Scenario 3: How to make UA-ready applications

Scene 1

Visual: Random words in multiple scripts (Arabic, Chinese, Cyrillic, Devanagari, etc.) scroll through the screen diagonally, like it is slowly raining words. They come in and out of focus.

Overlay: "Universal Acceptance for Developers."

Narration: Universal Acceptance, or UA, is a transformative approach ensuring that all valid domain names and email addresses, regardless of script, language, or length, are accepted by all applications and systems. Unfortunately, not all systems are properly set up to handle this diversity yet.

Scene 2

Note: Our example email is 用户@例子.商业’(translation: user@example.business).

Visual: The scene is split between the left side and the right side of the screen, divided by a line of some sort, which may be stylized. Each side displays a different website.

On the left side, an older, rigid and Westen website is displayed. There is a form field and we show the characters being typed. An error window pops up when ‘OK’ is clicked. The pop up reads: “用户@例子.商业 is not a valid email address.”

Below and to the left, after the error is shown, in an old computer (terminal) style font: 'Error 400: Invalid character encoding'

On the right, a more modern and vibrant website is displayed. After following the same process as the other, a successful registration occurs, which leads to a message being displayed below the form: “Thank you for joining us!”

Below and to the right, stylized as before: ‘User registered with username 用户@例子.商业’

Narration: This example shows the difference between a website that is UA-ready and one that is not. In both cases, the objective is for the user to register themselves using a form, but when attempting to use an Internationalized Email Address, results are quite different. IDNs are part of global technical standards, [**Visual**: flash relevant RFCs like ‘RFC 5891’ on the screen] and should be supported in the same way as any other email address.

Scene 3

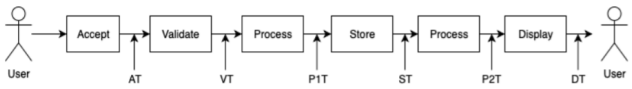
Visual: The UA computer mascot appears, explaining the subject with a concentrated expression, pointing to a blackboard. Different devices appear on the board as the narration goes on, like a smartphone, a computer, a router and so on. Additionally, icons that approximate the applications listed should attempt to be displayed if there is a sensical way to do it.

Narration: Such limitations emerge from the various components of software, such as programming languages, libraries, databases, and utilities. The medium through which user interaction occurs also matters, such as browsers and email clients. This means there is a need to test, configure, and potentially adapt these different components in order to become compliant with Universal Acceptance principles.

Scene 4

Visual: [By and large this can be a copy the UA Verbs general presentation from the first video, making relevant changes based on the text below]

Narration: These different components are abstracted into five verbs, which are 'Accept, Validate, Process, Store, and Display'. These elements normally work in succession, making them into a chain of sorts. If one step fails, the next one will likely fail as well. This is why creating Universal Acceptance is a process which requires the revision of different parts of a system, seeing as it might be the case that a single component is at fault, and the fix might be simpler than it is imagined. Of course, this is different in each case.



[Instead of "User", it should read "API"; each end should be a computer instead of a person to reflect that]

Transition

Narration: Let us study each verb individually now.

Scene 5: Accept

Visual: Here we start a sequence of scenes that compose a journey through the five verbs of UA, which should flow from one to the other in a seamless manner, with stylistic consistency. The focus is on illustrating what is being described in the narration, but in more interesting

ways. These should not be full-blown animated scenes, but still contain enough motion to keep the viewer engaged.

In this first scene, we focus on reproducing an input interface, much like has already been done in the previous videos.

Narration: Many applications have input fields that the user is asked to complete. Generally, more attention is paid to accepting all characters in a person's name, although at times users are still asked to write a transliteration of their name or omit diacritics. For email addresses, developers need to ensure the input field accepts all valid characters, including those in Unicode.

Successful acceptance of all characters normally involves enabling UTF-8 input in the programming language or framework being used [**Visual**: Print on screen the following: "encoding='utf-8'"] and ensuring that the environment is configured to expect that input. Java Swing and other current GUI solutions inherently support UTF-8 input, potentially making this step quite simple.

Scene 6: Validate

Visual: When citing regular expressions, this should be printed on the screen: "^[A-Z0-9+_.-]+@[A-Z0-9.-]+$", and then marked with a cross for being the wrong approach. To illustrate the libraries, it is a good analogy to think of books, but in this case, each library is an encyclopedia that contains a lot of information about a specific subject. By using the right encyclopedias, the user validates EAI addresses without issues.

Narration: If accepting input is important, then validation is essential, as it flags a given email address as valid and usable. Our studies show that many coders and libraries still use regular expressions to perform validation, which makes UA difficult to achieve, given that an ASCII-based logic is usually employed in such cases. a-z 0-9.

There are, however, a series of compliant libraries that make this implementation much easier, such as ICU for comprehensive Unicode support in Java and C/C++, and IDNA for handling internationalized domain names in Python and Rust. These libraries also assist in supporting longer Top-Level Domains that exceed the maximum of four characters that used to be expected in the past. By performing validation in this way, developers can offload the heavy lifting to specialized teams that actively maintain and test such libraries, saving time and avoiding errors.

Overlay, code is typed on screen:

```
import idna
```

```
def handle_idn(domain):
  try:
    # Encode the domain to ASCII using Punycode
    ascii_domain = idna.encode(domain).decode('ascii')
    print(f"ASCII domain: {ascii_domain}")

    # Decode back to Unicode
    unicode_domain = idna.decode(ascii_domain)
    print(f"Unicode domain: {unicode_domain}")
  except idna.IDNAError as e:
    print(f"Error handling IDN: {e}")
```

S**cene 7: Process**

Visual: Different aspects of processing will be illustrated as the narration mentions them, in this way:

- **Normalization**: Start with the word “Café” and animate the decomposition of it into the formats below, with the title of each format above them.
    - **Word**: Café; **Precomposed Form**: é (U+00E9); **Decomposed Form**: e (U+0065) + ´ (U+0301)
- **Decoding**: Start with both elements separated and show the combination of both codes into the assembled character “é”.
    - (U+0065) + (U+0301) = é
- **Sanitization**: A filter (or something that invokes the same idea) has Unicode text passing through it. We show harmful characters being denied entry and kicked out of the filter. The most relevant malicious characters are the one below, comma-separated.
    - &, <, >, ", ', /, --, %, ?, =, #, +, :, `

Narration: Proper input processing maintains the integrity of user data, ensuring it is consistently interpreted and managed. One of the key aspects of this step is data normalization, transforming Unicode data into canonical form, which facilitates tasks such as searching and alphabetizing. Processing also involves testing the decoding of data to ensure compatibility and integrity, making sure characters are being correctly interpreted. Finally, an important additional security practice related to this activity that is worth mentioning is sanitization of input in order to remove or escape harmful characters, as this prevents potential security vulnerabilities such as injection attacks.

A tool that assists these various tasks is the previously mentioned ICU library, which includes various functions for Unicode normalization and other text processing tasks. For Python, the “unicodedata” library performs a similar task.

Overlay, code is typed on screen:

```
import com.ibm.icu.text.Normalizer2;

public class NormalizeUnicode {
   public static void normalizeUnicode(String text) {
      // Get an instance of the NFC normalizer
      Normalizer2 normalizer = Normalizer2.getNFCInstance();

      // Normalize the text
      String normalizedText = normalizer.normalize(text);

      System.out.println("Original text: " + text);
      System.out.println("Normalized text: " + normalizedText);
   }

   public static void main(String[] args) {
      // Example usage
      String text = "your text here";
      normalizeUnicode(text);
   }
}
```

```
import unicodedata

def normalize_unicode(text):
   # Normalize the text to NFC (Normalization Form C)
   normalized_text = unicodedata.normalize('NFC', text)
   print(f"Original text: {text}")
   print(f"Normalized text: {normalized_text}")
```

Scene 8: Store

Visual: Here we can attempt to follow the library theme. The database is like a bookshelf that is progressively filled with documents and books, that at times need to be reorganized and shuffled around. We can show how Unicode documents can't fit or get stored in an older decaying bookshelf, while they fit nicely in a modernized UTF-8 bookshelf. The illustration of the rest of the narration can follow along these same general lines.

Narration: For our purposes, databases need to be set up to support UTF-8 encoding, which can be achieved both in SQL and NoSQL. Proper configuration ensures that all characters are stored accurately, preserving information integrity for both writing and reading. For instance, in MySQL, character set and collation need to be configured, while many NoSQL solutions like

MongoDB already natively support UTF-8. Additionally, configuring indexes and full-text search features to account for UTF-8 can significantly improve performance and accuracy when querying this data.

Overlay, code is typed on screen:

```
-- MySQL database
ALTER DATABASE YourDatabase CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Scene 9: Display

Visual: This can follow the same general pattern of “Accept”, as it is mostly about showing user interface actions.

Narration: Displaying internationalized data correctly involves considering a few aspects. First, fonts used by the application should support a wide range of characters, and employing font fallback mechanisms can be a useful safeguard. Another key consideration is the display of characters in Right-to-Left (RTL) orientation, which is required for Semitic languages, with the application dynamically adjusting text direction based on necessity.

This also calls for the implementation or revision of responsive design techniques to adapt the UI for different scripts and languages, ensuring that the desired spacing and layout is maintained in all scenarios.

Scene 10

Overlay: During the whole scene, this link is shown centered at the bottom: https://github.com/icann/ua-code-samples

Visual: We briefly show the GitHub main repository page [https://github.com/icann/ua-code-samples] and transition to the zoomed in readiness folder [https://github.com/icann/ua-code-samples/tree/master/readiness-sample-code], panning it down slowly to showcase the different subfolders within it. Different relevant code snippets are then displayed, as shown below.

Java

```
public class Icu4jMain extends IdnaMainRunner {

 public Icu4jMain() {
  cliName = "icu4j-sample";
```

```
  domainHelp = "The domain name to convert in A-Label";
 }
```

PHP

```
  protected function convertUrlToALabel(): ?string
  {
     $idnConverter = new IDNConverter();
     // quick hack to transform host to A-label
     $domain = parse_url($this->url, PHP_ URL_HOST);
     $convertedDomain = $idnConverter->convert_to_alabel($domain);
     if (!$convertedDomain) {
        return null;
     }
     $pos = strpos($this->url, $domain);
     return substr_replace($this->url, $convertedDomain, $pos, strlen($domain));
  }
```

Kotlin

```
object IDNAUtils {

  private const val flags =
                IDNA.CHECK_BIDI or IDNA.CHECK_CONTEXTJ or IDNA.CHECK_CONTEXTO or
IDNA.NONTRANSITIONAL_TO_ASCII or IDNA.USE_STD3_RULES
  private val idna: IDNA = IDNA.getUTS46Instance(flags)
  private val info = IDNA.Info()

  /**
   * Convert URL host in ASCII with IDNA 2008 compliant
   */
  fun hostToAscii(url: String): String {
     val inUrl = URL(url)
     val hostAscii = domainToAscii(inUrl.host)
     return URL(inUrl.protocol, hostAscii, inUrl.port, inUrl.file).toString()
  }
```

Narration: Developers do not need to start from scratch to achieve Universal Acceptance, as the ICANN Github repository hosts a set of minimal viable products and best practices to help on this matter, available within the ‘ua-code-samples’ folder of the repository. These resources are available in multiple programming languages, namely: Java, JavaScript, Kotlin, PHP, Python, and

Swift. The code samples can serve both as bases and as examples, and their usage is open to any interested project.

Scene

Visual: Gradually display an approximation of a timeline illustrating a phased approach to UA Readiness implementation. These visuals should be made up mostly of already existing assets/styles from prior scenes to exemplify how knowledge is built up over the video and this is the same process but organized differently.

- **Phase 1**: New long and short ASCII TLDs
- **Phase 2**: Integration of IDNs
- **Phase 3**: Full support of EAI

Narration: UA implementation can also be divided into phases in order to make it more manageable. This is what ICANN itself did, as detailed in the case study published by the Universal Acceptance Steering Group on uasg.tech.

The first step is to ensure that newer ASCII-based TLDs work, such as those using more than 4 characters. Next, integration should focus on supporting Internationalized Domain Names, which is well-documented and widely supported. The final step is to ensure that Email Address Internationalization is fully integrated, which requires heavier lifting, but also builds upon the previous phases. This makes it so that achieving UA Readiness can be a project that generates smaller deliverables and in which progress is measurable.

Overlay: “ICANN UA Implementation Case Study” https://uasg.tech/download/uasg-013c-2-icann-case-study-en/

Scene

Visual: A globe with various languages and scripts radiating from it, symbolizing global connectivity. Text prompts such as "Update Your Systems," "Innovate Your Product," and "Join the Global Movement" are displayed."

Narration: Enhance your competitive edge and contribute to a multilingual, interoperable Internet. Update your systems, innovate your products, and make them UA-ready. Join leading companies in this movement and open your services to the next billion users. The time to act is now.

Catchy slogan: "Evolve! Embrace UA! Connect the world one email at a time!"