201

Intermediate Bixby: Modeling & JavaScript

Bixby Developers

Bixby Resources - Reminder

Bixby Homepage
www.bixbydevelopers.com

Bixby on Github
github.com/bixbydevelopers

Bixby Tutorial Videos
bixby.developer.samsung.com/newsroom/

Agenda

- Concept Modeling
- Action Modeling
- Endpoints
- JavaScript Actions
- Properties/Secrets
- Library Capsules

Concept Modeling - Enumerations

- Constrain to a list of possible values
- Often use with Vocabulary (`vocab` file). Allows for synonyms
- Vocabulary is required if an `enum` is used in natural language (training)
- Example
  - `enum` for months*
  - `vocab` for abbreviations

```
enum (Month) {
  description (Months of the year)
  symbol (January)
  symbol (February)
  symbol (March)
  symbol (April)
```

```
vocab (Month) {
  "January" {"January" "Jan"}
  "February" {"February" "Feb"}
  "March" {"March" "Mar"}
  "April" {"April" "Apr"}
```

* When using dates, check if the viv.time library capsule can be used for flexibility

Concept Modeling - Vocabulary

- Vocabulary allows setup of synonyms
- In an action, any synonym will be accepted. Will resolve to the original word

```
enum (MovieGenre) {
  description (MovieGenre)
  symbol (action)
  symbol (adventure)
  symbol (animation)
  symbol (comedy)
  symbol (documentary)
}
```

```
vocab (MovieGenre) {
  "action" {"action"}
  "adventure" {"adventure"}
  "animation" { "animation" "animations" "cartoon" "cartoons" }
  "comedy" { "comedy" "comedies" "funny" "funniest" }
  "documentary" { "documentary" "docudrama" "narrative" }
}
```

Concept Modeling - Extends

- `extends`: Like the OO inheritance extends in many languages
- Classic Animal example:

```
structure (Animal) {
  description (Animal Information)
  property (name) {
    type (Name)
    min (Required) max (One)
  }
}
```

```
structure (Bird) {
  description (Bird Information)
  extends (Animal)
  property (canFly) {
    type (CanFly)
    min (Required) max (One)
  }
}
```

Concept Modeling – Role-Of

- `role-of` creates a new "instance" of a concept. It cannot be extended
- Typical use case: re-use Enums

```
enum (City) {
  description (California Cities)
  symbol (Los Angeles)
  symbol (San Diego)
  symbol (San Jose)
  symbol (San Francisco)
```

```
enum (DepartureCity) {
  description (Departure City)
  role-of (City)
}
```

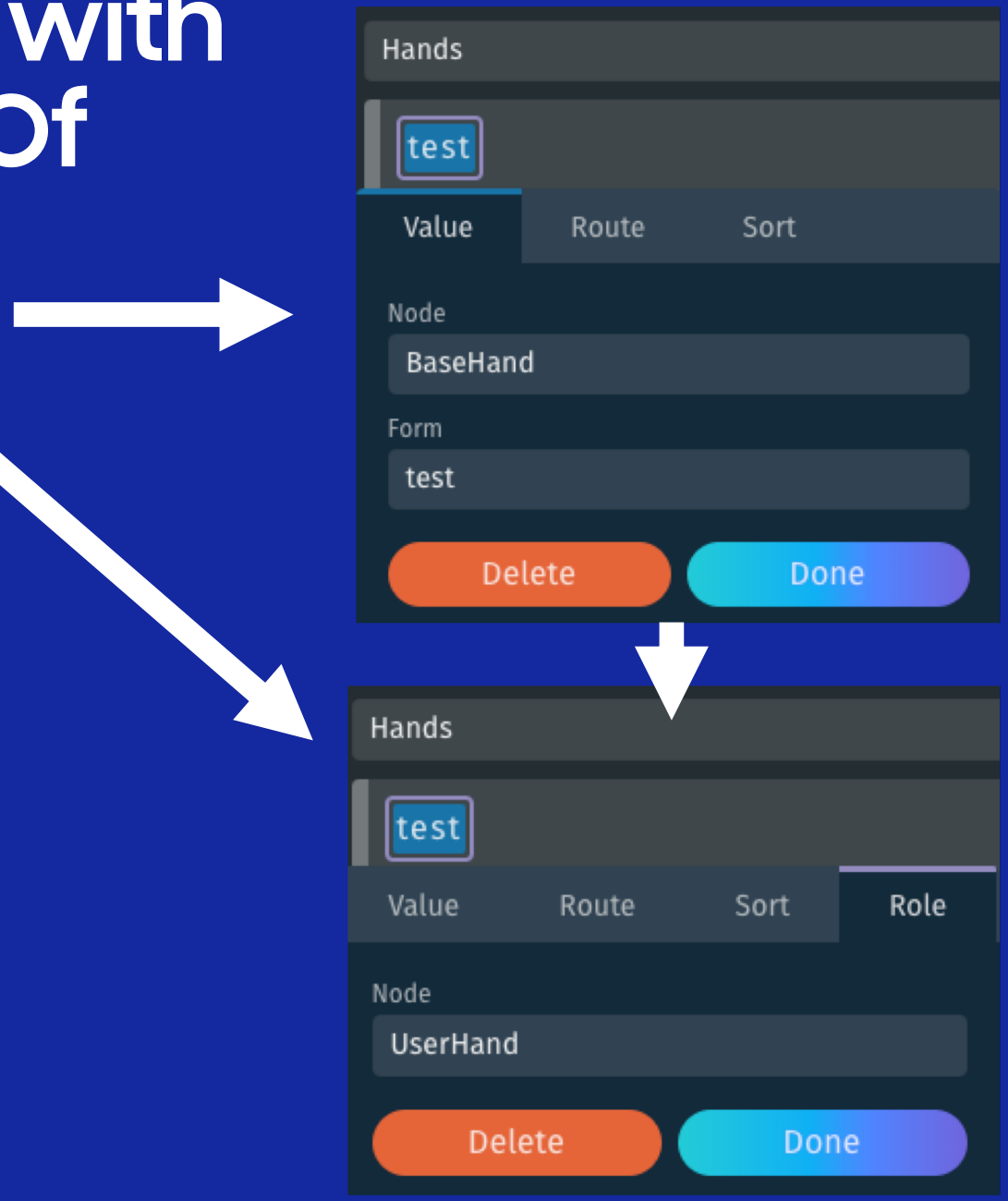
```
enum (ArrivalCity) {
  description (Arrival City)
  role-of (City)
}
```

Training with Role-Of

- Training with role-of requires:
  - Se the Value Node to the base concept
  - Set Role Node to the concept using role

```
enum (BaseHand) {
  description (base hand concept)
  symbol (test)
```

```
1 enum (UserHand) {
2   description (User's hand)
3   features {
4     transient
5   }
6   role-of (BaseHand)
7 }
```



Action Modeling - Cardinality

Cardinality defined by

- `min`: `Optional` or `Required`
  If `Required` and the action is called, Bixby will elicit input
- `max`: `One` or `Many`: If `One` and more than one input is provided, Bixby will prompt the user to select one

```
action (FindCapital) {
  description (Find State Capital)
  type (Fetch)
  collect {
    input (state) {
      type (State)
      min (Required) max (One)
    }
  }
  output (StateInfo)
}
```

Action Modeling – Prompt Behavior

Q: You want Bixby to elicit a response but answering it is optional. How do you do this?

```
collect {
  input (firstName) {
    type (contact.FirstName)
    min (Optional) max (One)
    prompt-behavior (AlwaysSelection)
  }
```

A: Use `prompt-behavior` – this tells Bixby to always elicit a response. If the concept is not required, Bixby will accept a null input

Action Modeling – Input Groups

- Allow cardinality to span across multiple inputs – often used in search
- Example: find person based on first or last name
- requires
  - `OneOf` – exactly one
  - `OneOrMoreOf` – one or more
  - `ZeroOrOneOf` – at most one
  - `ZeroOrMoreOf` – zero or more

```
action (FindPerson) {
  description (FindPerson)
  type (Search)
  collect {
    input-group (name) {
      requires (OneOrMoreOf)
      collect {
        input (firstName) {
          type (contact.FirstName)
          min (Optional)
        }
        input (lastName) {
          type (contact.LastName)
          min (Optional)
        }
      }
```

Action Modeling – Input Validation

- Simple validation possible in actions
- Structure: `validate` + `if`
- More complex validation should be done in JavaScript

```
input (number) {
  type (Number)
  min (Required) max (One)
  validate {
    if (number < 1 || number > 5) {
      prompt {
        dialog ("Please enter a number between 1 and 5")
      }
    }
  }
}
```

Action Modeling – Error Handling

- An error thrown by JavaScript can be caught in an action
- After being caught, several options to handle. See docs/example code
- Note http errors are handled this way as well

```
module.exports.function = function echoNumber(number) {
  if (number % 2 == 0) {
    return (number)
  }
  throw fail.checkedError('Number not even', 'NumberNotEvenError', {})
  return 'Not even';
}
```

```
action (GetNumber) {
  type (Calculation)
  description (get number)
  collect {
    input (number) {
      type (Number)
      min (Required) max (One)
    }
  }
  output (TheNumber) {
  throws {
      error (NumberNotEvenError) {
        on-catch {
          replan {
            dialog("Please enter an even number. ")
            intent {
              goal: TheNumber
              route: GetNumber
            }
```

Github Examples: github.com/bixbydevelopers/capsule-samples-collection/tree/master/error-handling

Action Modeling - Default Values

- Use `default-init` to specify default values
- Option: Add `prompt-behavior` to elicit an input with the default value

```
collect {
  input (number) {
    type (Number)
    min (Required) max (One)
    default-init {
      intent {
        goal {Number}
        value {
          Number (4)
        }
      }
    }
    prompt-behavior (AlwaysSelection)
  }
}
```

Action Modeling – Instantiation Strategy

- Separate `[name].strategy.bxb` file
- Easy way to specify multiple options for input
- Also useful for getting smart defaults for location, currency and other user values – see the documentation

```
instantiation-strategy {
  id (default-color)

  match {
    Color
  }

  strategy {
    intent {
      value: Color (Red)
      value: Color (Green)
      value: Color (Blue)
      goal: Color
    }
  }
}
```

Endpoints - Local vs Remote

- Remote endpoints are flexible as far as the API call
  - Parameters
  - Authorization
  - HTTP methods
  - Headers
- BUT, data from remote endpoints must map to the action output object – so flexibility is limited
- For remote endpoint calls with greater flexibility, use a local endpoint and call the remote endpoint in Javascript – use the http class

JavaScript – REST*

- http object supports: get, post, delete, put
- No needs for promises/callbacks like traditional JavaScript - async is built in
- 3 parameters
  - url
  - options: for get, use query to add parameters to the call
  - params: for post, delete and put, values for the body of the request

```
var id = "1234", location = "here"
var response = http.getUrl('https://api.example.com', {
  format: 'json',
  query: {
    location: location,
    id: id,
  }
})
```

https://api.example.com?location=here&id=1234

*SOAP is also supported – see documentation

Github - github.com/bixbydevelopers/capsule-samples-collection/tree/master/http-api-calls

Javascript - $vivContext

- `$vivContext` is a useful parameter to pass to your Javascript functions
- Multiple attributes including
  - User data e.g. locale, time format etc.
  - Device information
  - Permissions
- See the documentation

```
endpoints {
  action-endpoints {
    action-endpoint (GetNumber) {
      accepted-inputs (number, $vivContext)
      local-endpoint (EchoNumber.js)
    }
  }
}
```

```
module.exports.function = function echoNumber(number, $vivContext) {

  console.log ("locale = " + $vivContext.locale)
```

Github - github.com/bixbydevelopers/capsule-samples-collection/tree/master/user-context

Capsule.properties

- `Capsule.properties` – store static properties
- Handles modes e.g. default in example
- `config.get` in JavaScript
- Sensitive data e.g. API keys, passwords should not be stored here

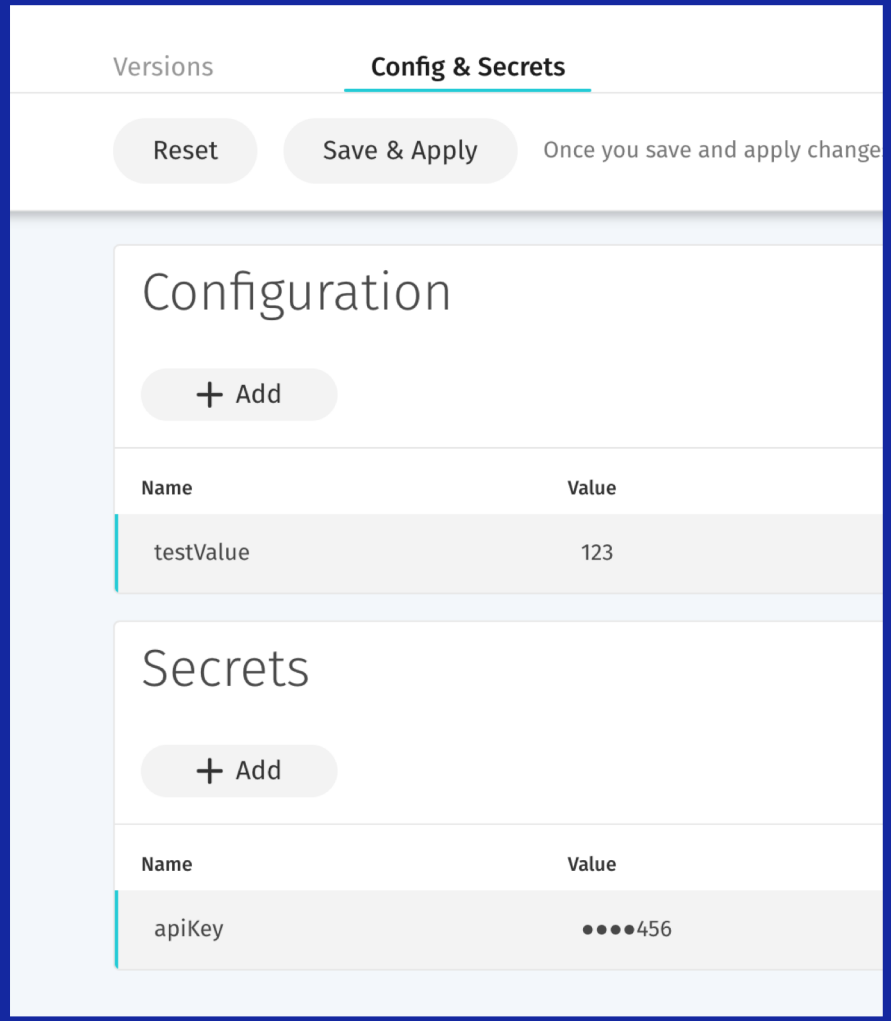
```
capsule.config.mode=default
config.default.testValue=123
config.default.foo=abcd
config.debug.foo=xyz
```

```
console.log("testValue = " + config.get("testValue"));
console.log("foo = " + config.get("foo"));
```

testValue = 123
foo = abcd

Config/Secrets

- Sensitive data e.g. API keys, passwords etc
- Must be stored in Config/Secrets – name/value pairs
- In Teams and Capsules section of BixbyDevelopers.com
- Javascript
  - `config.get()` – gets configuration
  - `secrets.get()` – gets secrets



Library Capsules - Basic

- Library capsules provide built in functionality for working with complex concepts e.g. date, times and geography
- To use them - import into your capsule and refer via alias – `as`
- Common library capsules:
  - `viv.time` – date/time functions
  - `viv.profile` – user profile data
  - `viv.geo` – locations
  - `viv.money` – currency and prices
  - `viv.image` - images

```
capsule-imports {
  import (viv.image) {
    as (image)
    version (2.12.0)
  }
}
```

Thank you

Roger Kibbe | Bixby Developer Evangelist | Twitter: @rogerkibbe

202

Intermediate Bixby: Dialog, Layouts and Training

Bixby Developers

Bixby Resources - Reminder

Bixby Homepage
www.bixbydevelopers.com

Bixby on Github
github.com/bixbydevelopers

Bixby Tutorial Videos
bixby.developer.samsung.com/newsroom/

Agenda

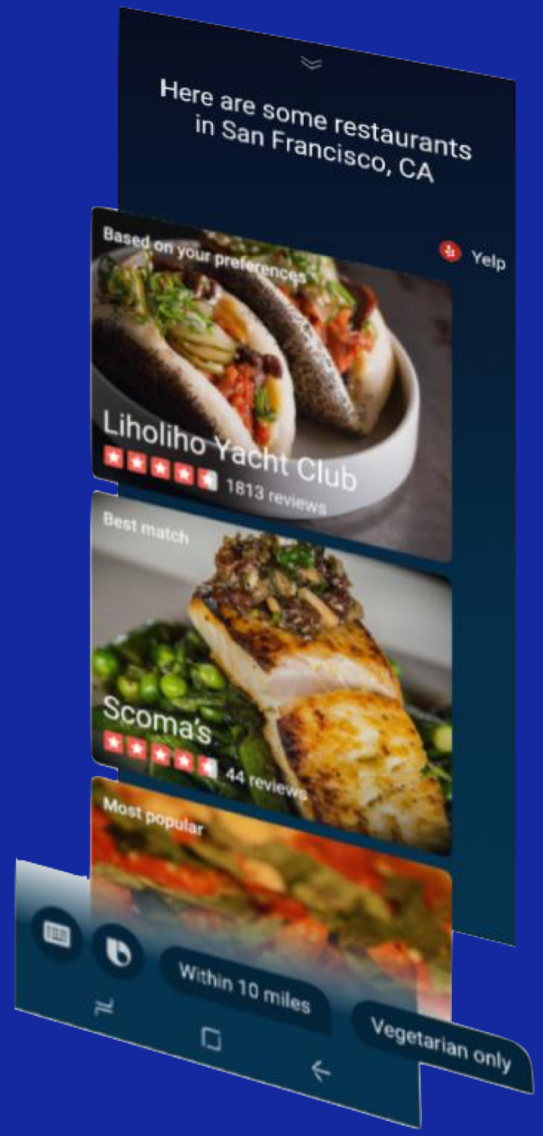
- Dialog
- Match Patterns
- Views
- Layouts
- Conversation Drivers
- Hands Free
- Training

Bixby Output Components

Dialog: Text header/Voice

View: UI

Extras: Follow on actions



Dialog

- Output at top of screen, always spoken.
- Consists of:
  - `event`: dialog event type
  - `match-pattern`: match to concept(s)/action(s)
  - `template`: output

```
dialog (%event%) {
  match: %match-pattern%
  template (%text%)
}
```

Example:

```
dialog (Result) {
  match: Output (output)
  template ("Hello World, I am a dialog")
}
```

Dialog – Event Types

- `Result`: Results information
- `NoResult`: If no results
- `Progress`: In progress information
- `Selection`: User makes selection
- And more – see documentation

```
dialog (Selection) {
  match: Teams (teams)
  template ("Select your team")
}
```

```
dialog (Progress) {
  match: Scores (scores)
  template ("Looking up scores")
}
```

```
dialog (Result) {
  match: Scores (scores)
  template ("${value(scores.team)} scores")
}
```

```
dialog (NoResult) {
  match: Scores (scores)
  template ("I could not find the team")
}
```

Dialog Optional Fragments

- Enclosing fragments in [ ] makes them render only if the variable in them can render (is not null)

```
template ("Here are some[ #{value(input.decade)}][ #{value(input.genre}] movies[ with #{value (input.actor)]")
```

The above dialog could render as:

- Here are some 1980's comedy movies with Eddie Murphy
- Here are some 1980's movies with Eddie Murphy
- Here are some comedy movies with Eddie Murphy
- Here are some movies with Eddie Murphy
- Here are some comedy movies
- . . . And more

Dialog - Plural/Singular

- Common use case: render a different dialog depending upon whether one item or multiple are available
- Using the EL `plural` function, a different dialog can be rendered

```
dialog (Result) {
  match: Car (this)
  switch (plural(this)) {
    case (One) {
      template ("There is one car available")
    } default {
      template ("Which car would you like?")
    }
  }
}
```

Dialog - Macros

- Abstract macro logic to a different file
- Very useful with complex Capsules with lots of similar Dialog

```
template-macro (capOrNot) {
  param (c) {
    expression (city)
  }
  param (s) {
    expression (state)
  }
}
```

```
template-macro-def (capOrNot) {
  params {
    param (c) {
      type (City)
      min (Required)
    }
    param (s) {
      type (State)
    }
  }
  content {
    if (exists(s)) {
      template ("#{value(c)} is the capital of #{value(s)}")
    } else {
      template ("#{value(c)} is not a capital")
    }
  }
}
```

Dialog - Speech

- Default behavior: Template text rendered is also spoken
- Override by the `speech` key. Often spoken text should include additional information from the view.
- Speech useful for no display devices

```
template ("Template text"){
  speech ("This will be spoken by Bixby")
}
```

Match Patterns

- Used to tie things together. Used to determine the right dialog, layout, strategy etc.
- Can match on actions or concepts. Links can create a more elaborate match pattern.
- Example: Match on `FindCapital` action, from input `State` concept and output `StateInfo` concept

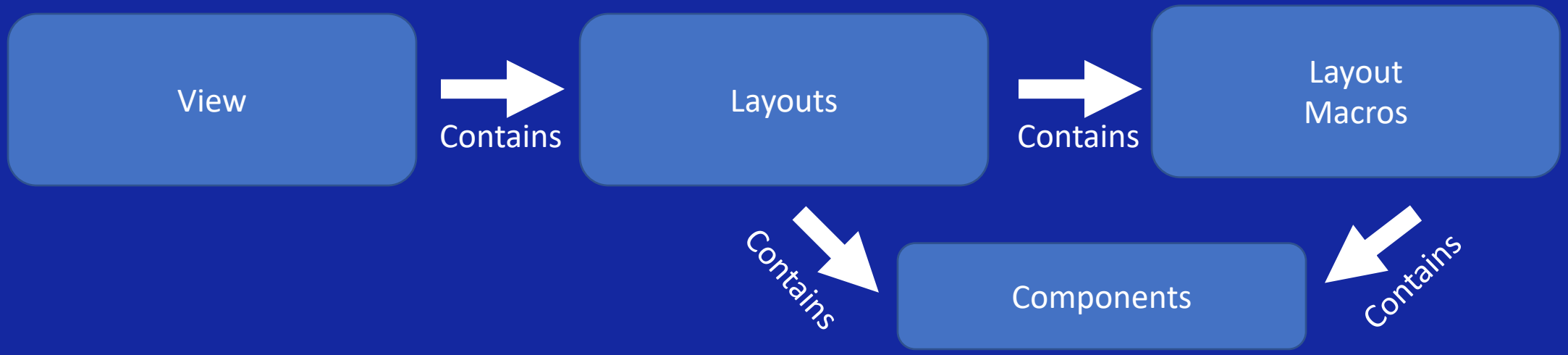
```
dialog (Result) {
  match: FindCapital (findCapital) {
    from-input: State (state)
    to-output: StateInfo (stateInfo)
  }
  template("The capital of ${value(state)} is ${value(stateInfo.capital)}")
}
```

Github - github.com/bixbydevelopers/capsule-samples-collection/tree/master/match-pattern

Views

Views drive the Bixby UI. There are three types of views

1. Input Views: Used for input. Fields, buttons and other input elements typically tied to a concept
2. Confirmation Views: For user approval/review
3. Result Views: UI for Bixby results.



Input Views

- Match on DateInterval concept
- Example: Get a date range
- Uses calendar component

```
input-view {
  match: DateInterval (this)
  render {
    calendar {
      allow-range (true)
      restrictions {
        block-past-dates (true)
      }
    }
  }
}
```

Github Examples: github.com/bixbydevelopers/capsule-samples-collection/tree/master/input-forms

Input Views
Text Field

- Match on State concept
- Message is a Dialog (can use macros/speech etc)
- Form render
  - text-input component
  - on-submit required

```
input-view {
  match {
    State (state) {
      to-input: FindCapital
    }
  }
  message {
    template ("What is the state?")
  }
  render {
    form {
      elements {
        text-input {
          id (state)
          label (State)
          type (State)
          value ("#{raw(state)}")
        }
      }
      on-submit {
        goal: State
        value: viv.core.FormElement(state)
      }
    }
  }
}
```

Github Examples: github.com/bixbydevelopers/capsule-samples-collection/tree/master/input-forms

Result Views

- Shows a state image graphic
- Match on StateInfo concept
- Uses layout and image component

```
result-view {
  match: StateInfo (stateInfo) {
  }
  render {
    layout {
      section {
        content {
          image {
            aspect-ratio (1:1)
            url ("images/#{value(stateInfo.stateGraphic)}")
          }
        }
      }
    }
  }
}
```

Github: github.com/bixbydevelopers/capsule-samples-collection/tree/master/bixby-views

Layouts

- Control the UI. Parts:
  - match: match pattern
  - render: render enclosed layout
  - layout: start layout
  - section: section of layout
  - content: components go here
  - image-card component
- Also includes a conversation driver (covered later)

```
result-view {
  match: Hands (hands) {
  }
  render {
    layout {
      section {
        content {
          image-card {
            aspect-ratio (16:9)
            title-area {
              halign (Start)
              slot1 {
                text {
                  value ("#{value(hands.bixbyHand)} hand")
                  style (Title_M)
                }
              }
            }
            image-url ("images/#{value(hands.bixbyHand)}.png")
          }
        }
      }
    }
  }
  conversation-drivers {
    conversation-driver {
      template ("Play Again")
    }
  }
}
```

Layout Macros

- Reusable layout code
- Very useful for lists of items with small variances

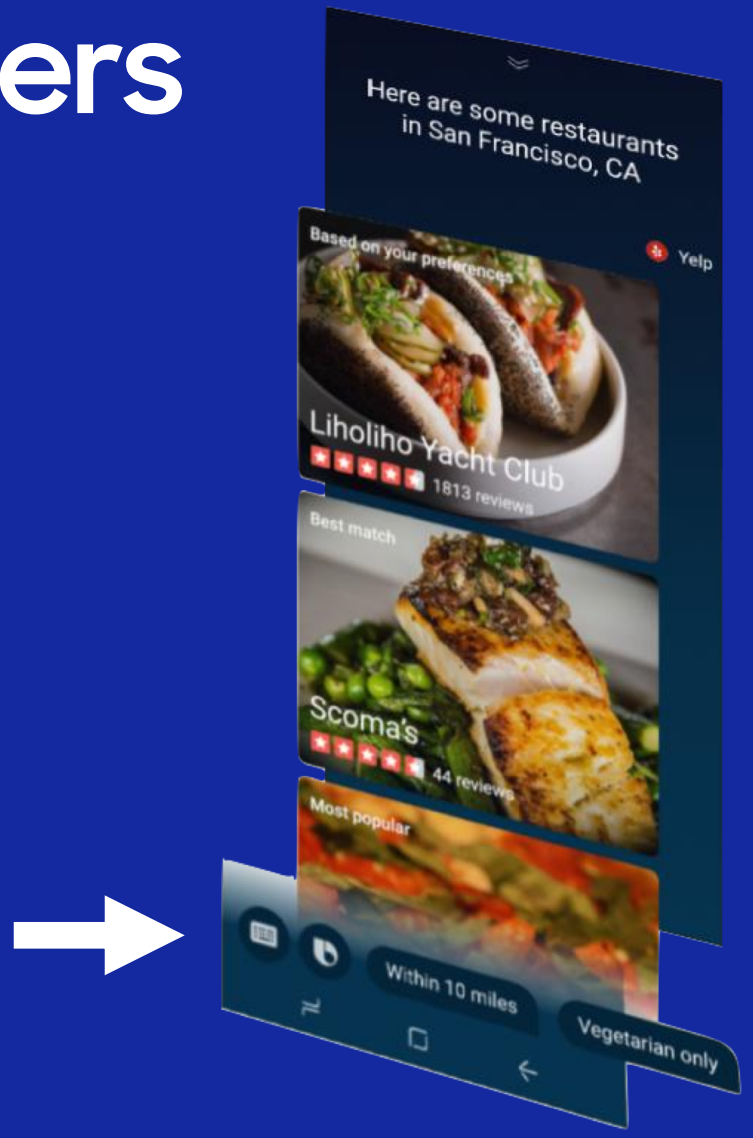
```
result-view {
  match: Hands (hands) {
  }
  render {
    layout {
      section {
        content {
          layout-macro (largeHand) {
            param (hand) {
              expression(hands.bixbyHand)
            }
            param (who) {
              literal ("My")
            }
          }
        }
      }
    }
  }
  conversation-drivers {
    conversation-driver {
      template ("Play Again")
    }
  }
}
```

```
layout-macro-def(largeHand) {
  params {
    param (hand) {
      type (BaseHand)
      min (Required)
      max (One)
    }
    param (who) {
      type (WhichHand)
      min (Required)
      max (One)
    }
  }
  content {
    image-card {
      aspect-ratio (16:9)
      title-area {
        halign (Start)
        slot1 {
          text {
            value ("#{value(who)} hand")
            style (Title_M)
          }
        }
      }
      image-url ("images/#{value(hand)}.png")
    }
  }
}
```

Conversation Drivers

- Shown at bottom of the screen
- Calls to action, next steps
- Template content is a natural language input to training

```
conversation-drivers {
  conversation-driver {
    template ("Play Again")
  }
}
```

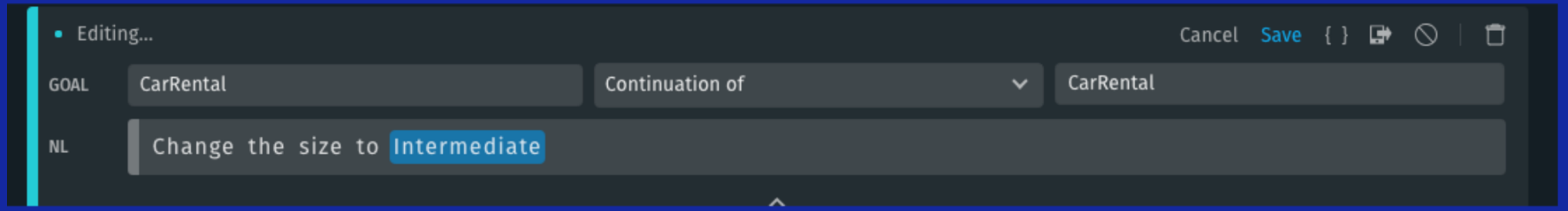


Hands Free List Navigation

- Allow navigation of lists via voice
- Can read and move forward via “Next”
  - List items individually, asking user if they want to go to the next item
  - Pages, asking user if they want to go to the next page
- “Previous” is supported for going back
- “Repeat” repeats reading the list
- Allows ordinal selection via voice e.g. ”first”, “second” or “third”
- Many options – refer to the documentation

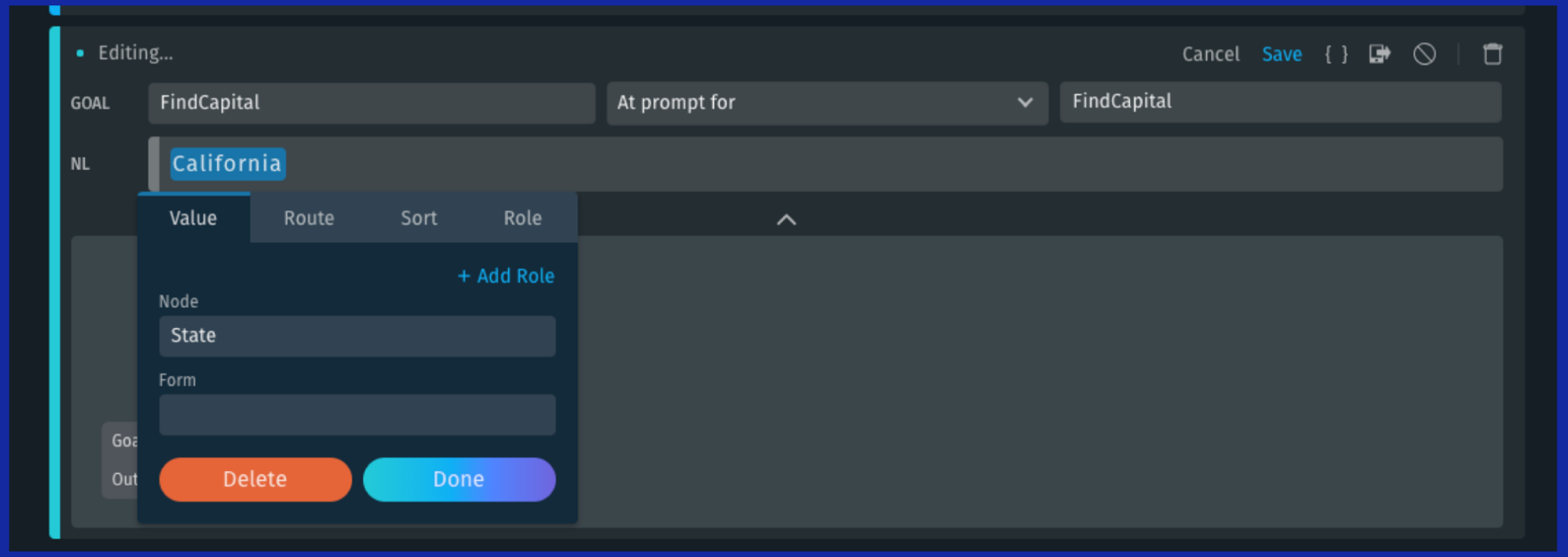
Training - Continuations

- Continuations are follow-up questions or refinements by the user
- Example: ”Rent this car” followed by “Change the size to Intermediate” – the second sentence is a continuation – it only make sense in the context of the previous utterance, not on it’s own
- Set by changing “No Specialization” to “Continuation of” and enter the Action e.g. ”CarRental” in our example



Training – At Prompt For

- At Prompt For training allows natural language input at prompts for input
- Example – “California” after prompted for a State
- Set by changing “No Specialization” to “At Prompt For” and enter the Action e.g. ”FindCapital” in this example



Thank you

Roger Kibbe | Bixby Developer Evangelist | Twitter: @rogerkibbe