode 3.2 (March 2002)
  - Provisions to use new characters (i.e. added after Unicode 3.2) as is

Two IDN Standards

- Second (and latest) version: named IDNA2008. No more using Stringprep and Nameprep, however encoding in ascii still uses Punycode and the same prefix (xn--). IDNA2008 is much more agile to support new characters added by Unicode over time.
- IDNA2008 is more restrictive than IDNA2003: valid domains under IDNA2003 may not be valid under IDNA2008.
- Therefore, it is highly recommended to use the IDNA2008 standard.
- Recommendation: make sure the libraries you are using are based on IDNA2008.

Two IDN Standards

- Example: ſſ.example (i.e. U+017F U+017F)
  - valid in IDNA2003 and mapped into ss.example
  - invalid in IDNA2008
- To “facilitate” the transition from IDNA2003 to IDNA2008, Unicode defined a transitional feature in the UTS 46 specification. Under UTS 46, ſſ.example is mapped to ss.example. IETF does not recommend the use of UTS 46. ICANN supports only IDNA2008, therefore an IDNA2003 or a UTS46 transitional domain are not valid.

Public Suffix List (PSL)

- An attempt to help developers to know if a TLD (or any sub-tld) is allocated or not. Volunteer-based (Mozilla). A TLD has to (manually) register itself and related rules to the PSL maintainers. Governance model is basic.
- The goal has been to enable browsers to check the validity of domain names and tlds right in the address bar and even suggest/propose/correct tlds based on a static list, instead of generating DNS queries.
- https://publicsuffix.org
- If used,
  - a developer has to keep its local copy current.
  - Any TLD not in the public suffix list would then be considered as non-existant.
  - A TLD may not be in the public suffix list because the registry did not register it to the public suffix list, or because of some PSL policies, or because the suffix list in the software is not current.

Universal Acceptance (UA)

- How to appropriately support internationalized identifiers and long TLDs
  - internationalized identifiers:
    - idn
    - eai

ICANN

Universal Acceptance (UA) (cont.)

- longer string TLDs:
  - some time ago, TLDs were two or three characters long (i.e. .ca, .com). Then TLDs were longer strings (i.e. .info, .google).
  - some applications are still verifying that the TLD entered by a user has a maximum of 3 characters…

Universal Acceptance (UA) (cont.)

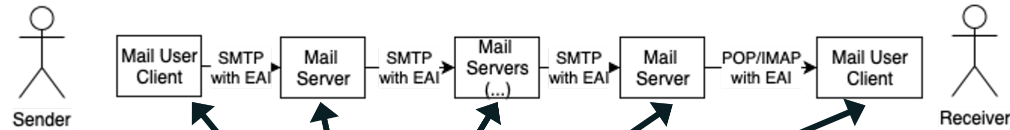
added/removed TLDs:

- TLDs come and go often. Some applications are verifying the correctness of a TLD based on a static list which is not the latest one.

Email Address Internationalization

- email syntax: leftside@domainname
- domainname can be internationalized as an IDN
- leftside (also known as local part/mailbox name) with Unicode (UTF8) is **EAI**
- examples: kévin@example.org, すし@快手.游戏
- side effect: Mail headers need to be updated too to support EAI. Mail headers are used by mail software to get more information on how to deliver email.
- As not every email servers are supporting EAI, a negotiation protocol is used to only send EAI when the target server supports it. If not, then it falls back. The SMTPUTF8 option is used within the mail transfer protocol (SMTP: Simple Mail Transport Protocol)

EAI Delivery Path Considerations

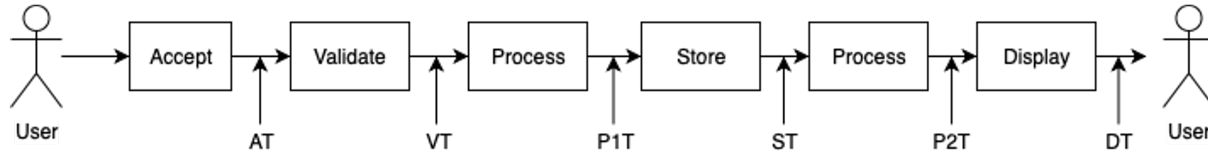


- to send and receive an email with EAI:
  - all email parties involved in the delivery path have to be updated for EAI support
  - if a single SMTP server in the path does not support EAI, then the email is not delivered.

Application Components

Application Components Model

- Based on UASG026, this model of application components is a simplified model to put emphasis to the processing of internationalized identifiers.
- Each gate has its own set of requirements and processing.



Validating User Input

- Validating user input, or any input, is very useful for various reasons such as: better user experience, security, avoid irrelevant issues
- Validating email addresses and domain names is useful.
- Some validation methods:
    - syntax: is the syntax of the string correct? For example, an email address must contain ‘@’. A domain name must contain ‘.’
    - Is the domain name valid?
        - is the top-level domain (tld) in use?
        - is the whole domain name in use?
    - Is the email address valid?
        - domain name part (see above)
        - local part

ICANN

Validating Domain Name

- Syntax:
  - ascii: RFC1035
  - idn:
    - using A-Labels
    - using U-Labels
- Is the top-level domain (tld) in use?
  - list of TLDs
  - with DNS requests
- Is the whole domain name in use?
  - with DNS requests

Resolving Domain Name

- After validation, a software would then use the domain name identifier as:
  - a domain name to be resolved in the DNS
- Therefore, to be UA Compliant, the software has to use proper methods that support UA.
  - for example, passing a U-Label to the traditional gethostbyname() call may not succeed, as it is not expecting a UTF8 domain name.

Validating Email Addresses

- Email address is composed of localpart@domainname
- For domain name, see before
- Syntax of local part:
  - ascii
  - utf8 (EAI)
- Can domain name receive emails?
- Is the localpart able to receive emails?

ICANN

Sending Email

- After validation, a software would then use the email identifier as:
  - an email address could be used as a user id
  - an email address could be used to send an email to
- Therefore, to be UA Compliant, the software has to use proper methods that support UA.
  - for example, passing an UTF-8 localpart email address to a mail sender routing may not succeed, as it is not expecting a UTF8 localpart in the email address.

Test Cases

- A comprehensive list of UA test cases is documented in UASG0004
- A developer is strongly encouraged to use these test cases in its unit and system testing.

Java

Java Version

- Code examples were tested against Java 11 (Oracle version) and Android API/SDK 26 where applicable
- It is possible that older versions have issues.
- Some libraries may require (not necessarily because of UA) newer versions of Java
- Unless explicitly stated, this tutorial should apply to any VM flavor: Oracle, OpenJDK or Android (Dalvik/ART)

Libraries Versions

- This tutorial shows many libraries, as found in the field. While the list is not exhaustive, it is still comprehensive to help you assess which and how to use a library, specially if the software has already been developed.
    - A future report of UASG will provide detailed information about the libraires compliance to UA. See https://uasg.tech
- Librairies shown in this tutorial have been tested on the current version available at the time of writing.
- It is very possible that new versions of some libraries have fixed issues or made enhancements that would change the recommendations.
- Please check at the time of your development the status of these libraries.

Type Holder for UA identifiers

- UA identifiers are domain names and email addresses that may contain UTF8 data.
- Java String type is well suited to hold those identifiers as it is natively supporting Unicode. Therefore, most libraries expect the String type.
- Default charset (Charset.defaultCharset()) is typically UTF-8 in most systems. Verify (`java -XshowSettings`) or change the default charset in the Java VM you are using.
    - for more info, see this JEP

Some Basic Test Cases

- The code examples throughout the tutorial will use these inputs, which provides basic UA test cases (non exhaustive):

```
List<String> testDomains = List.of(
  "example.org",                 // ascii.ascii
  "example.undefinedtld",        // unexistant tld
  "example.recentTld",           // recently allocated tld
  "example.accountants",         // allocated longer than 7 char tld
  "exâmple.org",                 // ulabel.ascii
  "xn--exmple-xta.org",          // alabel.ascii
  "exâmple.ไทย",                 // ulabel.ulabel
  "exâmple.xn--o3cw4h",          // ulabel.alabel
  "xn--exmple-xta.xn--o3cw4h"    // alabel.alabel
);
List<String> testLocalParts = List.of(
  "user",
  "kévin"
);
```

Validating Domain Name

Using Plain Java

- Traditional way of doing hostname resolution and sockets resolution
- `import java.net.InetAddress;`
- `getByName(String host); getAllByName(String host);`
- `Socket(String host, int port);`
    - uses underlying getByName()
- Throws a UnknownHostException for any failure:
    - no IP addresses returned
    - invalid host
    - bad syntax
    - ...
- Passes the host String as is to the underlying OS, without validation.
    - therefore, invalid domains (such as invalid Ulabels) are passed.
    - depends on the implementation of the underlying OS

Using Plain Java: Usage

```
import java.net.InetAddress;

try {

InetAddress[] hosts = getAllByName(input);

    } catch (UnknownHostException e) {

}
```

Using Plain Java: Usage

```
import java.net.InetAddress;

try {

InetAddress host = getByName(input);

    } catch (UnknownHostException e) {

}
```

ICANN

Using Plain Java: Usage

```
import java.net.InetAddress;

try {

Socket socket =  new Socket(input, 1234);  // 1234 = port number)

    } catch (UnknownHostException e) {

}
```

ICANN

Using Plain Java: Recommendation

- Do not use as is directly with a hostname
- Instead:
  - validate the hostname before calling getByName()
    - to avoid delays waiting for answers of queries that will throw anyway
    - provide better feedback to the user: because the throw will not tell you if the hostname was wrong or if the hostname was ok but the query did not return data.
  - prepare the hostname (e.g. convert idn to a-labels) using another lib and then use these base calls
    - this makes your code more portable across underlying OS
  - or use another lib to prepare and do the call

JRE-IDN

- included in JRE
- Implements IDNA2003.
- `import java.net.IDN;`
- String domain = IDN.toASCII(Utf8DomainString);
- throws an IllegalArgumentException
    - but just basic validation. does not really validate if a UTF-8 string is a valid label

ICANN

JRE-IDN: Usage

```
import java.net.IDN;

try {

String asciiEncodedDomain = IDN.toASCII(input);

    } catch (IllegalArgumentException e) {



}
```

JRE-IDN: Usage

```
import java.net.IDN;

try {

String unicodeEncodedDomain = IDN.toUnicode(input);

    } catch (IllegalArgumentException e) {



}
```

ICANN

JRE-IDN: Recommendation

- do not use
    - since it is based on IDNA2003

Apache Commons Validator

- Has domain and email validators
- https://github.com/apache/commons-validator
- Maven Repository:
  - ```
    <dependency>
     <groupId>commons-validator</groupId>
     <artifactId>commons-validator</artifactId>
     <version>1.6</version>
    </dependency>
    ```

ICANN
    ```

Apache Commons Validator (cont.)

- Has a static list of TLDs in code
  - the list is updated for new releases of code
    - therefore, between releases, and the day after the release, the list is outdated
    - if one deploys its software with the most current lib version, then the behavior of your validation code will be changing depending on the state of the actual list
  - so it will provide wrong results for some TLDs:
    - the TLD that were retired after the date the maintainers of the lib incorporated the content of the IANA registry of TLDs into the code as a static list
    - the TLD that were added after the date the maintainers of the lib incorporated the content of the IANA registry of TLDs into the code as a static list

Apache Commons Validator: Usage

```
import org.apache.commons.validator.routines.DomainValidator;

DomainValidator validator = DomainValidator.getInstance();

    if (validator.isValid(input)) {

}
```

Apache Commons Validator: Recommendation

- very good library, but do not use
  - since it has a static list of (always outdated) TLDs
- Not Recommended

International Components for Unicode (ICU)

- The gold standard library for Unicode. Developed by IBM. Now managed by Unicode. In sync with Unicode standards.
- Has a Java version (ICU4J): http://site.icu-project.org/home
- ICU4J is not really Java-ized. it is a direct mapping to the C version. Java developers may not like it for that reason.
- Maven Repository:
    - ```
      <dependency>
       <groupId>com.ibm.icu</groupId>
       <artifactId>icu4j</artifactId>
       <version>65.1</version>
      </dependency>
      ```

ICANN
      ```

ICU

- IDNA Conversion is based on Unicode TR46 (which supports transition from IDNA2003 to IDNA2008). However it is possible to configure for non supporting transition (recommended)
- IDNA Conversion includes normalization as per IDNA (good!)
- Do not use IDNA2003 methods (convertTo*)
- The output of the methods may contain bad domain names as disallowed characters are replaced by U+FFFD.
- Check if there are errors in the conversion by calling info.hasErrors()

ICU: Usage

```
import com.ibm.icu.text.IDNA;

IDNA validator = IDNA.getUTS46Instance(
    IDNA.NONTRANSITIONAL_TO_ASCII
  | IDNA.NONTRANSITIONAL_TO_UNICODE
  | IDNA.CHECK_BIDI
  | IDNA.CHECK_CONTEXTJ
  | IDNA.CHECK_CONTEXTO
  | IDNA.USE_STD3_RULES);
StringBuilder output = new StringBuilder();
IDNA.Info info = new IDNA.Info();
validator.nameToASCII(input, output, info);

if (info.hasErrors()) {}
```

options to not use UTS46 transitional feature and to enhance validation.

ICU: Recommendation

- for Unicode processing needs, most up to date library
- for idn domains, set the options to restrict the validation and use to IDNA2008.

Guava

- Utility library developed by Google. Has a hostname method.
- https://github.com/google/guava
- Maven Repository:

```
<dependency>
 <groupId>com.google.guava</groupId>
 <artifactId>guava</artifactId>
 <version>28.2-jre</version>
</dependency>
```

ICANN

Guava

- isValid(String domain) does very basic validation
  - a invalid Ulabel goes through ok
- Methods such as from(String domain) and is*Suffix methods are using the public suffix list.
  - therefore, may not be synchronized with the current set of deployed TLDs
  - the public suffix list version your code would be using against will be the one imported into the Guava library for the version you are using in your application.

Guava: Usage

```
import com.google.common.net.InternetDomainName;

 if (InternetDomainName.isValid(input)) {

}

InternetDomainName domain = InternetDomainName.from(input);

if (domain.isPublicSuffix()) {

}
```

Guava: Recommendation

- Not useful for validation
- If used,
  - be aware that it depends on the public suffix list, statically set into the library
  - update the lib frequently

ICANN

Xcode

- Utility library developed by Verisign. Has an "Idna" object
- https://www.verisign.com/en_US/channel-resources/domain-registry-products/idn-sdks/index.xhtml
- No Maven Repository (only a zip downloadable with a jar file in it)

Xcode: Usage

```
import com.vgrs.xcode.common.Unicode;

import com.vgrs.xcode.idna.Idna;

import com.vgrs.xcode.idna.Punycode;



 Idna idna = new Idna(new Punycode(), true, true);

 int[] output = idna.domainToUnicode(input.toCharArray()); // see domainToAscii for
roundtrip

 String domain = new String(Unicode.decode(output));
```

ICANN

Xcode: recommendation

- Slow (tests show the processing of a domain can take up to 5 seconds)
- No Maven repo
- But implements IDNA2008 perfectly

Higher in the stack

- HTTP Frameworks
- They may be using Java URL/URI

Java URL

- import java.net.URL;
- supports all protocols (e.g. not only http/https but ftp, file, …)
- does not validate hostname part.
- when invalid, throws MalformedURLException

Java URL: Usage

```
import java.net.URL;

  try {

    URL url = new URL("http://" + input);

  } catch (MalformedURLException e) {

  }
```

Java URI

- identical to URL
- when invalid, throws URISyntaxException

Java URI: Usage

```
import java.net.URI;

    try {

      URI uri = new URI("http://" + input);

    } catch (URISyntaxException e) {

    }
```

Java URI/URL Recommendation

- ok to use but does not validate hostname
- use another library to validate hostname

Making an HTTP Request

Java 1.1 HttpURLConnection

- old way
- uses java.net.URI
  - therefore inherit its characteristics
- As much superior HTTP libraries/packages/frameworks exist nowadays, should consider using another one.

Apache HTTPClient

- old way
- No validation of domain
- Maven:

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.10</version>
</dependency>
```

ICANN

Apache HTTPClient: Usage

import org.apache.http.client.methods.HttpGet;

Example:

```
try {
   HttpGet request = new HttpGet("http://" + input);
 } catch (IllegalArgumentException e) {
 }
```

ICANN

Apache HTTPClient: Recommendation

- As much superior HTTP libraries/packages/frameworks exist nowadays, should consider using another one.

OkHTTP

- More recent, kept up to date (supports http/2), active, uses Builder() constructs and more popular than previous ones. Created by Square.
- Was written in Java, but then moved to Kotlin (still compatible with Java)
- https://square.github.io/okhttp/
- Maven:

```
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
    <artifactId>okhttp</artifactId>
    <version>4.2.2</version>
</dependency>
```

OkHTTP

- Provides some method to use public suffix list (but not used by default)
- Does not validate host part in URL
- Encodes automatically idn Ulabels by calling java.net.IDN
    - therefore inheriting characteristics of java.net.IDN

OkHTTP: Usage

```
import okhttp3.Response;

OkHttpClient httpClient = new OkHttpClient();
Request request = new Request.Builder().url("http://" + input).build();
try {
  Response response = httpClient.newCall(request).execute();
} catch (IOException e) {
}
```

OkHTTP: Usage

```
import okhttp3.HttpUrl;

HttpUrl url = HttpUrl.parse(“http://” + input);

if (url == null) { }
```

ICANN

OkHTTP: Recommendation

- Better not use public suffix list methods
- validate and prepare hostnames prior to use OkHTTP
- use a IDNA2008 library to convert to A-Labels so that OkHTTP will not try to convert using java.net.IDN which is IDNA2003.

Java 11 HTTP Client

- new HTTP Client bundled in Java 11 with modern constructs (Builder,...)
- Does not support any UTF8 as hostname. Throws an IllegalArgumentException
- uses java.net.URI
  - therefore inheriting characteristics of java.net.URL/URI
- Does not check if an idn is valid (i.e. invalid punycode is not checked)
- Assumes classic hostname (ascii, RFC1035).

Java 11 HTTP Client: Usage

```
import java.net.http.HttpRequest;

try {
  HttpRequest request = HttpRequest.newBuilder()
    .GET()
    .uri(URI.create("http://" + input))
    .build();
} catch (IllegalArgumentException e) {


}
```

Java 11 HTTP Client: Recommendation

- As included in Java, no need for additional libs or packages, therefore no dependency and versioning management
- However, as it does not prepare UTF8 idn nor validate, then add a preparation/validation step of hostname before calling the HTTP client.

Google Java HTTP Client

- Google Library
- has a GenericURI() class for handling URI
    - which uses java.net.URL/URI underneath
        - therefore inheriting characteristics of java.net.URL/URI
- https://github.com/googleapis/google-http-java-client
- Maven:

```
<dependency>
  <groupId>com.google.http-client</groupId>
  <artifactId>google-http-client</artifactId>
</dependency>
```

Google Java HTTP Client: Usage

```
import com.google.api.client.http.GenericUrl;

try {
  HttpTransport HTTP_TRANSPORT = new NetHttpTransport();
  GenericUrl url = new GenericUrl("http://" + input);
  if (url.host = “”) { }
  HttpRequest request = HTTP_TRANSPORT
    .createRequestFactory()
    .buildGetRequest(url);
  HttpResponse response = request.execute();
} catch (IllegalArgumentException e) {}
```

Google Java HTTP Client: Recommendation

- As it relies on java.net.URI, it does not validate nor prepare hostnames.
- prepare and validate hostnames using another library before calling GenericURL()

ICANN

Validating Email Addresses

Email Regular Expressions (Regex)

- Basic: something@something
  - ^(.+)@(.+)$
- From owasp.org (security):
  - [^[a-zA-Z0-9_+&*-]+(?:\.[a-zA-Z0-9_+&*-]+)*@(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,7}$]
  - does not support EAI (i.e. no UTF-8 local parts allowed: [a-zA-Z0-9_+&*-])
  - does not support ascii TLD longer than 7 characters: [a-zA-Z]{2,7}
  - does not support ULabels in idn TLD: [a-zA-Z]
  - But OWASP is _the_ reference for security.
    - Therefore you may end up fighting with your security team to use a UA compatible Regex instead of the “standard” one from OWASP.

Email Regular Expressions (Regex) (cont.)

- Example of Regex suggested in various forums: ex: List of proposals
  - ^[A-Za-z0-9+_.-]+@(.+)$
  - ^[a-zA-Z0-9_!#$%&'*+/=?`{|}~^.-]+@[a-zA-Z0-9.-]+$
  - ^[a-zA-Z0-9_!#$%&'*+/=?`{|}~^-]+(?:\\.[a-zA-Z0-9_!#$%&'*+/=?`{|}~^-]+)*@[a-zA-Z0-9-]+(?:\\.[a-zA-Z0-9-]+)*$
  - ^[\\w!#$%&'*+/=?`{|}~^-]+(?:\\.[\\w!#$%&'*+/=?`{|}~^-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}$
  - All are not EAI compliant as they either:
    - do not support UTF8 on the local part
    - have length restrictions for the tld
    - do not support ULabels

Email Regular Expressions (Regex) (cont.)

- One may come up with an EAI-IDN compatible regex using various Unicode codepoints characteristics, but just for IDN, it would start looking like a reimplementation of the IDNA protocol tables in regex!
- Moreover, given that both sides of an EAI may have UTF8, then one regex for an EAI could be .*@.* which is only verifying the presence of the ‘@’ char.

Jakarta Mail

- Most used Java package to send email
- Has also a validate() method to validate email addresses
- import javax.mail
- Maven:

```
<dependency>
  <groupId>com.sun.mail</groupId>
  <artifactId>jakarta.mail</artifactId>
  <version>1.6.5</version>
</dependency>
```

ICANN

Jakarta Mail

- validate() does a good job in validating email addresses, specially the local part.
  - It verifies illegal characters such as: ()<>,;:"[]\ , whitespaces, etc...
  - it verifies the characters are only digit and letters per the definition of Unicode classes.
  - it does not validate idn

Jakarta Mail: Usage

```
import javax.mail.internet.InternetAddress;
try {
 InternetAddress emailAddr = new InternetAddress(input);
 emailAddr.validate();
} catch (AddressException e) {}
```

Jakarta Mail: Recommendation

- Good library to use.
- Add (idn) domain validation and preparation as an additional step
- Do not use the old Java Mail package (com.sun.mail:javax.mail), since there were multiple UTF-8 bugfixes when Java Mail became Jakarta Mail

Apache Commons Validator

- Has domain and email validators
- Has a static list of TLDs!!! OUTDATED! (always…)
- https://github.com/apache/commons-validator
- Maven Repository:
  - ```
    <dependency>
     <groupId>commons-validator</groupId>
     <artifactId>commons-validator</artifactId>
     <version>1.6</version>
    </dependency>
    ```
    ```

Apache Commons Validator

```
EmailValidator validator = EmailValidator.getInstance();
 boolean emailValid = validator.isValid(input);
                if (emailValid) {
```

Apache Commons Validator: Recommendation

- do not use as it relies on a static list of TLDs

ICANN

EmailValidator4J

- https://github.com/egulias/EmailValidator4J
- claims to support EAI!
- State of development and maintenance is unknown.
- To watch

ICANN

Sending Email

JakartaMail

- same lib as before. see above
- Maven:

```
<dependency>
  <groupId>com.sun.mail</groupId>
  <artifactId>jakarta.mail</artifactId>
  <version>1.6.5</version>
</dependency>
```

JakartaMail: Usage for Sending Email

```
Properties properties = System.getProperties();
properties.setProperty("mail.smtp.host", host);
Session session = Session.getDefaultInstance(properties);
try {
  MimeMessage message = new MimeMessage(session);
  message.setFrom(new InternetAddress(from));
  message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
  message.setSubject("This is the Subject Line!");
  message.setText("This is actual message");
  Transport.send(message);
} catch (MessagingException e) {
}
```

JakartaMail: Recommendation

- Good library to use.
- Supports EAI (properly since 1.6.5)
- See discussion before for validation

Simple Java Mail

- Jakarta Mail wrapper. simpler calls to send emails.
- Handles a lot of modern features
- More modern constructs (Builder)
- https://github.com/bbottema/simple-java-mail/
- http://www.simplejavamail.org
- Maven:

```
<dependency>
  <groupId>org.simplejavamail</groupId>
  <artifactId>simple-java-mail</artifactId>
  <version>6.0.5</version>
</dependency>
```

ICANN

Simple Java Mail Validation

- uses (and includes) another library for email validation
  - https://github.com/bbottema/email-rfc2822-validator.git
- Maven:

```
<dependency>
  <groupId>com.github.bbottema</groupId>
  <artifactId>emailaddress-rfc2822</artifactId>
  <version>2.1.4</version>
</dependency>
```

Simple Java Mail Validation (cont.)

- Uses various Regex
- considers any UTF8 as invalid, therefore no Ulabels in domains, no EAI

Simple Java Mail: Usage

```
Mailer mailer = MailerBuilder
  .withSMTPServer("smtp.host.com”)
  .async();
Email email = EmailBuilder.startingBlank()
  .to("user@example.org")
  .buildEmail();
mailer.sendMail(email);
```

Simple Java Mail: Recommendation

- While it provides a modern interface for sending Email, it does not support EAI nor U-labels for domains.
- Internal validation based on the obsolete RFC2822

Frameworks

SpringBoot Framework

- very well used in Java, server side
- https://spring.io
- has @Email annotations, http requests, send email

SpringBoot HTTP Request

- internally uses java.net.URI
    - therefore inherit characteristics from java.net.URI
- Maven:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.2.4.RELEASE</version>
</dependency>
```

ICANN

SpringBoot HTTP Request

- Does not validate domains
- Converts UTF8 domain labels into percent encoded labels -> WRONG

SpringBoot HTTP Request: Usage

```
import org.springframework.web.client.RestTemplate
RestTemplate restTemplate = new RestTemplate();
try {
  String result = restTemplate.getForObject("http://" + input, String.class);
} catch (RestClientException e) {}
```

SpringBoot HTTP Request: Recommendation

- Validate and prepare hostnames before using this library

SpringBoot Send Email

- Wrapper using Java Mail.
    - therefore inherit characteristics of Java Mail
- Does not validate
- https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/mail.html
- Maven:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-mail</artifactId>
  <version>2.2.4.RELEASE</version>
</dependency>
```

SpringBoot Send Email: Usage

```
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.JavaMailSenderImpl;

SimpleMailMessage msg = new SimpleMailMessage();
msg.setTo(emailAddress);
msg.setText(emailtext);
try{
  mailSender.send(msg);
} catch(MailException ex) {}
```

SpringBoot Send Email: Recommendation

- Validate and prepare email addresses before using SpringBoot Mail
- Verify dependencies versions to use the right Jakarta Mail conforming to EAI.

Database Considerations

Database Considerations

- SQL
  - Domain names: max: 255 octets, 63 octets per label. However, in UTF-8 native, variable length.
  - Recommendation use variable length String columns
  - Consider/Verify the Object-relational mapping (ORM) driver/tool if you are using one.

- noSQL
  - Already UTF8 variable length

Android

android.icu.text.IDNA

- Same ICU library, integrated into android OS
  - no need to add packages dependencies
- https://developer.android.com/reference/android/icu/text/IDNA
- Same considerations as discussed before for the icu4j library

Best Practices

Best Practices

- Validation for EAI, IDN, UA input is complicated.
- Never rely on a static list of TLDs. They come and go. More to come.
- Do not code any specific syntax or length other than what is in the standards. For example, a label, such as a TLD, maybe of length up to 63 octets in A-Label/ascii format which includes the ‘xn--’ prefix for an IDN,
  - so coding that a tld has a maximum of 6 or 7 octets is just wrong.
- Use String type to hold domain name and email addresses
- Make sure to do normalization of input before storing, comparing, processing.
- If storing identifiers in database, make sure the whole code path up and including the database itself is UA compliant. For example, UTF8 should be the charset of the column storing the identifier (domain or email) in SQL databases.

Best Practices (cont.)

- Prefer to do some basic validation and then make requests and trap the errors, than trying to do too much validation.
- Use a UA-conformant library/framework (IDNA2008 for domains)
- Do unit and system testing of UA identifiers
  - Consider using UASG0004 as a good starter for test cases.
- Consider converting domain names to their A-Label equivalent when passing to libraries, so it is more safe to go through the whole code path (that may include libraries that are dependencies and are not UA conformant)

Best Practices (cont.)

- If using HTTP Requests, most frameworks depend on underlying Java URL/URI which do not validate and do not prepare hostnames.
  - Prepare hostnames using an IDNA2008 library and then pass the result to the HTTP framework.

Domain Name Validation and Resolution

- Convert U-labels into A-Labels and then pass to the standard methods
- Think of U-labels as for display.
- There is a one-to-one direct lossless conversion between A-Labels and U-Labels, therefore no need to keep both. Keep A-Labels as they are more supported everywhere in the code and dependencies
- However, U-Labels are useful for sorting, comparing and searching, as they will be sorted based on the real value: i.e. the UTF8 string, instead of its punycode encoding.
- When about to display a String, always convert to U-Labels, as the end-user expects the U-Labels, because the libraries will do nothing if the domain is in A-label format.

Email Validation and Sending

- Do normalization of the UTF8 local parts if the email is received as input
- Always use normalized local parts when comparing, sorting or searching
- for domain part, see domain section
- Validate using the right lib before sending
- if sending an email to an EAI address, prepare that:
    - the email could be refused by your mailer
    - the email may not reach its destination, if one of the mail servers in the path does not support EAI.

Back to the Company: preparing an email address

- We now know the company dev team did part of the job. Some emails (like customer@ஸ்ரீ.co) will still be rejected by Jakarta Mail because of the domain part not prepared. Here is the complete example

It prepares email address with A-labels in domain, which is then used as input to any libraries/package /frameworks. However, the local part remains UTF8 which may not be working in the mail delivery path.

```java
IDNA validator = IDNA.getUTS46Instance(
    IDNA.NONTRANSITIONAL_TO_ASCII
    | IDNA.NONTRANSITIONAL_TO_UNICODE
    | IDNA.CHECK_BIDI
    | IDNA.CHECK_CONTEXTJ
    | IDNA.CHECK_CONTEXTO
    | IDNA.USE_STD3_RULES);
IDNA.Info info = new IDNA.Info();
String localpart = email.substring(0, email.lastIndexOf("@"));
String domain = email.substring(email.lastIndexOf("@") + 1);
StringBuilder output = new StringBuilder();
validator.nameToASCII(domain, output, info);
email = localpart + "@" + output.toString();

if (isEmailValid(email)) {
  subscribe();
} else {
  throw new Exception("Invalid email address, please review it and submit again");
}
```

Conclusion

- Be aware that UA identifiers may not be fully supported in software and libraries
- Use the right libraries and frameworks
- Adapt your code to properly support UA
- Do unit and system testing using UA test cases to ensure that your software is UA ready
- Code examples of this tutorial available at https://github.com/icann/ua-java-tutorial

UA References

- http://uasg.tech
- Use Cases for UA Readiness Evaluation, UASG-0004
- Reviewing Programming Languages and Frameworks for Compliance with Universal Acceptance Good Practice, UASG-018
- Evaluation of Software libraries for UA Readiness: http://uasg.tech/software
- Universal Acceptance Readiness Framework, UASG-026

IDN References

- Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.
- Klensin, J., "Internationalized Domain Names in Applications (IDNA): Protocol", RFC 5891, DOI 10.17487/RFC5891, August 2010, <https://www.rfc-editor.org/info/rfc5891>.
- Faltstrom, P., Ed., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)", RFC 5892, DOI 10.17487/RFC5892, August 2010, <https://www.rfc-editor.org/info/rfc5892>.
- Alvestrand, H., Ed., and C. Karp, "Right-to-Left Scripts for Internationalized Domain Names for Applications (IDNA)", RFC 5893, DOI 10.17487/RFC5893, August 2010, <https://www.rfc-editor.org/info/rfc5893>.
- Klensin, J., "Internationalized Domain Names for Applications (IDNA): Background, Explanation, and Rationale", RFC 5894, DOI 10.17487/RFC5894, August 2010, <https://www.rfc-editor.org/info/rfc5894>.
- Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, DOI 10.17487/RFC3492, March 2003, <https://www.rfc-editor.org/info/rfc3492>.

EAI References

- Klensin, J. and Y. Ko, "Overview and Framework for Internationalized Email", RFC 6530, DOI 10.17487/RFC6530, February 2012, <https://www.rfc-editor.org/info/rfc6530>.
- Yao, J. and W. Mao, "SMTP Extension for Internationalized Email", RFC 6531, DOI 10.17487/RFC6531, February 2012, <https://www.rfc-editor.org/info/rfc6531>.
- Yang, A., Steele, S., and N. Freed, "Internationalized Email Headers", RFC 6532, DOI 10.17487/RFC6532, February 2012, <https://www.rfc-editor.org/info/rfc6532>.
- Hansen, T., Ed., Newman, C., and A. Melnikov, "Internationalized Delivery Status and Disposition Notifications", RFC 6533, DOI 10.17487/RFC6533, February 2012, <https://www.rfc-editor.org/info/rfc6533>.
- Levine, J. and R. Gellens, "Mailing Lists and Non-ASCII Addresses", RFC 6783, DOI 10.17487/RFC6783, November 2012, <https://www.rfc-editor.org/info/rfc6783>.
- Resnick, P., Ed., Newman, C., Ed., and S. Shen, Ed., "IMAP Support for UTF-8", RFC 6855, DOI 10.17487/RFC6855, March 2013, <https://www.rfc-editor.org/info/rfc6855>.
- Gellens, R., Newman, C., Yao, J., and K. Fujiwara, "Post Office Protocol Version 3 (POP3) Support for UTF-8", RFC 6856, DOI 10.17487/RFC6856, March 2013, <https://www.rfc-editor.org/info/rfc6856>.
- Fujiwara, K., "Post-Delivery Message Downgrading for Internationalized Email Messages", RFC 6857, DOI 10.17487/RFC6857, March 2013, <https://www.rfc-editor.org/info/rfc6857>.
- Gulbrandsen, A., "Simplified POP and IMAP Downgrading for Internationalized Email", RFC 6858, DOI 10.17487/RFC6858, March 2013, <https://www.rfc-editor.org/info/rfc6858>.