Showcasing UA Readiness and Conducting Bug Reporting with Programming Languages

Cofomo inc

2022-06

Introduction

This document describes code snippets for different languages and libraries, extracted from code samples. written to achieve Universal Acceptance readiness.

Depending on the libraries, the samples illustrate compliance with Internationalized Domain Names in Application (IDNA 2008) or Email Address Internationalization (EAI).

The following languages and libraries are nominally used in the code samples:

Language	Framework/Library	Version
Java	commons-validator	1.7
	guava	31.0.1-jre
	icu	70.1
	jakartamail	2.0.1
Python	idna	3.3
	email-validator	1.1.3
	smtplib	Python 3.10

Javascript	idna-uts46	1.1.0
	validator	13.7.0
	nodemailer	6.7.2

Code snippets description

Note

Depending on the library and languages, compliance cannot always be achieved. The snippet will explain the workaround and limitation encountered when writing UA ready code samples.

Java

Commons Validator

Snippets

Commons validator provides many validators. For our code samples we focused on domain validator, email validator and URL validator.

Commons validator uses a static list of TLDs that is shipped with the sources, therefore, depending on the version of the library the TLD list can be very obsolete. Anyway, as the list of TLD is regularly updated, any static list will be quickly obsolete.

To mitigate this, the following snippet downloads the list of TLDs from ICANN website:

```
private static final String IANA_TLD_LIST_URL = "https://data.iana.org/TLD/tlds-alpha-by-domain.txt";

public static String[] retrieveTlds() {
 StringBuilder out = new StringBuilder();
 try (BufferedInputStream in = new BufferedInputStream(
    new URL(IANA_TLD_LIST_URL).openStream())) {
  byte[] dataBuffer = new byte[1024];
  int bytesRead;
  while ((bytesRead = in.read(dataBuffer, 0, 1024)) != -1) {
   out.append(new String(dataBuffer, 0, bytesRead));
  }
 } catch (IOException e) {
  // handle exception
 }
 return Arrays.stream(out.toString().split("\n"))
    .filter(s -> !s.startsWith("#"))
    .map(String::toLowerCase).distinct().toArray(String[]::new);
}
```

This list can then be provided when creating a domain validator instance with:

```
List<Item> domains = new ArrayList<>();
domains.add(new Item(GENERIC_PLUS, retrieveTlds()));
DomainValidator validator = DomainValidator.getInstance(false, domains);
```

It is also possible to avoid managing the full list of TLD and completely bypass the TLD existing check by providing the TLD of the domain to be tested directly to the domain validator with:

```
public static DomainValidator createDomainValidatorInstance(String domain) {
 String tld = domain;
 if (domain.contains(".")) {
  tld = domain.substring(domain.lastIndexOf(".") + 1);
 }

 // Convert TLD to A-Label
 int flags = IDNA.CHECK_BIDI
    | IDNA.CHECK_CONTEXTJ
    | IDNA.CHECK_CONTEXTO
    | IDNA.NONTRANSITIONAL_TO_ASCII
    | IDNA.USE_STD3_RULES;
 IDNA idna = IDNA.getUTS46Instance(flags);
 IDNA.Info info = new IDNA.Info();
 StringBuilder tldAscii = new StringBuilder();
 idna.nameToASCII(tld, tldAscii, info);
 // if there is an error, do nothing, validator will fail
 if (!info.hasErrors()) {
  return DomainValidator.getInstance(false,
     List.of(new Item(GENERIC_PLUS, new String[]{tldAscii.toString()}));
 }
 return DomainValidator.getInstance();
}
```

The domain validator is also used by other validators, such as email validator, therefore, it should also be instantiated with an accurate list of TLDs when creating other validators instance.

Then, the domain validator can be used for validation:

```
public static boolean validate(String domain) {
 DomainValidator validator = DomainValidator.createDomainValidatorInstance(domain);
 return validator.isValid(domain);
}
```

In case a full URL needs to be validated, one can also use the URL validator:

```
public static boolean validateUrl(String url) {
  String domain;
  try {
   domain = new URL(url).getHost();
  } catch (MalformedURLException e) {
   return false;
  }
  UrlValidator urlValidator = new UrlValidator(null, null, 0L,
     createDomainValidatorInstance(domain));
  return urlValidator.isValid(url);
}
```

Similarly, email validation can be performed as follows:

```
public static boolean validateAddress(String email) {
  String domain = email.substring(email.lastIndexOf("@") + 1);
  EmailValidator validator = new EmailValidator(false, false,
     createDomainValidatorInstance(domain));
  return validator.isValid(email);
}
```

Notes

The tested version includes a way to create instances of domain validator with a list of TLDs to consider valid which was not the case before. Former versions only allowed to override the list before the domain validator instance creation, which would therefore make the list become obsolete after a certain amount of time in long running applications.

The validation examples do not check if the TLD exist but with the *retrieveTlds* method above one can easily provide an accurate list of TLDs to the validator. If the validator is used repeatedly, the TLDs list should be cached and updated when necessary.

Bug reports

Domain validator converts the domain to A-label with *java.net.IDN* that is only IDNA 2003 compliant:

- https://issues.apache.org/jira/browse/VALIDATOR-483

Guava

Snippets

Guava only performs some basic check (label length, no ending dash, …) on domain names and allows checking whether the domain is in the public suffix list. Therefore, to perform a real UA compliant domain validation, another validation method should be used, making Guava obsolete except for the public suffix list check.

To check if a domain is in the public suffix list with Guava you can do as follows:

```
public static boolean isInPublicSuffixList String domain) {
  try {
    InternetDomainName internetDomainName = InternetDomainName from(domain);
    return internetDomainName.hasPublicSuffix();
  } catch (IllegalArgumentException e) {
    // the domain is invalid, but a real UA compliant validation should have already been made before
  }
}
```

Notes

Guava does not bring benefits for UA ready domain name validation. If one wants to use the public suffix list it can be of interest, but this list may not always be up to date with the latest TLD and should only be used for extremely specific use cases.

Bug reports

We intended to report a bug for IDNA 2008 support but when searching for existing bug reports we noticed that Guava maintainers do not want to depend on third-party libraries, and they recommend doing the conversion with such library before using it in Guava. See https://github.com/google/guava/issues/5808#issuecomment-987099900.

We then submitted a pull request to improve their documentation on IDN that is misleading:

- https://github.com/google/guava/pull/5929

ICU

Snippets

ICU can be used to convert domain to A-label prior to using it on wire. This would mitigate the risk induced by other libraries which may have a poor UA compliance level.

To be fully compliant with IDNA 2008, some flags must be provided when creating the instance:

```
private static final IDNA idnaInstance = IDNA.getUTS46Instance(IDNA.NONTRANSITIONAL_TO_ASCII
    | IDNA.CHECK_BIDI
    | IDNA.CHECK_CONTEXTJ
    | IDNA.CHECK_CONTEXTO
    | IDNA.USE_STD3_RULES);
```

Then, the instance can be used to convert a domain name to A-label:

```
public static Optional<String> toALabel(String domain) {
 StringBuilder output = new StringBuilder();
 IDNA.Info info = new IDNA.Info();

 idnaInstance.nameToASCII(domain, output, info);

 String domainALabel = output.toString();
 if (!info.hasErrors()) {
  return Optional.of(domainALabel);
 } else {
  // you can retrieve the errors with info.getErrors()
  return Optional.empty();
 }
}
```

Notes

As ICU performs validation as well as conversion from/to A-label, it should also be used for domain name validation.

Bug reports

ICU performs some normalization on domain and therefore, some invalid characters as per IDNA 2008 but valid as per UTS46 are considered valid:

- https://unicode-org.atlassian.net/browse/ICU-21922

Jakarta mail

Snippets

Jakarta mail offers compliance with EAI but some valid Unicode characters as per IDNA 2008 are rejected. Moreover, the domain part needs to be NFC normalized before being provided to it. As much software may need to store and query those email addresses, it is a good practice to perform normalization of the whole address prior to any other use.

To normalize the domain part, the following can be done:

```
String localPart = compliantTo.substring(0, emailAddress.lastIndexOf("@") - 1);
String domain = compliantTo.substring(emailAddress .lastIndexOf("@") + 1);
// Email address with domain normalized
emailAddress = localPart + "@" + Normalizer.normalize(domain, Normalizer.Form.NFC);
```

Then, to mitigate the risk of rejecting valid email addresses, when Jakarta mail considers an address as invalid, another library can be used to check whether the domain is valid and if so, the domain should be converted to A-label to avoid being rejected by Jakarta mail:

```
try {
  InternetAddress internetAddress = new InternetAddress(emailAddress, false);
  internetAddress.validate();
} catch (Exception e) {
  // Email address is still rejected, convert domain to A-label
  emailAddress = convertDomainToAlabel(compliantTo).orElseThrow(() -> e);
}
```

ICU that is fully compliant with UA can do both steps, thus, the conversion to A-label method can be for instance:

```
private static Optional<String> convertDomainToAlabel(String emailAddress) {
  String localPart = emailAddress.substring(0, emailAddress.lastIndexOf("@") - 1);
  String domain = emailAddress.substring(emailAddress.lastIndexOf("@") + 1);

  IDNA idnaInstance = IDNA.getUTS46Instance(IDNA.NONTRANSITIONAL_TO_ASCII
      | IDNA.CHECK_BIDI
      | IDNA.CHECK_CONTEXTJ
      | IDNA.CHECK_CONTEXTO
      | IDNA.USE_STD3_RULES);

  StringBuilder output = new StringBuilder();
  IDNA.Info info = new IDNA.Info();

  idnaInstance.nameToASCII(domain, output, info);

  String domainALabel = output.toString();
  if (!info.hasErrors()) {
    String converted = localPart + "@" + domainALabel;
    return Optional.of(converted);
  } else {
    return Optional.empty();
```

```
  }
}
```

Note: Conversion to A-label is not always performed independently of Jakarta mail validation result as it allows keeping the email address the closest possible to the user input.

After those steps, the email address is considered valid by Jakarta mail (providing that the email address is indeed valid). Then, we can configure and create a mail session with:

```
Properties props = System.getProperties();

props.put("mail.smtp.host", host);
props.put("mail.smtp.port", port);
// add any other configuration for the SMTP server

// enable UTF-8 support, mandatory for EAI support
props.put("mail.mime.allowutf8", true);

Session session = Session.getInstance(props, null);
```

The configuration flag *allowutf8* is particularly important here as it will enable EAI support in Jakarta mail.

Before sending the message, the SMTP server should be queried to ensure its support for the SMTPUTF8 option. Indeed, Jakarta mail detects when the server does not support the option and displays a warning in the logs but still sends the message. When the server does not support the options, downgrading measures should be performed.

The snippet below describes how to query the server to check for the SMTPUTF8 compliance and if not supported, converts the domain part of the email address to A-label if the local-part is ASCII only, else sends an error:

```
try (Transport transport = session.getTransport()) {
  transport.connect(host, port, null, null);  // add credentials if needed

  if (transport instanceof SMTPTransport &&
      !((SMTPTransport) transport).supportsExtension("SMTPUTF8")) {
    // server does not support SMTPUTF8 extension, convert domain to A-label as a downgrading
    // measure
    // first check the local-part is ASCII-only
    String localPart = compliantTo.substring(0, compliantTo.lastIndexOf("@") - 1);
    if (!CharMatcher.ascii().matchesAllOf(localPart)) {
      // local-part contains non-ASCII characters
      return;  // stop here as email will likely be rejected if kept as-is
    }
    // domain conversion should not fail as domain is already validated
    emailAddress = convertDomainToAlabel(emailAddress).orElseThrow(/* should not happen */);
  }
} catch (MessagingException e) {
```

```
 // failed to query the server
}
```

Now the email address has been validated and the server will be able to handle it, the message can be built and sent. The message is built with some proper header, to correctly handle character encoding for internationalized email:

```
try {
 MimeMessage msg = new MimeMessage(session);
 //set message headers for internationalized content
 msg.addHeader("Content-type", "text/HTML; charset=UTF-8");
 msg.addHeader("Content-Transfer-Encoding", "8bit");
 msg.addHeader("format", "flowed");

 msg.setFrom(new InternetAddress(sender));
 msg.setSubject(subject, "UTF-8");
 msg.setText(content, "UTF-8");
 msg.setSentDate(new Date());
 msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse(emailAddress, false));
 Transport.send(msg);
} catch (MessagingException e) {
 // failed to send the email
}
```

Bug reports

A bug report is already opened for valid domains rejected by Jakarta mail:

- https://github.com/eclipse-ee4j/mail/issues/589

Python

IDNA

Snippets

IDNA is a Python library implementing IDNA 2008. Therefore, it can be used to convert domain to A-label before using other libraries to make HTTP, DNS or other requests related to the domain name for instance.

IDNA does not perform any normalization before converting the domain, for IDNA 2008 the domain should be normalized in NFC form (RFC 5891 § 4.1), using Python Unicode module, as below:

```
domain_normalized = unicodedata.normalize('NFC', domain)
```

If this conversion is not performed, IDNA will reject it.

Then the domain can be manipulated with the IDNA library. The snippet below converts the domain to A-label, when the domain is not valid, the *idna.IDNAError* exception should be catch:

```
try:
   domain_a_label = idna.encode(domain_normalized).decode('ascii')
   logger.info(f"Domain '{domain}' converted in A-Label is '{domain_a_label}'")
```

```
    return domain_a_label
except idna.IDNAError as e:
    # The label is invalid as per IDNA 2008
    logger.error(f"Domain '{domain}' is invalid: {e}")
```

The snippet above describes one way to convert a domain name to A-label, the library offers other ways that are documented in its documentation.

Notes

The Python IDNA package is widely used and fully IDNA 2008 compliant. There is an ongoing discussion to make it a replacement to the Python idn module that implements IDNA 2003.

It can also be used to validate domain names.

Bug reports

No bug has been reported.

Email validator

Snippets

Email validator performs common email addresses validation. That means that some valid email addresses per RFCs may be rejected, but only for edge cases.

EAI, however, is correctly supported and internationalized domain names validation is performed with the IDNA library.

The snippet below describes the way the library can be used to validate an email address. If the address is invalid, an exception is raised and should be caught.

```
try:
    validate_email(address, check_deliverability=False)
    logger.info(f"'{address}' is a valid email address")
    return True
except EmailNotValidError as e:
    logger.error(f"'{address}' is not a valid email address: {e}")
    return False
```

Notes

validate_email method contains an option to make DNS query to ensure that the email address domain exists. Depending on the volume of email addresses to validate it can be interesting to enable it. The method also returns an object with accessors on some email address properties such as local-part, domain, a normalized form, and the email with domain converted to A-label which can be especially useful when the email address needs to be used afterwards.

As stated above, the email validation is not fully compliant with RFCs. For instance, it does not accept quoted local part which should be permitted. RFC 5321 recommends against using quoted string to ensure email deliverance, so the choice of the developers respects the RFC recommendations which makes it acceptable.

smtplib

Snippets

smtplib can be used to send EAI compliant emails. It makes a partial validation of the email address (e.g. it does not validate the domain compliance with IDNA 2008) therefore another validation method should be used before trying to send an email, for instance using the email-validator library.

smtplib should be correctly used to set the relevant options for internationalized emails. Multiple methods exist to send emails; *sendmail* would need the charset and relevant options to be set manually while *send_message* would do this automatically, therefore *send_message* is used in the following snippet.

```
try:
   # create the smtp instance
   smtp = smtplib.SMTP(host, port)
   # set debug level to true if you want to see all messages sent to and received from
   # the server
   smtp.set_debuglevel(False)
   # build the email message
   msg = EmailMessage()
   msg.set_content(content)
   msg['Subject'] = subject
   msg['From'] = sender

   # send the email
   smtp.send_message(msg, sender, to)
   smtp.quit()
   logger.info(f"Email sent to '{to}'")
except smtplib.SMTPNotSupportedError:
   # The server does not support the SMTPUTF8 option, you may want to perform downgrading
   logger.warning(f"The SMTP server {host}:{port} does not support the SMTPUTF8 option")
   raise
```

Exceptions should be correctly caught to check for any error when sending the message. In the snippet, we catch the *smtplib.SMTPNotSupportedError* as it is raised when the server does not support the SMTPUTF8 option.

In such cases, downgrading can be performed. An example of downgrading is the conversion of domain to A-label in case local-part is ASCII only. If using email validator, the library directly provides the email address with domain converted to A-label, else the snippet below can be used:

```
def email_to_ascii(email):
   local_part, domain = email.rsplit('@', 1)
   if not local_part.isascii():
      logger.error(f"Email local part '{email}' contains Unicode characters, "
                   f"cannot transform it to full ASCII")
      return

   normalized = unicodedata.normalize('NFC', domain)
    try:
       converted = idna.encode(normalized).decode('ascii')
    except idna.IDNAError as e:
```

```
        logger.error(f"Email domain '{domain}' is not valid against IDNA 2008: {e}")
        return

    return '@'.join((local_part, converted))
```

Notes

From the Python documentation, smtplib module supports SMTPUTF8 flag since Python 3.5.

JavaScript

Idna-uts46

Snippets

Idna-uts46 accepts a few options depending on which protocol (IDNA2003, IDNA2008) one wants and how strict he/she wants to be. Idna-uts46 does only domain conversion and does not support URL.

Server-side (NodeJS)

If we are server-side (JavaScript with the NodeJS engine), we can use the URL native object from NodeJS to extract the domain since it is IDNA2008 compliant:

```
function idna_convert(website, results) {
  let url = undefined;
  try {
    url = new URL(website);
  } catch (err) {
    return results.addResult(
      true,
      `[nodejs] parsing URL ${website} failed: ${err?.message}`,
      website);
  }

  // if node automatically converted the url (when compiled with ICU), convert it back
  // since it doesn't use the proper ICU flags,
  // see: https://github.com/nodejs/node/blob/97826d3d8321f4dd86f2973fa6838101d14e8082/src/node_i18n.cc#L612
  // where USE_STD3_RULES is missing in default mode
  const domain = domainToUnicode(url.hostname);
  url.hostname = domain;
  results.addResult(
    false,
    `[nodejs] successfully parsed the domain ${domain} from URL ${website}`,
    url.toString());
  const domain_normalized = domain.normalize('NFC');
  url.hostname = domain_normalized;
  if (!website.includes(domain_normalized)) {
    results.addResult(false, `[nodejs] Website domain has been normalized in NFC form: ${domain_normalized}`,
url.toString())
  }
  try {
    const domain_ascii = idna.toAscii(domain_normalized, {transitional: false, useStd3ASCII: true, verifyDnsLength: true});
    url.hostname = domain_ascii;
    if (domain_ascii !== domain_normalized) {
      results.addResult(
```

```
            false,
            `[idna-uts46] The website domain name has been transformed to A-Label to ensure IDNA 2008 compliant queries:
<a href=${domain_ascii}>${domain_ascii}</a>`,
            url.toString());
      }
      return results.addResult(false, `[idna-uts46] ${website} is a valid URL`, url.toString());
   } catch (err) {
      return results.addResult(
         true,
         `[idna-uts46] URL ${website} is invalid: ${err?.message}`,
         website);
   }
}
```

Unfortunately, NodeJS URL does not use STD3 Rule and thus accept many illegal ASCII characters in the LDH (letters, digits, hyphens) domain like "+" or "_" for instances. In addition to that, IDNA2008 “DISALLOWED” characters are completely ignored. If it were not the case, one would only need this URL object to convert a U-LABEL to an A-LABEL, this would be much simpler.

Because of that, after extracting the domain from the URL, the domain must be converted (again) with the proper options sent to the idna-uts46 lib:

```
const domain_ascii = idna.toAscii(domain_normalized, {
   transitional: false,
   useStd3ASCII: true,
   verifyDnsLength: true});
```

- Transitional=false ensures we are on the IDNA2008 protocol;
- useStd3ASCII=true ensures we allow only LDH domains;
- verifyDnsLength=true ensures we do not allow a domain label to exceed 63 characters

Client-side (Browser JavaScript engine)

The implementation of the URL object used above in many browsers use IDNA2003. Thus, we must slightly modify the code above and use an RFC compliant library like “uri-js” to extract the domain from the URL:

```
let url = undefined;

try {
    url = URI.parse(website, {unicodeSupport: true});
} catch (err: any) {
    console.log(`[uri-js] parsing URL ${website} failed: ${err?.message}`);
    return {
        invalidUrl: true
    };
}
const domain = url.host;
console.log(`[uri-js] successfully parsed the domain ${domain} from URL ${website}`)
try {
    const domain_ascii = idna.toAscii(domain, {
        transitional: false,
```

```
        useStd3ASCII: true,
        verifyDnsLength: true
    });
    if (domain !== domain_ascii) {
        console.log(`[idna-uts46] successfully convert ${domain} to ascii ${domain_ascii}`);
    }
    return null;
} catch (err) {
    return {
        invalidUrl: true
    }
}
return null;
```

Notes

The client-side version can be also used server-side.

Bug reports

- https://github.com/nodejs/node/issues/41976
- https://github.com/nodejs/node/issues/41977

validator

Snippets

Validator.js provides a simple API (isEmail function) that is EAI compliant (see this compliance benchmark: https://viagenie.ca/ua/test-results-20200814.html#js-validator-title):

```
function eai_validate(email, results) {
        if (!validator.isEmail(email)) {
                return results.addResult(true, `[validator] Email address ${email} is invalid`, email);
        }
        return results.addResult(false, `[validator] Email address ${email} is valid (no DNS verifications)`, email);
}
```

Notes

Validator.js can be used as-is server-side and client-side.

nodemailer

Snippets

Nodemailer is a smtp client for sending email in server-side JavaScript. Nodemailer automatically transforms the domain part into an A-LABEL when needed and checks the SMTPUTF8 extension. However, and surprisingly, it does not raise an exception when we send an email with non-ASCII character in the local part to a SMTP server not supporting the SMTPUTF8 extension. This snippet manually checks with the custom “support_SMTPUTF8” function the support of the extension:

```
const SMTPConnection = require("nodemailer/lib/smtp-connection");

class SmtpClient {
```

```
        constructor(nodemailer_transport) {
                this.client = nodemailer_transport;
        }
        async support_SMTPUTF8() {
                let connection = new SMTPConnection(this.client.options);
                return new Promise(contains_SMTPUTF8 => {
                        connection.connect(() => {
                                // wrapper to access _supportedExtensions. Since the maintainers did not
                                // make it public API, we cannot guarantee this name won't change in future
                                // versions of nodemailer
                                contains_SMTPUTF8(connection['_supportedExtensions'].includes('SMTPUTF8'));
                                connection.close();
                        });
                });
        }
}
async function eai_smtp(email, smtp_client, results, msg) {
        if (!email.includes('@') || !email.substring(email.lastIndexOf('@') + 1).includes('.')) {
                return results.addResult(true, "[sample] email does not contains the '@' AND '.' characters in this order",
email);
        }
        const localpart = email.substring(0, email.lastIndexOf('@'));
        const is_localpart_non_ascii = /[\u0080-\uFFFF]/.test(localpart);
        // we don't need to check the domain part, since it will be converted into a A-LABEL automatically by
nodemailer
        if (is_localpart_non_ascii && !await smtp_client.support_SMTPUTF8()) {
                return results.addResult(true, '[Note] SMTP server does not support SMTPUTF8', email);
        }
        try {
                await smtp_client.client.sendMail({
                        from: "ua@test.org",
                        to: email,
                        subject: "Registration successful",
                        text: msg
                });
        } catch (err) {
                return results.addResult(true,`[nodemailer] Failed to send email: ${err.message}`, email);
        }
        return results.addResult(false,`[nodemailer] Email has been sent successfully. See it <a
href="${process.env.MAILHOG}" target="_blank">here</a>!`, email);
}
```

Bug reports

- https://github.com/nodemailer/nodemailer/issues/1378