Localizing Capsules

If you want to make your capsule accessible worldwide, you need to make your capsule available in multiple languages. This guide describes how to create translatable files for your capsule.

Make sure any text you write for Bixby follows the dialog best practices as well as the Writing Dialog Design Guide.

Separate Files By Target

As described in the Organize Your Capsule Resources topic, you can separate your files by target, which can be further separated by more specific information such as device spec (like `bixby-mobile`), locale (like `en-US`), or even language (`en`). Because these higher-level resource folders exist, you can further separate your language-specific files into these folders.

The following files should have separate localized versions under each locale or language your capsule supports:

- Hints (`hints.bxb`)
- Capsule Info (`capsule-info.bxb`)
- Localized strings for dialog
- Localized strings for UI
- Training
- Vocabulary
- Navigation Support (`navigation.bxb`)

String Externalizing

Most translation processes involve sending separate files of pre-formatted strings to a localization group. That group translates these files of strings into various languages (manually or through software), without knowing the code that these strings accompany. We recommend following a

similar process of externalizing your text (in dialogs and layouts) for maximum efficiency when localizing your capsules. You would do this through macros for dialog.

Specifically, we recommend that you have a separate `macro-def` file for all layouts and, when reasonable, for dialogs. Then, for each language, you can place a translated copy in the appropriate resources language folder.

Your workspace might look similar to this:

```
resources/
    - base/
      - views/
        - MyCapsule.Results.view.bxb
        - MyCapsule.Input.view.bxb
      - capsule-info.bxb
    - en/
      - dialog/
        - MyCapsule.MyConcept.dialog.bxb
        - MyCapsule.UI-string.dialog.bxb
      - MyCapsule-en.hints.bxb
      - navigation.bxb
    - ko/
      - dialog
        - MyCapsule.MyConcept.dialog.bxb
        - MyCapsule.UI-string.dialog.bxb
      - capsule-info.bxb
      - MyCapsule-ko.hints.bxb
      - navigation.bxb
...
```

This example workspace assumes that the `base/capsule-info.bxb` file is in English, so you wouldn't necessarily need to add an additional one under the `en/` folder.

Externalizing strings enables you to easily identify what text needs to be translated without requiring your translation team to understand the complexity of underlying code.

Note

While you can still define `template-macro-def` for dialogs and `layout-macro-def` for layouts, those keys will be deprecated in a future release. We recommend you switch to using more generic macros. For more information, see Reusing Content With Macros.

Translated Dialog

This section briefly discusses the tools you can use when writing dialogs. For more information on dialogs in general, see Refining Dialog in the developer's guide and Dialog Reference in the reference documentation.

Dialog Fragments

Dialog fragments are a powerful tool that enable you to reuse dialog text multiple times in multiple places.

For example, you can use the following basic concept Dialog for a `SpaceResort`:

```
dialog (Concept) {
  match: SpaceResort (this)
  switch (plural(this)) {
    case (One) {
      template (space resort)
    }
    default {
      template (space resorts)
    }
  }
}
```

View master on GitHub

You can learn more about how to implement dialog fragments. Note that using dialog fragments involve trade-offs.

Dialog Template Macros

Dialog template macros are another useful tool that enable you to reuse dialog in multiple places. You define the text that is replaceable with the `macro-def` key, and then use `macro` to invoke the definition.

Note that you can choose to have as many macro definitions for dialog in a single file as you want. Here is an example for a space resorts capsule in which you declare multiple dialog macros:

```
macro-def (RECEIPT_ACTIVITY_CARD_MESSAGE) {
  params {
    param (receipt) {
      type (Receipt)
      min (Required) max (One)
    }
  }
  content {
    template ("Suit up, your trip[ to #{value(receipt.item.spaceResort.planet)}]
  }
}

macro-def (RECEIPT_ACTIVITY_CARD_DETAILS) {
  params {
    param (receipt) {
      type (Receipt)
      min (Required) max (One)
    }
  }
  content {
    if (receipt.canceled) {
      template ("Reservation canceled")
    } else {
      template("Reservation for #{value(receipt.item.numberOfAstronauts)}")
    }
  }
}

macro-def (RECEIPT_RESULT) {
  params {
    param (receipt) {
      type (Receipt)
      min (Required) max (One)
    }
  }
  content {
    if ("exists(receipt.canceled) && receipt.canceled == true") {
      template ("Your trip has been canceled.")
    } else {
      template ("I hope you have a great time[ at #{value(receipt.item.spaceResc
    }
  }
}

macro-def (SPACE_RESORT_SELECTION) {
```

```
  content {
    template("Which space resort would you like?")
  }
}

macro-def (NUMBER_OF_ASTRONAUTS_SELECTION) {
  content {
    template ("How many astronauts?")
  }
}

macro-def (HABITAT_POD_SELECTION) {
  content {
    template ("Which habitat pod would you like?")
  }
}

macro-def (DATE_INTERVAL_ELICITATION) {
  content {
    template ("When do you want to stay?")
  }
}

macro-def (COMMIT_ORDER_CONFIRMATION) {
  content {
    template ("Are you sure you want to book this trip?")
  }
}

macro-def (SPACE_RESORT_RESULT) {
  params {
    param (result) {
      type (SpaceResort)
      min (Required) max (Many)
    }
  }
  content {
    if (size(result) == 1) {
      template ("Check out this #{concept(result)}.")
    } else {
      template ("I found these #{concept(result)}.")
    }
  }
}

macro-def (HIGHLIGHT_LABEL_BY_PREFERENCES) {
```

```
  content {
    template ("Based on your preferences")
  }
}

// Exception handling
macro-def (UNSUPPORTED_SEARCH_OPTION) {
  content {
    template ("Unsupported search criteria.")
  }
}

macro-def (UNKNOWN_ERROR) {
  content {
    template ("Something's wrong.")
  }
}
```

View master on GitHub

You can learn more about implementing macros in the developer's guide.

Translated Views

We recommend creating a `UI-strings.macro.bxb` file for your capsule that handles any UI text for your layouts. For example, in this space resorts capsule, you have a `ui-strings.macro.bxb` file that contains `macro-def` declarations for all text in the UI. You can then use the `macro` keys in your views, layouts, and macros for layouts.

Here is an example:

```
macro-def (MakeReservation) {
  content {
    template ("Make reservation")
  }
}

macro-def (Details) {
  content {
    template ("Details")
  }
}
```

```
macro-def (NumberOfAstronauts) {
  content {
    template ("Number of astronauts")
  }
}

macro-def (Dates) {
  content {
    template ("Dates")
  }
}

macro-def (Package) {
  content {
    template ("Package")
  }
}

macro-def (ContactInformation) {
  content {
    template ("Contact information")
  }
}

macro-def (Name) {
  content {
    template ("Name")
  }
}

macro-def (PhoneNumber) {
  content {
    template ("Phone number")
  }
}

macro-def (Email) {
  content {
    template ("Email")
  }
}

macro-def (AstronautOne) {
  content {
    template ("astronaut")
```

```
  }
}

macro-def (AstronautMany) {
  content {
    template ("astronauts")
  }
}

macro-def (CancelOrder) {
  content {
    template ("Cancel Order")
  }
}

macro-def (KeepIt) {
  content {
    template ("Keep it")
  }
}

macro-def (NotSureWhatToCancel) {
  content {
    template ("Not sure what to cancel. I didn't find any recent orders.")
  }
}

macro-def (ThisOrderIsAlreadyCanceled) {
  content {
    template ("This order is already canceled!")
  }
}

macro-def (Book) {
  content{
    template ("Book")
  }
}
macro-def (ReceiptOrderTotal) {
  params{
    param (value) {
      type (viv.core.Text)
    }
  }
  content {
    template ("Order total: [#{value}]")
```

```
    }
  }
```

View master on GitHub

Note

While you should localize the UI strings for your capsule, we recommend that you try to keep the same main views and layouts files in your main `base` folder, to keep the look-and-feel of the capsule consistent between languages.

For more information on developing UI, see Building Bixby Views in the developer's guide.

Layout Macros

Layout macros are a way for you to use shared layouts in your capsules. While layout macros aren't necessary to localize your capsule, they can be useful when combined with macros for dialog. Similar to macro definitions for dialog, you use a `macro-def` file to define a general layout with different parameters as needed. Then, you use or `macro` to invoke the correct layout macro by using the ID you defined in the macro definition.

For more information on how to implement layout macros, see the Reusing Content With Macros Developers' Guide.

Trade Offs of Fragmentation

It is important to note that there is a trade-off between creating compositional dialog versus keeping entire sentences together for translation.

As explained in dialog fragments, there are several benefits to creating these building blocks of text. If you have the same concept that appears in 20 different places in your capsule's dialog, you would have one concept fragment dialog file that handles this text. If you have your capsule available in 4 different languages, this now becomes 80 different places you might have to keep track of and change!

However, depending on the language, these building blocks might not compose comprehensible sentences once translated. The resulting utterances from Bixby would then end up sounding

unnatural or worse, incorrect!

For example, English has no concept of gender with respect to inanimate objects like other Romance languages. When developing a capsule to find furniture for someone based on their favorite color, you might use the following dialog:

```
dialog (Concept){
  match: FindTables (output) {
    from-input: FavoriteColor (action)
  }
  template ("Here are some #{value(action.color)} tables!")
}
```

Your result prints *Here are some red tables* in English but the translated Spanish text might read *Aquí hay algunas mesas **rojos*** instead of the correct *Aquí hay algunas mesas rojas*. While this is clearly an over-simplified example, it can easily be extended to more complex situations, especially considering the different grammar between languages.

Therefore, when developing, keep in mind that there might be certain situations where your dialog would benefit from keeping output whole or from just depending on the platform to craft its own response.

Storage Prompts

To localize storage prompts, which are generated when Bixby asks for permission to store a preference in the user's profile, you can create localized `*.profile.bxb` files. These files contain `profile-support` blocks that specify match patterns and new dialog for prompts. These files go in the appropriate resources language folder, just like `macro-def` files:

```
resources
    - en
        - MyStoragePrompt.profile.bxb
    - ko
        - MyStoragePrompt.profile.bxb
```

An example `profile.bxb` file looks like this:

```
profile-support {
  match: FlightPreferences
```

```
    storage-prompt {
      confirmation-type (PreferenceConfirmation)
      message (Store your flight preferences?)
      confirm-label (Yes)
      deny-label (No)
    }
  }
```

For more details, see the `profile-support` reference documentation.

Prepare Your capsule.bxb File

In order to make sure that Bixby properly interprets the contents of your capsule, you will need to make sure that your `capsule.bxb` file lists all the `target` devices and locales that you want supported, as explained in the Provide Targets section of preparing your capsule. If you have a `target` specified, but do not want it enabled, set that target's `enabled` key to `false`.

The list of supported devices and locales are in the `target` reference.

You can also further customize the countries and device models that your capsule supports by providing store countries and providing supported device models.

Localized Training

You need to train your capsule in **each** localized language, so that Bixby can learn how to interact with its users.

1. Make sure you've added the targets to your `capsule.bxb` file.
   This step ensures that the right devices and languages can be set when creating a new training file.
2. Make sure your capsule is synced.
3. Start localized training!

It's important to note that training and annotating the utterances work essentially the same across all languages! While you might have subtle differences in tokenizing words, the general guidelines are the same for all locales. If there is an issue with your training entry, Bixby Developer Studio will

inform you of the invalidity. You should make sure to thoroughly train your capsule per language to improve Bixby's interaction with all users.

Related Resources

Developer Guides

- Introduction to Dialog
- Layouts
- Layout Macros
- Dialog Macros
- Reusing Content With Macros
- Working with the Marketplace
- Training for Natural Language
- Designing Conversations
- Writing Dialog
- Dialog Best Practices

Reference Docs

- dialog
- Dialog Modes
- macro-def
- capsule.targets.target
- profile-support